Hardware Architectures to Support Low Power Natural I/O Applications

by

Rajeev Krishna

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctorate of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

Doctoral Committee:

Associate Professor Todd M. Austin, Chair
Professor Richard B. Brown
Professor Trevor N. Mudge
Associate Professor Steven K. Reinhardt
Assistant Professsor Scott Mahlke

To my parents and friends, for all of their support.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my adviser, Todd Austin. Without his repeated motivational speeches, inexplicable enthusiasm, and keen insights, this work would likely never have been accomplished, and certainly would not have been as enjoyable. While I don't expect to ever want to tame the Sphinx again, I hope that we can find areas of common interest in the future (though I should probably get on his calendar now).

Secondly, I would like to thank my committee for all of their suggestions and contributions. I would particularly like to thank Scott Mahlke, who dedicated time to this work on a weekly basis. Though we had our share of frustrating arguments, the level of refinement of this document is owed to him.

The last five years would certainly have been maddening without the support of the friends I have made here. I could not have hoped for a more diverse environment, or a more interesting group of future intellectuals. I hope I am better at keeping in touch with them than history would suggest. Of this group, I must single out Jeffrey Cox, who from our first days as TAs for 270, made it his mission to teach me that there is more to life than work, and most of it is video games (and likely delayed my defense by several months in the process).

And finally, to my parents, who have never wavered in their support, what could I possibly say? I swear I'll get a job someday?

I would like to acknowledge the work of the CMU-Speech group in their development of the Sphinx speech recognition infrastructure used throughout this research. This work is supported under the DARPA/MARCO Gigascale Silicon Research Center. Additional support was provided by the National Science Foundation, grant number CSA-0310511. Substantial equipment support was provided by the Intel Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

APPENDIX

CHAPTER 1

Introduction

With advancements in computing performance and availability, software applications are being enhanced with an ever increasing set of features. Examples of this trend are all around. Tasks and conveniences from 'font smoothing' to 'spell checking as you type' and any number of others are made possible by the extra computational power available in modern processor designs. Much of this extra computational power, however, is directed entirely at user experience. The afore mentioned examples, as well as others ranging from advanced graphical interfaces to 'natural language' help systems, serve to emphasize the importance of this area of computation.

The recent proliferation and current pervasiveness of computing platforms brings the issue of user experience and interaction ever more to the forefront of modern system design. This is particularly true from the standpoint of computer-to-human interfaces and interactions. As we steadily approach the holy grail of ubiquitous computing, the old guard of computer interfaces represented by keyboards, mice, and monitors, quickly reach the limits of their usefulness. As computing proliferation and supporting technological advances brought an end to the days of punch-cards, and room–sized mainframes, so too do they now herald the end of modern interface devices.

This realization, and the increasing availability of computational power, has lead to extensive research into more natural forms of computer interface design. These interfaces, generally classified as "natural I/O" represent a domain of applications that deal with

human computer interactions through methods thus far only found in human-human interactions. This domain includes such applications as speech/text recognition (recognizing the spoken/written word at a syntactic level), natural language processing (recognizing language constructs at a semantic level), natural language generation (constructing a natural language response to a query), and computer vision.

The challenge of such interfaces comes from the subtle and inexact nature of human–human communication, and the very natural estimation and context utilization abilities the human mind uses to accommodate them. Consider the specific example of speech recognition in an environment where the inner processes of the mind are laid bare: trying to understand someone in a loud and crowded restaurant. In such an environment, where our natural abilities are tested, the mind's use of context, estimation, and even extra sense information such as visual knowledge of the speaker's mouth movements, becomes plain. In normal communication, these processing constraints are not an issue, and the subtleties of communication serve to add meaning and enrichment to our interactions.

In the discrete world of computing systems, however, these individual variations and the absence of strict standards creates a problem. Applications in the natural I/O domain must make use of advances in modeling and evaluation of stochastic processes in order to accommodate these variations, resulting in a dramatic increase in program and task complexity. Such applications must also maintain information necessary to the task, such as acoustic and linguistic data for speech recognition, or visual feature information for computer vision. This data can stretch into the hundreds of megabytes depending on the scope and accuracy demands of the task in question, and the input driven nature of data access can lead to poor locality and predictability, placing further demands on the underlying computing platform.

The resulting behavior of these applications is fundamentally different from more common application domains. They demonstrate high computational demand, as well as resource demand characteristics that do not lend themselves well to modern general purpose computers. Capabilities such as speech recognition are, therefore, only just emerging for

high end computing systems. [95, 9, 20]

This level of computational complexity is particularly unfortunate in that it limits natural I/O applications to domains in which existing interface technologies are not, as yet, inadequate. While speech recognition on a high performance desktop machine is a compelling goal, its benefit is mitigated by the fact that standard interface devices are often more than adequate to the tasks. Indeed, the area of computing in which natural I/O capabilities have the potential for greatest impact is not in high end computing systems, but in low end, low-power, portable systems such as handheld devices. Standard interface devices are essentially incompatible with the portable or embedded nature of such systems, and the ability to perform natural I/O tasks would be boon.

Unfortunately, low power systems are currently incapable of performing natural I/O functions for a couple of particularly relevant reasons. The nature of such devices requires that careful constraints be places on the power consumption of their components. This, in turn, limits the computational capabilities available, putting natural I/O out of reach for the foreseeable future. This is plainly evident in Figure 1.1, which continues with the use of speech recognition as an example, and shows the average number of words per minute that can be processed by modern high–end mobile and low end systems for a moderate complexity, moderate accuracy speech recognition task. In this particular instance, a knowledge base with a vocabulary of around 10,000 words, and estimate the recognition accuracy to be around 70-80% was employed. To provide perspective, commercial systems may have vocabularies ranging from 30,000 to as many as 120,000 words, and demand far higher recognition accuracy. This graph is further annotated with the time each processor could operate on a single "AA" battery. It is clear from this figure that while high end microprocessors are able to achieve recognition rates comparable to normal speech patterns, the power consumption of these processors excludes them from low power domains. By contrast, the SA-1110 processor (currently used in the Compaq IPAQ), simply does not have the computational capability to provide real-time speech recognition. The XScale, Intel's next generation of embedded processor architecture, performs somewhat better in this re-

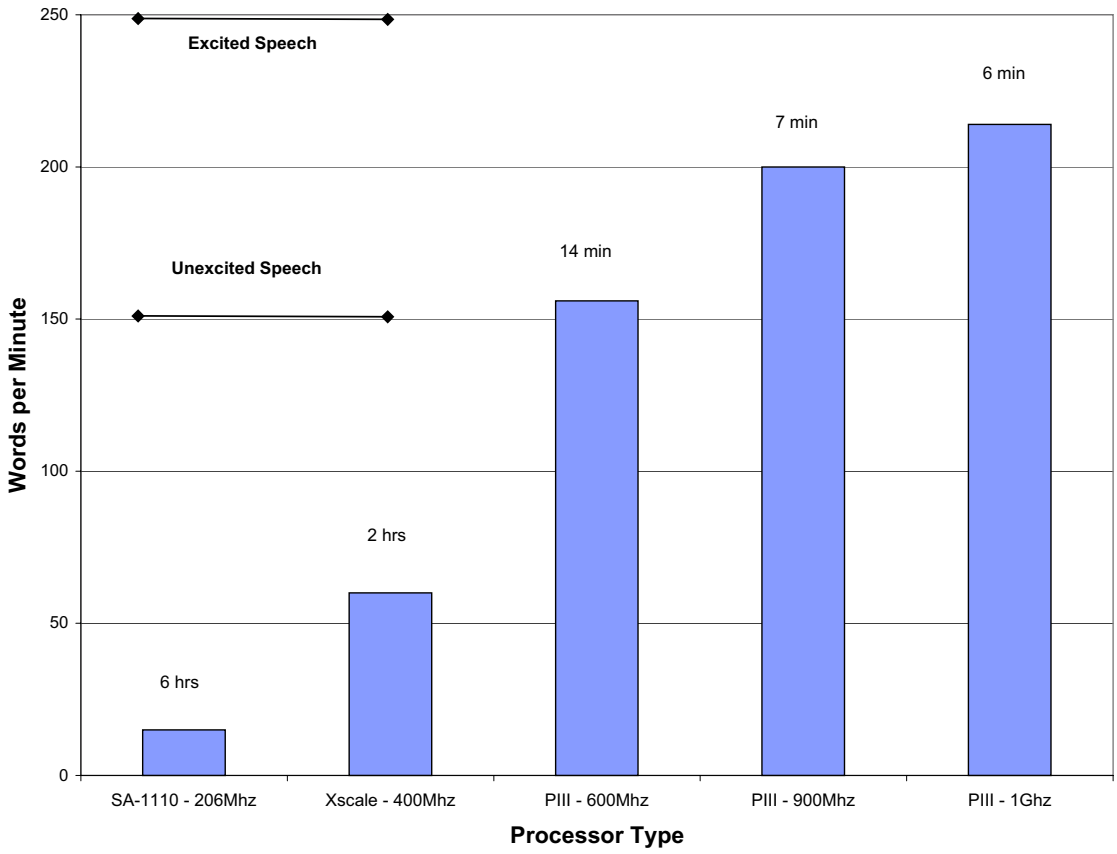


Figure 1.1: Performance of Speech Recognition on Selected Architectures - This chart shows the achievable speech recognition rate (in words per minute) on a set of modern architectures. The SA-1110 and XScale models represent current and future embedded system processor types, while the PIII architectures represent current mobile platforms. In order to consider power consumption, the annotations on each bar represent the amount of time each processor could run on a single "AA" battery. It is important to note that these samples are at fairly low accuracy (70-80%). Higher accuracy demands would correspondingly increase computational demands.

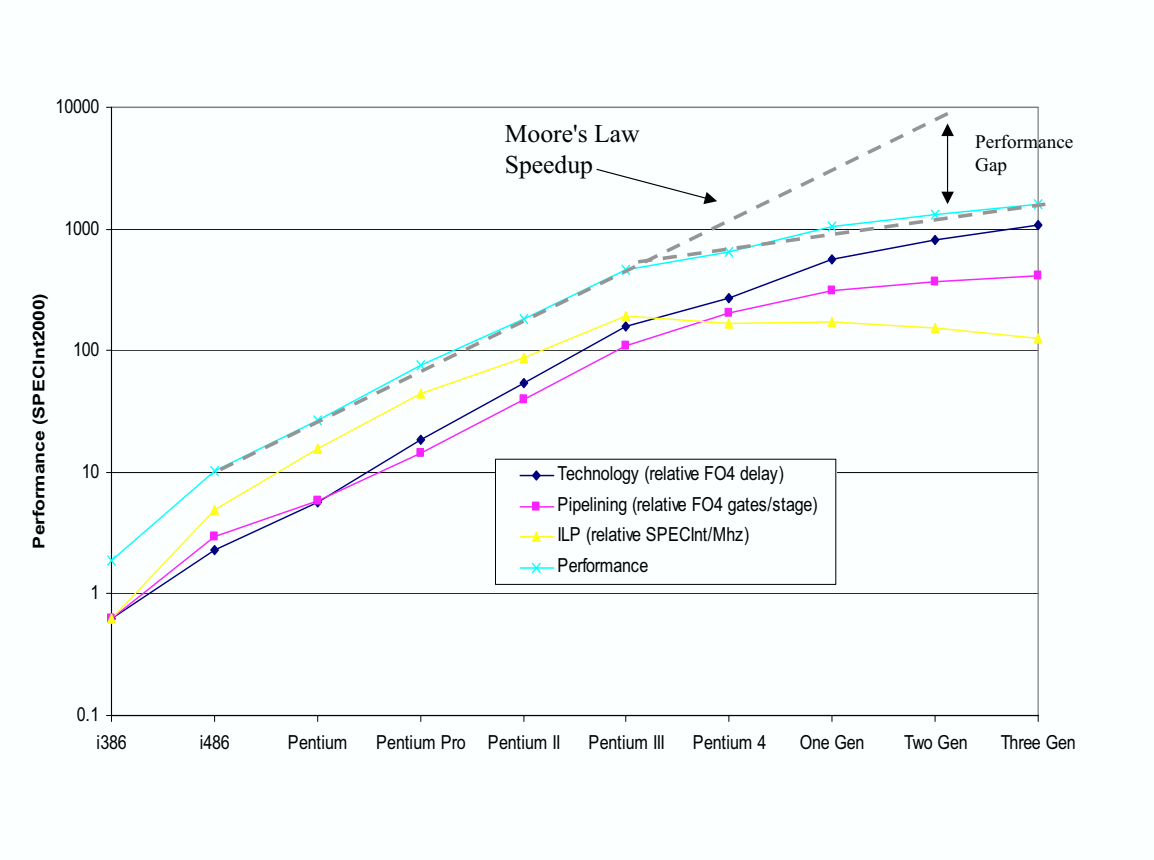


Figure 1.2: Technological Trends in Processor Performance - This figure shows technological trends for a number of previous generation processors, projecting out to three future generations. The loss of return from standard performance enhancement techniques is clearly visible, heralding the need for different approaches.

gard, but we observe that its power consumption really push it into the domain of appliance level processors rather than portable / hand-held processors. In considering these power estimates, it is important to note that we consider only the processor, not the full system. As such, these running times are truly best-case. Given the relatively slow rate of battery lifetime improvements (approximately 5% per year), and the ever present desire for higher accuracy and larger vocabularies, the constraints on low power systems are not likely to be alleviated much in the foreseeable future.

The most obvious response to this analysis of low-end systems is that, while battery technology may not solve the problem, surely processing technology will. The unfortunate truth, however, is that our ability to extract performance from standard architectural and

processing techniques has waned over the recent years. This is shown in Figure 1.2, which depicts overall performance, and the contributions made by various standard technologies. It is extended out to a few future generations based on data from the ITRS (International Technology Roadmap for Semiconductors). A further key observation is that this graph depicts high end performance. Bringing similar performance to low end systems, particularly in the range of hand-held systems, is a much more difficult problem. As such, we must begin to explore other approaches to achieving the desired performance within given constraints.

1.1 Research Summary and Key Contributions

The usefulness of natural I/O programs and the divergence in characteristic behavior between these programs and the more general class of applications makes this a prime target for domain specific optimizations. A number of previous works have demonstrated that targeting a high level domain of applications with common properties can provide dramatic improvements in performance without the cost overhead of application specific solutions [29, 100, 24, 8]. This work will seek to apply such techniques to architectural designs that exploit the unique characteristics of natural I/O applications in a cost and power conscious manner. We focus on the specific task of speech recognition due to the availability of high quality research systems, and the obvious and immediate usefulness of speech recognition in the low–power portable domain.

Detailed Characterization of Speech Processing

This work begins with a detailed characterization of a sample speech recognition infrastructure (the CMU-Sphinx2 speech recognition engine). This characterization focuses on memory system performance, which is identified to be a key bottleneck in this application domain. It is found that, due to the streaming nature of program references to vast amounts of knowledge base data, speech recognition often demonstrates far poorer performance than even the most memory intensive Spec2000 benchmarks (as low as 20 instructions per cache

miss versus several hundred for benchmarks such as crafty and gcc on similar cache configurations). This poor locality violates many of the basic assumptions inherent in the design of high performance processors, and simple hardware based prefetching schemes are found not to be a useful solution. The probabilistic nature of the speech recognition task, however, presents enormous opportunities for thread level concurrency, making this the focus of optimization efforts in the remainder of this work.

Application Specific Parallel Programming Model

In order to exploit this concurrency potential, it must first be exposed to the runtime environment. Software thread management incurs far to much overhead to manage the fine grain concurrency potential of this domain, necessitating alternative approaches. Recognizing certain simplifications possible due to the nature of the computations performed, this work develops a programming model in which the application designer exposes all available concurrency to the underlying architecture, utilizing a combination of techniques such as initial workload partitioning and low–overhead, fine–grain mutual exclusion facilities to notify the architecture of concurrency hazards. This, by intent, allows the architecture to make decisions about exploited concurrency, allowing the system to tune dynamically to current capabilities.

Architectures for Exploiting Thread Level Concurrency

Following this programming philosophy, this work develops a parallel architecture to support speech recognition performance demands. We employ a hybrid multi–threaded, chip–multiprocessor design intended to mask long system latencies through hardware thread switching and maximize performance through concurrent execution. Each processing component of this design is trimmed down to the essential computational functions necessary for most operations in this domain, minimizing the energy consumption of added hardware resources. We incorporate hardware level support for concurrency management and fine grain mutual exclusion, allowing this architecture to maximize processor utilization even

in the presence of significant system latency. This approach is found to be quite effective when presented with parallelized speech recognition code, achieving near ideal speedups for added hardware resources.

Memory System Designs to Support Application Demand

Unfortunately, inclusion of a realistic memory system with constrained bandwidth (as opposed to just large latency), is found to severely limit achievable performance. We therefore begin an exploration of memory system architectures, investigating how program characteristics are altered by our parallel architecture and how bandwidth demand can be tolerated. Despite the streaming nature of access to large blocks of program knowledge base data, high locality is observed in references to program metadata. Furthermore, our parallel architecture creates a degree of "artificial locality" in knowledge base reference data as well, due to simultaneous evaluation of multiple knowledge base elements. Consequently, a multi–level cache hierarchy with an L2 cache size designed to match program metadata and current evaluation data size leads to substantial reductions in memory bandwidth demand. As the overall size of this data component is unlikely to vary as dramatically as knowledge base data in general, this is a particularly promising result. While the computation necessary for "realtime" performance can vary substantially with the specific speech recognizer, knowledge base, and configuration parameters used, this L2 based system is able to achieve realtime on moderately sized evaluation workloads. As part of this memory design exploration, we determine that use of flash and ROM components to store static knowledge base data with lower power consumption, and use of advanced fabrication technologies such as on–chip embedded DRAM are all feasible and potentially useful approaches to the memory bottleneck problem in the future.

Domain Specific Memory and Control Optimizations

This work concludes by evaluating a number of potential optimizations to the base model. First, the use of more sophisticated thread scheduling is considered, taking ad-

vantage of a property inherent in our programming model that allows the architecture to dynamically vary the amount of concurrency exploited based on run–time constraints. While such concurrency throttling is is found to be quite effective at maintaining optimal performance levels, we conclude that it is better to tune system resources beforehand if possible than provide extra resources that often go unutilized.

Second, the potential use of compressed knowledge base data is considered, hiding decompression time in the other latency tolerating mechanisms of this architecture. As knowledge base size, and the hardware required to store knowledge base data, could have a significant impact on overall implementation and monetary cost as well as overall energy dissipation, the benefits of a substantially smaller physical knowledge base without corresponding loss of accuracy are numerous. We find that compression in memory, transparent to the processor itself, can be performed with no significant impact on performance. Due to the access characteristics of the application, however, techniques such as compressed caches are ineffective at improving performance, and only place unnecessary constraints on the sophistication of the compression algorithm used.

Dynamic Power Management Strategies for Speech Processing

Finally, we consider power management techniques in this domain, evaluating low power standby modes versus dynamic frequency and voltage scaling, and briefly consider the implications of processing technology on that tradeoff. It is found that, within the scope of this evaluation framework, scaling voltage and frequency to match realtime constraints achieves substantially higher energy savings than entering a low–power standby state during idle cycles. Analysis suggests, however, that at smaller processing technologies with increased leakage, it will be better to finish processing quickly and enter a low–power, low–leakage state.

In conclusion, this approach to providing realtime speech recognition is quite effective at meeting application demands as well as the energy constraints of low–power systems. Within the scope of the selected moderate complexity recognition tasks, a fairly small addition

of processing resources (2–8 processing elements with 2–4 contexts each), combined with multi–level caching and tuned memory system design is found to meet or exceed realtime performance, leaving considerable room for expansion as the field of speech recognition itself develops and problem complexity grows.

This thesis will begin in Chapter 2 by considering background material on probabilistic search theory, speech recognition in general, and the CMU-Sphinx recognition engine in particular. This background should provide much of the intuition behind our design decisions and clarify subsequent evaluations. We then present our detailed analysis of speech recognition, considering memory, performance, and concurrency issues is Chapter 3. Chapter 4 will re–analyze the program flow of Sphinx, considering regions that may be parallelized, and present initial results of concurrent execution using standard techniques. Chapter 5 proceeds through two revisions of our architecture, arriving at the system and programming model discussed here. The parallel performance and architectural bottlenecks of this system are considered in Chapter 6, followed by a detailed memory space exploration in Chapter 7. Finally, we consider some extended optimizations such as concurrency throttling, compression, and power management in Chapter 8 before concluding with a few general views of the result space and suggesting future directions in Chapter 9. Details of our evaluation infrastructure and power estimation framework are discussed in appendices.

CHAPTER 2

Background Information

We begin this work by considering some background information on probabilistic search theory in general, and speech recognition theory in particular. Section 2.1 will present an overview of the problems inherent in performing search in the presence of uncertainty in input and model data. We present some common solutions and heuristics used to deal with these challenges. Section 2.2 narrows the scope of this exploration to the domain of speech recognition, briefly considering the individual components that are common to most continuous speech recognition systems, and demonstrating where the probabilistic models of the previous section come into play. Finally, in section 2.3 we will present an overview of the CMU-Sphinx speech recognition engine, which is the speech recognition system utilized for performance evaluation in this work. The goal of this chapter is to provide the foundational intuition for the architectural models discussed in this work, and set the stage for the analysis that follows.

2.1 Probabilistic Search Theory

Over the years, numerous techniques have been developed to support efficient searching of well bounded data. Combinations of sorting and searching techniques such as hash tables and red-black trees provide efficient means of representing, maintaining and accessing such data [23]. Due to the input dependent, abstract nature of human communication,

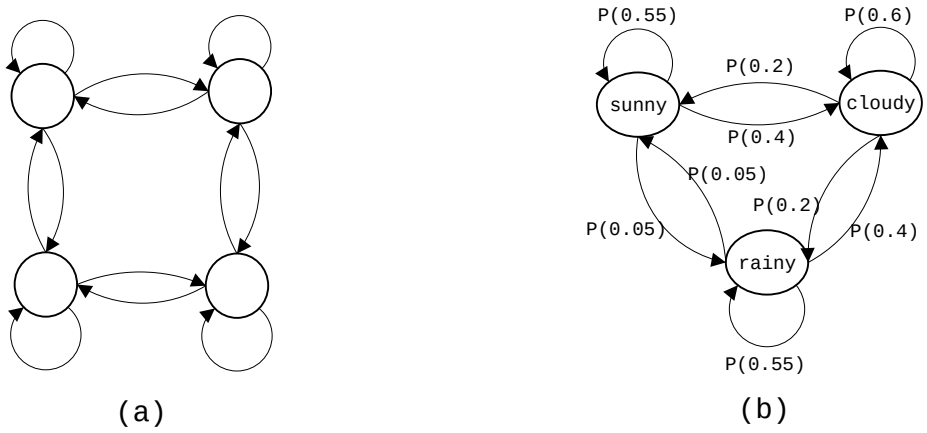


Figure 2.1: Sample Markov Network - (a) This figure depicts the overall structure of a Markov network. Note the interconnected network of transition probabilities. (b) A simple example of a Markov model to describe weather patterns.

however, natural I/O applications are often unable to take advantage of such techniques, depending instead on probabilistic models and associated search algorithms. Available data representations and algorithms for such models are far less prevalent. In general, probabilistic search is achieved by traversal of derivatives of Markov Chains, with one of the more common and useful approaches being the hidden Markov Model. This section will begin by describing Markov models and proceed to a description of hidden Markov Models and their applications to natural I/O tasks, particularly speech recognition. We will continue with descriptions of search techniques used to explore such stochastic models. These are generally based off of standard breath-first search, but include a number of dynamic programming elements to support the probabilistic nature of the underlying data. It should be noted that the techniques described here are well known and many of the examples are taken from cited works [97, 23, 75, 56].

2.1.1 Data Representation: Markov Models and Derivatives

Many stochastic search and NP-Complete class problems do not utilize a fixed data representation format, but rather have unique data models depending on the problem class. For example, satisfiability solvers have no particular need for persistent knowledge base data, requiring instead only the description of the current problem. Many useful applications

of stochastic search, however, do require persistent underlying data, or utilize common conceptual constructs to guide their search. This is often true when these probabilistic systems are attempting to model real-world characteristics such as in computer vision, speech recognition and other natural I/O applications. This modeling often introduces considerable ambiguity in the search space itself. By contrast, problems such as satisfiability or the traveling salesman have no ambiguity of state space, just a vast amount of it. Put another way, given an instance of TSP, there is no ambiguity during the search as to whether a particular path has covered a particular city or not. In speech recognition, however, there is often considerable ambiguity in whether a particular element of sound was actually 'heard' or not.

The Markov model in a fundamental approach to describing such probabilistic systems. These models, and their derivative forms, are therefore commonly used to structure the persistent data required by many of the afore mentioned problems. A simple example of a Markov model is shown in Figure 2.1a. Each 'node' in this model represents a single observable system state. The transitions between nodes represents the probability of transitioning to the specified new state, given the state the system is currently in. If, for example, this chain were used to represent daily weather, the nodes may be considered to represent states such as 'sunny' or 'rainy' (as depicted in Figure 2.1b). The probability of transitioning from a 'sunny' state directly to a 'rainy' state in this example is 0.05. Note, however, that the probability of transitioning from 'sunny' to 'rainy' *through* 'cloudy' is higher (0.08). This demonstrates how the product of transition probabilities over any path through the network can be used to determine the probability of seeing any given sequence of weather conditions in order, and the nature of such paths can account for probabilistic ordering constraints (it is more likely to rain once cloudy than when sunny). If we assume that the 'alphabet' of observable states (or 'symbols') is properly representative of a desired domain, it should be clear how such a model can be used to determine the likelihood of a particular set of observations.

While Markov models and their most direct derivatives have a number of useful ap-

plications, they do not have enough flexibility to describe many of the more interesting modeling scenarios. This problem can arise due to a number of factors. For example, a simple Markov model requires that each node be an observable system state that remains constant. No allowance is made for uncertainty in the observable result of a particular node. In an environment such as speech recognition, variations in speaker intonations and vocal patterns make such a degree of certainty impossible to achieve. Thus, it is necessary to expand the model to make allowances for such points of uncertainty.

The solution in this case is to abstract the observable state of the system through one more level of probability, producing a 'hidden' Markov model. The Hidden Markov Model (HMM) [74] is a state modeling system that has proven exceptionally useful in describing complex probabilistic systems. While originally applied to speech recognition tasks, this modeling format has quickly gained increasing popularity in areas ranging from data mining [58, 87], to computer vision, to gene / protein sequence analysis [90, 19, 38, 51, 52, 13] and fingerprint identification [86]. Much of this popularity comes from the HMM's ability to characterize these complex systems in a mathematically tractable way.

The HMM is derived from standard Markov models by a fairly straightforward modification that can initially be difficult to grasp conceptually. Instead of representing a single state, each node is made to represent a distribution describing the probability of that node emitting any observable symbol in the alphabet. While a sequence of observations can still be made visible at the end of a search, the sequence of states that produced them is *hidden* behind this extra level of indirection. More formally, the underlying stochastic process described by the model (for example, the weather) in not directly observable, but rather can only be observed through another set of stochastic processes which generate the set of observations.

Hidden Coin Tosses

To help further clarify the concept of an HMM, we present a simple example of a coin toss experiment [75]. Consider a scenario in which you are collecting the results of a

standard coin toss experiment by noting which of two lights (heads or tails) is illuminated on a panel during a particular time interval. The lights are controlled by an individual who is not observable. The job of this individual is simply to flip one coin out of some n available to him, and report the results of that flip by activating the corresponding light on the panel. If we assume $n = 1$, then we are able to describe the state of the system by a simple (observable) Markov model. Specifically, the only unknown in this case is the probability of a given state occurring (say, the probability of observing "heads"). This probability corresponds to the bias of the coin involved, and can be used then to compute the probability of occurrence of any arbitrary set of observations.

Now, let us explore the effects of allowing another coin into the mix and setting $n = 2$. This scenario produces a new set of unknowns. Not only is it necessary to know the bias of a single coin, it is also necessary to know the bias of the second coin, and the probability of a given coin being selected on any iteration of the experiment. Recall that the final set of observations is still simply the "heads" or "tails" observable on the lighted panel. It should be immediately obvious that modeling this scenario with a standard Markov model would be difficult at best and prone to unnecessary inaccuracy. The setup, however, naturally lends itself to the hidden Markov model approach. Given our previous description of HMMs, the experiment could best be described as a two state HMM network, in which each state represents a single coin. Thus, the 'probabilistic distribution' associated with each state is the bias of a given coin, and the transition probabilities between states represent the likelihood that a particular coin is selected given the last coin that was used. It is clear that, while a final set of observations is visible, the sequence of states traversed to arrive at a given sequence of observations (that is, the order in which the coins were selected) is essentially hidden from the observer. This example can be extended arbitrarily to any number of coins, with any number of 'sides'.

This, then, is the construct of a hidden Markov model. The extra level of probabilistic indirection provides the necessary descriptive capabilities to model a wide array of stochastic systems from human speech patterns to gene sequence patterns. The methods used to search

through this model produce many of the behavioral features observed in many natural I/O applications.

Most applications of HMMs involve a different sort of analysis than the previous discussion might imply. In the examples presented thus far, we considered how an HMM could be constructed to match the behavior of a given set of observations. The coin toss example is framed almost as if one were simply using the HMM to generate a particular type of random value distribution. Most applications, however, use a pre-built HMM based on historical observations to determine the probability of a new set of observations. For example, in the domain of speech recognition, the HMM is built from known speech patterns, and is used to generate and test a number of possible hypothesis as to what was said in the current instance. The model provides a likelihood that a particular hypothesis is the correct one, with the hypothesis that comes through the search carrying the highest probability value considered the recognition result.

In order to clarify this point, we turn once again to the coin toss example. This time, imagine that we are given a set of observations (heads/tails), and are asked to determine the order in which coins were selected by the unseen coin flipper. This is essentially the same as asking what sequence of phonetic units or 'phonemes' was uttered by a speaker, if the observations represent acoustic feature sets, and the 'coins' (HMM states) represent phonemes. The congruence between our simplified example and actual problems should be clear.

Returning to our example, let us continue to assume only two coins, and thus two states in the HMM. The first observation (say, 'heads') could logically have been generated by either coin (though potentially with different probabilities). The second observation could once again have been produced by either coin (with the same relative probability as the first). Furthermore, it could have been generated by the same coin as the first or by the other. Due to the probabilistic nature of the search, the path through the HMM that has the highest probability will not be known until all observations have been applied. Thus, at this stage, it is necessary to track all four possible path hypothesis. In order to consider

the third observation, it will be necessary to track eight paths, and in order to consider four observations, sixteen paths. In truth, this is a fairly naive approach, and techniques discussed later, as well an numerous pruning techniques, can be used to bound the number of active paths that must be evaluated. Despite such techniques, the sheer number of candidate paths that must be evaluated remains a major source of computational complexity in such evaluating such models.

2.1.2 Search Techniques

As described in the previous section, the nature of many natural I/O applications necessitates the need to search over data that, for various reasons, cannot be well ordered, and sometimes cannot even be well bounded. They may employ any number of standard search techniques from the AI world such as breath/depth first, iterative deepening, A* and the like [23, 81], as well as a number of more domain specific search techniques (such as the Viterbi beam search [74, 94] presented later in this section).

While detailed descriptions of standard search procedures will not be presented here, a key observation to providing architectural optimizations is that many forms of search are essentially the same. Nearly any search can be described as a current state and a queue of 'next' states. The aspect of different search algorithms that distinguish them is how states are inserted into the queue. For example, if all children of the current node are added to the back of the queue, the result is a breath first search. Similarly, if they are added to the front of the queue (in the correct order), the result is a depth first search. While some search methods (such as iterative deepening) do require algorithm specific behaviors, this general viewpoint of search is able to describe the important aspects of most search algorithms.

In many instances, the ability to conform to this pattern indicates a number interesting properties. First of all, the individual 'next state' nodes are very likely to be conceptually independent, having at most a few read only data structures in common (and the queue itself). This strongly suggests that many of these search techniques are amenable to parallelization techniques. In fact, a number of previous works have demonstrated how NP-

Complete and stochastic search problems can be parallelized [14, 83, 76, 26]. Second, even a stochastic search process often can predict with relatively high accuracy which states are going to be accessed in the next few iterations. This predictability is entirely algorithmic and dynamic. It can certainly not be made available through profiling, due to the input / problem dependent nature of many such programs, and may not even show up in the underlying memory reference stream at runtime. This suggests, however, that an efficient method of informed prefetching may help to shift from a bottleneck of memory latency to one of memory bandwidth, particularly on low end systems with simple memory systems.

One particular search technique is very relevant to traversal of hidden Markov Models which, as described previously, have a wide range of applications. Our immediate interest in this algorithm is derived from its applicability to speech recognition systems. This search technique is known as time synchronous Viterbi beam search, and is presented in detail here.

Time Synchronous Viterbi Beam Search: HMM Traversal

Given the description of HMM's presented in the last section, it should be evident that the more common search techniques cannot be applied directly. HMM theory defines three specific operation types that must be performed on these models:

The Evaluation Problem - Given a model and a set of appropriate parameters, determine the probability that it will generate a particular (given) sequence of observations.

The Decoding Problem - Determine the state sequence used by the evaluation phase to arrive at the resulting probability.

The Learning Problem - Optimize/refine the HMM parameters given a set of training data or corrective updates.

For the scope of this work, the first two problems are of particular interest. The third is often done offline in a "compute once, use indefinitely" manner, and as such, discussion of techniques to solve this problem are left to cited works [74, 56]. The discussion of HMM's

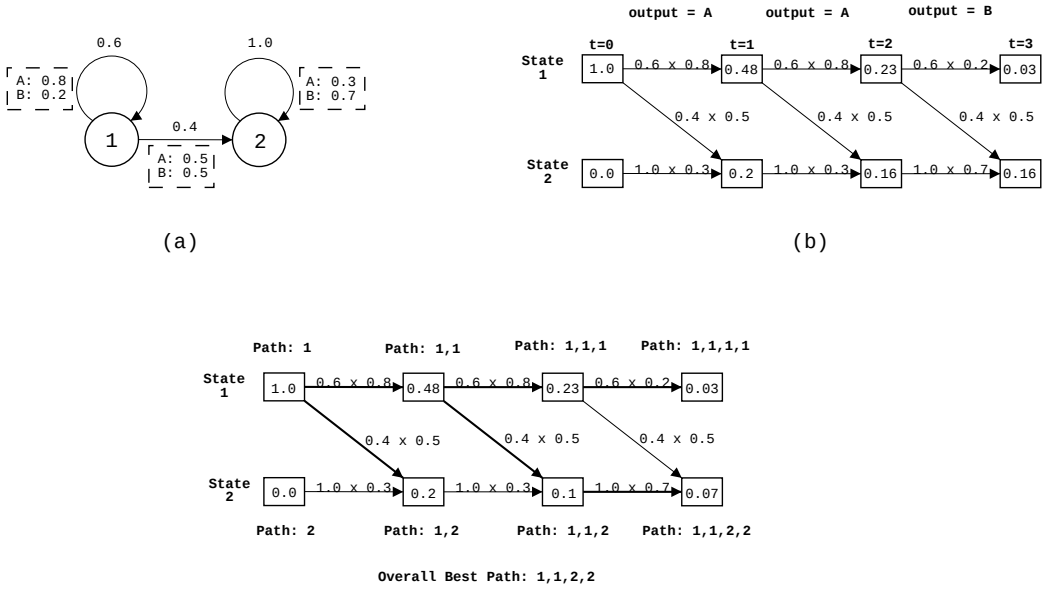


Figure 2.2: HMM Traversal Techniques - (a) A sample HMM which is traversed in b. (b) Depiction of the forward computation on the sample HMM in (a) for the observation sequence "A A B", taken from cited works [56]. The same traversal is shown in (c), this time using the Viterbi algorithm. Best candidate paths are highlighted on each iteration, and the selected path candidate is annotated for each state.

presented in the previous section introduces an approach by which to resolve the the first two (essentially closely related) problems.

The first problem can be addressed by a technique known as the "forward algorithm". Given an observation set S of length T, the algorithm essentially sums the probabilities of all paths of length T that generate S, where the probability of each path is the product of the transition and output probabilities of each step. Remember that the question addressed by the first problem is simply to determine how well the model can be used to describe the observed sequence, not to identify a specific path that matches it. By the very nature of HMMs, such a path is unknowable in anything more than a probabilistic way. Figure 2.2, taken from cited works [56], shows a simple HMM (a) and an illustration of a forward algorithm sweep through that HMM for the observed sequence $A\ A\ B$ (b). Each cell in 2.2b shows the cumulative probability for the given state at a given 'time', where a time iteration corresponds to the availability of the next observation symbol. The algorithm

begins by assigning a value of 1.0 to an initial state, and zero to all others. The output probability of each state for the given observation, as well as the transition probability along the current path, are used to compute the probability associated with that state on the next time cycle. This procedure is repeated until the final observation has been incorporated into the model. The summation of probabilities of all final states represents the overall probability that the given model would generate the given observation sequence. A key observation to make at this point is that the search is *time-synchronous*. That is: all states are recomputed based on a given input symbol before the search proceeds to the next input symbol. It is, in fact, conceptually very similar to a breath first search algorithm.

By maintaining cumulative probabilities at each state and summing the final probabilities, the forward algorithm has essentially explored all paths through the network in a probabilistic manner (without explicitly exploring each individual path, which would be computationally infeasible for any but the smallest problems). The problem that arises, however, is that it is often important to determine a state sequence through the network. This is the second "task" from HMM theory and corresponds to asking for the order of coin selection in our original HMM example. Since an exact solution to this problem is impossible, all that can really be determined is the state sequence that was *most likely* taken in traversing the model. This is done by a simple modification to the forward algorithm. It is now necessary to note the best path that was used to arrive at a given model node for a given time iteration. For lack of an exact solution, this is be considered to be the best path to the current state, concatenated with the best path to the previous state. Put another way, rather than simply summing the probabilities of edges arriving at a state, the algorithm instead selects the highest probability path and adds it to an ongoing list. This technique is know as the Viterbi Algorithm [94]. A very simplified example of this is shown in Figure 2.2c, based on the same HMM and observation sequence used previously.

This algorithm presents some positives and some negatives. One clear benefit is that the number of paths which must be tracked and computed for any given iteration is bounded by the number of states in the model. It should be immediately obvious that no "lower

probability" path entering a particular node can ever achieve a higher overall probability value than the highest probability path entering the node, as all future probability values accumulated into such paths will be identical. Thus, it is never useful to track more than one path at each node at the end of an iteration. The down side, however, is that within an iteration, it is necessary to evaluate all incoming edges of a node (and, consequently, all outgoing edges). Thus, for strongly interconnected networks of thousands to millions of nodes, the computational effort required is still substantial, even if the computational complexity is bounded. A standard solution to this problem is to simply throw away low probability paths. The result is a "beam" of probability values that are carried through on each iteration, with any paths that fall outside of the beam discarded. This is the essence of the time-synchronous Viterbi beam search. It is important to note, however, that discarding low probability paths in such a manner is not the same as discarding lower probability paths entering a node. The paths discarded by the beam search may very well proceed though paths that significantly increase their probability in later stages of the search. Thus, unlike the earlier path elimination, the application of a search threshold does trade computational time for accuracy.

2.2 Speech Recognition

This section attempts to describe the current state of the art in speech recognition algorithms, discussing general themes and functionality necessary for all speech recognition applications. Though we employ CMU–Sphinx as an evaluation infrastructure, our intent is not to produce an architecture for running Sphinx. Thus, while we present the specifics of Sphinx later, this section contains the underlying algorithmic motivation for our work.

Many of these algorithmic components have uses in a wide array of other application domains. For example, the front end processing techniques discussed in this chapter are fundamentally DSP style operations; Probabilistic scoring using Gaussian approximations is common to a wide range of stochastic applications, particularly within the natural I/O domain; and modeling of stochastic processes with hidden Markov Models is a technique

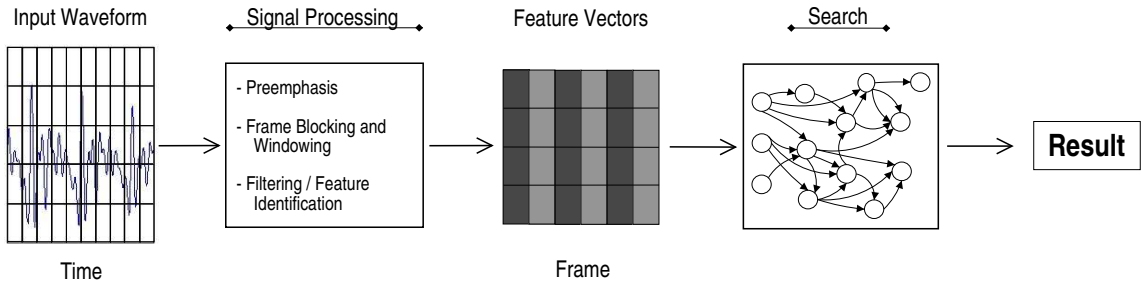


Figure 2.3: Overview of Steps in Speech Recognition - The procedure followed to perform automated speech recognition generally involves processing of the input waveform to minimize noise effects, divide the input into time slices, and generate a set of discreet numeric feature vectors that describe each time-slice. This set of feature vectors is then scored against knowledge base reference data, and the scores are used by the search phase to traverse a linguistic model of human speech patterns, producing the recognition result.

utilized by fields as diverse as computational biology and fingerprint identification. Thus, the potential uses of this work extend beyond the our specific research goal.

Before we begin discussing algorithms, we must recognize that there are a number of variations in speech recognition itself. These variations range from utterance processing to continuous recognition, small to large vocabularies, offline to real-time processing capabilities, and speaker independent to speaker dependent training levels. In this work, we will focus on large vocabulary, speaker independent, continuous, real-time speech recognition, which we believe to be the most generally applicable and interesting form. This eliminates a number of trivial speech recognition techniques (such as waveform matching), which are viable solutions to applications such as utterance recognition. It also expands the scope of the problem, allowing our solution to not only subsume the more narrow speech recognition tasks, but also gain some applicability to other related problem domains.

2.2.1 Overview

As we begin this discussion, it is important to understand the overall process of speech recognition and how the steps involved compare with other applications and domains. Essentially, the process of speech recognition can be divided into three phases: signal processing/feature extraction, feature scoring, and search or hypothesis generation and validation

(Figure 2.3). The first phase can be summarized as a sophisticated A/D conversion process. The goal is to take a continuous, analogue input signal (speech waveforms in this case, but more generally any "natural" input) and convert it into a discrete set of quantified values, extracting relevant and useful information from each discrete unit in the process. The sampling rate is generally chosen to be high enough that the features of interest can be considered constant within the scope of a given sample. The general result of this process is a set of vectors for each discretized unit, where each vector represents the quantified values of a particular signal feature. As front end processing techniques are not the primary focus of this work, we will present only a brief overview with the intent of providing insight into the data models upon which the search process operates. The second and third phases of speech recognition, the back-end scoring and search, are the areas of primary interest in the context of this work. These phases are strongly inter-related, contribute the bulk of the computational demands of speech recognition, and are the source of many of the characteristic behaviors discussed. Their purpose is to probabilistically match the incoming feature stream to acoustic knowledge base reference data, and then use this score to progress through a linguistic model of human speech, producing the recognition result.

2.2.2 Front End Processing

The signal processing and feature extraction phases can essentially be considered preprocessing on input data. In the general sense, all "natural" input applications require a method by which to represent the analog world in a digital format. More importantly, the conversion process must extract the most useful aspects of the incoming input stream, and divide the input stream into small, discreet blocks for reasons that will become clear in the later discussion of the back-end recognition phase. This "scrubbing" of the input serves to reduce the need for later computation and increase accuracy. Though this discussion describes specifically the front end processing techniques commonly used for speech recognition, the need for such front end processing is clearly extensible to other applications as well.

A key observation in the design of a front end processing system that best isolates human speech is that human speech is produced by the human vocal tract and is intended to be heard by human ears. Logically, an approach to signal processing that is intended to work with human speech should focus on the characteristics of the acoustic speech wave that are emphasized by the human vocal and auditory systems. The two most commonly used front end processing techniques for speech recognition address each of these attribute sets in turn. The "Linear Predictive Coding" or "LPC" filter approach generates a set of parameters for a speech signal by considering properties of the human vocal tract. The "Mel Scaled Cepstral" approach generates parameter sets by considering the frequency response characteristics of the human auditory system. In the end, however, both approaches try to represent speech data in the same essential format: as a far more basic input signal passed through a set of easily parameterizable resonance (foraments) filters.

Linear Predictive Coding

The LPC approach to speech parameterization is based on the observation that speech is generated by the human vocal tract, which places some fairly tight constraints on the scope of possible sounds and allows certain optimizations to be considered in representing acoustic data. Essentially, LPC treats an incoming speech waveform as having been generated by a buzzer (vocal cord) at the end of a hollow tube (throat and mouth). This theoretical view is represented for clarity in Figure 2.4. The "buzzer" quantity (essentially an impulse train for vowels and voiced consonants, and white noise for unvoiced consonants) of the sound wave is characterized by its intensity and frequency, while the "hollow tube" can be characterized by its resonants (the formants). An LPC analyzer attempts to predict the formants and discount their effects from the input sample, then characterize the remaining acoustic parameters in terms of an impulse train or white noise. Since speech presents a time varying signal, this is normally performed on small, relatively discreet time slices. The formants for any given time slice, or frame, are estimated by way of a difference function, which expresses each frame as a linear combination of previous frames. The coefficients

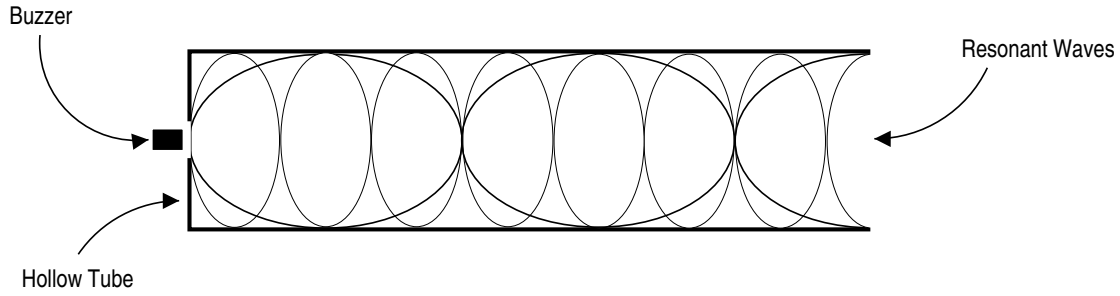


Figure 2.4: Theoretical Basis of LPC Analysis - Linear Predictive Coding is based on a view of the human auditory system as a buzzer at the end of a hollow, lossless tube. This simplification is valid to a first approximation, though characteristics such the inherently lossy nature of the vocal tract, the side-channels produced by the esophagus and nasal passages, and variations in tract curvature and mouthed vocal effects detract from the model's accuracy. The buzzer is represented as an impulse train for voiced sounds, and as a white noise generator for unvoiced sounds. LPC analysis attempts to identify and describe the resonant effects of the tube, isolating these from the (relatively simple) source sound generated by the buzzer.

of the difference function represent the parameterized representation of the formants of a frame, and are selected in order to minimize the mean-square error between the predicted signal and the original signal. A number of techniques are then used to characterize the remaining signal (or residue) in an attempt to account for inaccuracies between the "buzzer and hollow tube" model and the actual human vocal tract which result in residue waveforms that are not easily described by a simple "intensity and frequency" measure. LPC parameterization, however, remains far more accurate on voiced sounds (which match the "buzzer", or impulse train representation) than unvoiced sounds. The result of these computations is a set of parameters associated with each frame of the input signal that provide one view of the features found in that frame.

Mel-Frequency Cepstral Analysis

The 'Mel' is a unit of pitch proposed by Stevens, Volkmann, and Newmann in 1937. To contrast against the Hertz scale, which is essentially a linear scale across the frequency domain, the gradiations of the Mel scale are constructed to match frequency differences identifiable by the human ear. In other words, it is a scale of pitches that a sample of

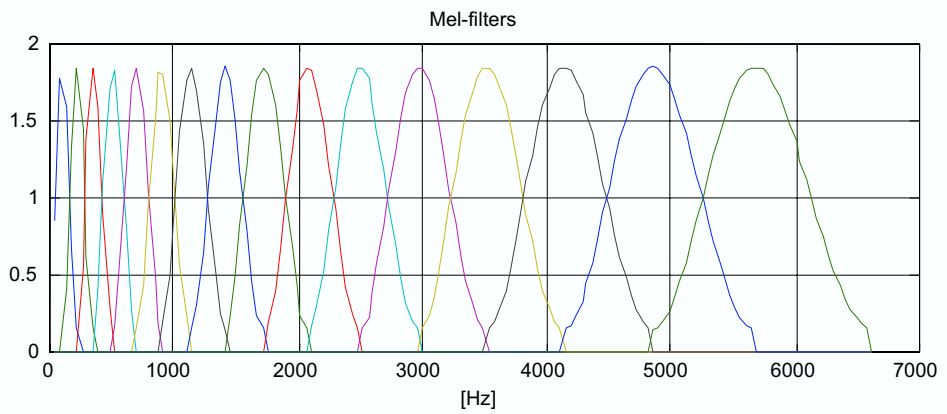


Figure 2.5: Sample Mel-Scaled Filterbank - This simplified filterbank demonstrates the general approach used to generate Mel-scaled data. A key feature of such a filterbank is the progressive increase in bandwidth at higher and higher frequency ranges. This matches the frequency response patterns of the human auditory system, essentially providing higher feature resolution in regions of the frequency domain in which ears are better able to distinguish between frequencies.

listeners have judged to be of equal distance to each other. The two scales essentially coincide below about 500 Hz. Above this value, a single Mel corresponds to larger and larger intervals in Hertz. As the human auditory system basically operates in the frequency domain, the Mel scale provides a much better metric by which to judge which components of a sound wave are most relevant in human hearing.

Mel-frequency cepstral analysis uses a "bank of filters" approach to feature extraction. The basic theory is to pass the input waveform through a set of bandpass filters constructed with center frequencies across the useful audible range. Each filter is then multiplied by the frequency power spectrum of the input waveform (taken, again, at a particular time-slice or frame), producing a single value corresponding to the given filter and frame. The results are considered Mel-scaled when the selection of filter banks in organized around intervals on the Mel scale. A sample of such a filter bank is shown in Figure 2.5 taken from cited work [10]. As a final step, the output from the filters is used to generate a set of cepstral coefficients. Cepstral analysis is a source-filter separation technique (that is, attempting to separate the signal into a source and a set of filter parameters) resulting, once again in a parameter set that emphasizes the formants of the input signal. The final parameters

passed on to the recognition phase are the cepstral coefficients calculated in this process.

While LPC parameterization has historically been the most popular method of feature extraction for speech recognition tasks, Mel-scaled Cepstral analysis has gained popularity in recent years. This is most likely due to its straightforward implementation, and the ability to perform many of the requisite operations (such as fast Fourier transforms) quickly and efficiently. This is the feature extraction technique used in the Sphinx II application model that will be used in this research.

Standard Preprocessing Procedure

While previous sections provide some insight into how feature vectors are constructed, it is important to understand that there are a number of other transformations, options, and processing steps. For example, it is standard practice to perform some pre-emphasis and normalization procedures on the input signal before either feature extraction technique is applied. These serve to remove the DC signal component, and normalize various amplitude features, making later transformations more productive.

The feature vector itself can be based on a number of parameters ranging from band energy, overall energy, changes in features from previous frames (derivatives), and so forth. The goal in this selection process is again to pick a set of features that demonstrates identifiable and useful contrast when presented with distinguished input sets.

2.2.3 Probabilistic Feature Scoring

While the front end DSP processing serves to break up the analogue acoustic signal into discrete chunks with finite values that may be manipulated by digital devices, the actual values are continuous in nature. That is, the feature value produced may be any value within the range of values passed by the filter, and is limited only by the floating point accuracy level of the underlying architecture.

The actual reference values stored in the acoustic knowledge base of a speech recognition program, by contrast, must be somewhat discrete in nature. The storage needed to

capture and fully specify all possible variations of a given sound element would otherwise be prohibitive. As speech recognition is inherently a probabilistic search, most recognition systems model the feature values of sound elements as a single (or small number of) reference value and an easily described probability distribution by which to match the "closeness" of any incoming feature to the reference value. While the choice of training methods used to produce the reference value, and the assumptions regarding the nature of the associated probability distribution vary across speech recognizers, the common goal of the feature scoring phase of computation is to use these values to calculate the probability that a newly seen feature vector matches a given reference set in the knowledge base. This probability score can then be incorporated seamlessly into the remainder of the probabilistic search, allowing the system to accommodate a wide range of input variety without an increase in needed reference data.

In practice, this overall procedure opens the door to a number of other estimation techniques which can reduce the size of reference data or the amount of runtime computation required. One commonly used technique is that of vector quantization, where the probability distributions are essentially pre-computed and stored as a set of discrete codebooks. These codebooks serve at runtime as high speed estimators, mapping feature values within a given range to a fixed, discrete result score. While vector quantization in generally utilized as a lossy compression technique, it has been shown that it can be applied to this particular problem with very little loss of overall recognizer accuracy [36, 50]. Such efforts are useful and important, as this feature scoring phase often accounts for more than half of the overall search phase computation time in modern recognizers [65, 20].

2.2.4 Linguistic Model Evaluation

The elements of the speech recognition process we have discussed thus far are somewhat divorced from the actual task of reconstructing the sequence of spoken words given as input. While each is a necessary step in the process, it is the final phase of speech recognition, the evaluation of the linguistic model, that is most directly responsible for producing a

recognition result. It accepts from the probabilistic scoring phase a set of probabilities for all relevant acoustic references in the knowledge base, and uses this data to step through an internal model (represented, for example, as a HMM) of words and sentences in the target language, exploring all viable possibilities until the entire input sequence has been consumed. Upon full processing of the input, the path through the linguistic model that accumulated the highest overall probability (acoustically and linguistically) is returned as the recognition result.

The fundamental representative unit in speech recognition is the phoneme. A 'phoneme' is simply a basic unit of a phonetic system, such as the velar 'k' sound in the word 'cool'. There is no official standard list of English phonemes, but most common phonemes can be considered essentially standard. While the necessities of higher input sampling rates and recognition fidelity force speech recognizers to operate on sub–phonetic units, a convenient way of thinking about the work of linguistic model evaluation is the sequential identification of phonemes using input frames, and the sequential identification of words using identified phonemes. In this section, we will discuss both the difficulties of making such identifications as inherent in the speech recognition problem, and some historic and current solutions to these problems.

Complexities of Language Modeling

In order to understand the necessity of the complexity that exists in speech modeling and speech recognition systems, it is important to understand the difficulties inherent in the task. We therefore begin this overview by considering a few of the variations in speaking that are easily managed by human auditory and speech processing centers, but present great difficulties for computing systems. These can be considered as falling into two general categories: acoustic variations (time warping and co–articulation) and linguistic variations (word selection and language modeling).

Time Warping and Co–Articulation

At its most basic level, the task of speech recognition consists of nothing more than taking the feature set for each incoming frame, finding the internal model that best matches it, and assembling such data until a list of phonemes can be reported. Unfortunately, a number of features of the speech signal itself makes such a naive analysis nearly worthless. Primary among these features are time warping and co-articulation. Time-warping is nothing more than stretching and shrinking of various portions of a word. For example, a speaker with a southern drawl will stretch out portions of various words that mid-westerners would not. Thus, without very specific training data, a simple "time-frame for time-frame" analysis will fail, requiring that the recognition algorithm have the capability for dynamically matching knowledge base data against (relatively time warped) input data. A number of techniques have been employed to give recognition systems this necessary property, and will be discussed shortly.

Co-articulation effects involve the modification of a particular phoneme by the phonemes that surround it. Due to the short duration of any given phoneme and the fact that the vocal system can not instantaneously switch between the configurations needed to produce consecutive phonemes, the actual speech signal tends to be a blending, transitioning smoothly from the previous sound, through the current sound, an on to the next one. Thus, an exact match to a given phoneme in the knowledge base is rarely possible. While there is no clear solution to this problem, a relatively common technique is to build up a knowledge base with acoustic models not of single phonemes, but of, for example, all useful combinations of two or three phonemes. While this allows better matching with the center phoneme, it also adds complexity to the search process.

Word Selection and Language Models

While the problems of time warping and co-articulation occur at the acoustic level, the difficulties in achieving accurate speech recognition do not end there. Even a system with perfect phoneme identification will generate a poor recognition hypothesis due to word identification effects. Two examples of this are word boundary identification and word

form identification. Word boundary identification simply means identifying when one word ends and the next begins. For example, given the phrase "Their car", an improper word boundary identification could lead to the recognition result "The ear car" which sounds very similar. While, in the general case, this result is obviously wrong to the human observer, an automated system operating on phonetic information does not have sufficient intuitive abilities to recognize the absurdity of the hypothesis. Word form identification presents a very similar problem. Given the same example, the recognition system has no way to distinguish between the words "their" and "there" which sound exactly the same.

The general solution to these problems is to incorporate higher level "language" models into the knowledge base. Such models provide the recognition system with a sense of what combinations of words are or are not likely, and provides some ability to identify absurd constructs. Once again, though, we see the need for a probabilistic assessment, as phrases such as "The Ear Car" while fairly unlikely, are certainly not impossible (in this case, "Ear Car" being a proper noun). This same theoretical approach could be applied to higher and higher abstraction levels, providing not only language information, but semantic information, allowing the recognition system to identify not only absurd word sequences, but absurd sentence sequences as well.

Search Phase Recognition Algorithms

Many of the problems described in the last section present challenges for the search phase of speech processing. Since this phase of computation is responsible for matching incoming feature vector sets with knowledge base information and producing a reasonable recognition hypothesis, it must take into account such factors as co-articulation and time warping, and must attempt to take advantage of higher level language constructs to help refine the recognition result.

Dynamic Time Warping - A First Solution

Dynamic time warping [82] is one solution to some of the difficulties involved in speech

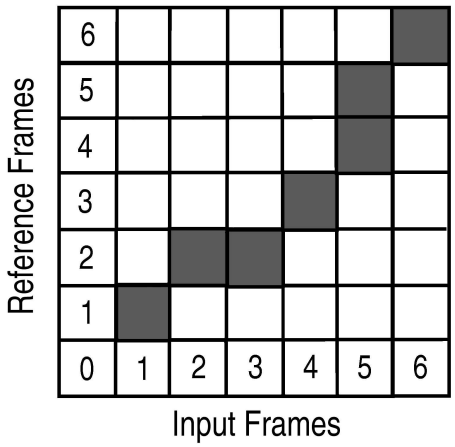


Figure 2.6: Dynamic Time Warping Example - This figure demonstrates the conceptual distance matrix between a six frame input sequence and a six frame knowledge base sequence. Thee "best path" is highlighted. In this example, the first frame of the input best corresponds to the first frame of the knowledge base reference. The second and third frames of the input, however, both best correspond to the second frame of the reference, representing a "stretched" input fragment. Similarly, the fifth frame of the input corresponds best to frames four and five of the reference, representing a "compressed" input fragment. By this approach, the input sequence is matched probabilistically to all possible reference entries in the knowledge base, and the best overall path is selected as the recognition hypothesis.

recognition. Specifically, it is used to help solve the problem of time warping. For the most part, it has been superseded by the use of hidden Markov models, which present a number of advantages in representation of probabilistic state. The concept of dynamic time warping, however, is very straightforward and presents a good lead-in to the somewhat more conceptually difficult hidden-Markov model approach.

The idea behind dynamic time warping is simple: compare all input frames to all knowledge base frames on every cycle, computing an error distance for each combination. The set of pairings that follow chronological constraints and has the lowest error distance represents the recognition hypothesis. Consider a simplified example of a single word input matched to a single knowledge base entry as shown in Figure 2.6. This example visualizes the match between a six frame input sequence and a single six frame knowledge base reference entry. Recall that the problem with time warping involves stretching or shrinking of portions of the input sequence relative to the reference sequence due to variations in speaker accent and

intonation. This relative warping makes a direct frame by frame comparison between the input and the reference set nearly useless. The dynamic time warping approach solves this problem by making no assumptions about which input frames should match which reference frames. Rather, distance vectors are generated for each possible combination of input and reference frame, for all reference frames in the knowledge base. In the figure, this is represented by the entire 6x6 matrix (imagine that each square contains a distance computation between the input frame and reference frame for that square). The "path" through the matrix that matches some logical constraints (eg: the pattern must start at (1,1), move at most one step at a time, and be monotonically increasing) and indicates the lowest total error distance is considered the best candidate path. Given a full knowledge base (as opposed to a single sample), the lowest distance across all reference patterns is considered the best recognition hypothesis. A hypothetical best path in the figure is represented by the shaded entries, and show how dynamic time warping can accommodate for relative warping of the input sequence. In this example, the second and third frames of the input sequence both best match with the second frame of the reference sequence, representing a stretched input sequence. Correspondingly, the speaker went through reference frames four, and five much faster than the reference sample, causing frame five of the input to best match both reference frames.

It should be clear from this example that dynamic time warping does achieve the desired goal of accommodating relatively stretched or shrunk input data. A number of obvious problems with the approach should also become immediately evident. Greatest among these is the inherent need in the algorithm to perform a comparison with, at best, a very large portion of the data in the knowledge base (to perform distance calculations) on each frame. Search pruning techniques can be brought to bear on this problem, but without maintaining a set of candidate paths during the distance computation process, it becomes very unclear how to prune the search space without first performing the search. Other problems arise with how to include higher level language information into the search process, and how to train the enormous set of reference samples necessary in order to recognize even a fairly

basic vocabulary. As we will see, the more recent approach of hidden Markov models seeks to alleviate many of these problems.

Advanced Probabilistic State with HMMs

As mentioned during their introduction, one of the earliest applications of hidden Markov models was in the speech recognition domain. This modeling format is essentially a replacement for the dynamic time warping approach, and the added flexibility it brings has proven very useful is describing linguistic speech data. While there are a number of variations on how speech data is represented in a HMM, the general approach treats each node of the HMM as one frame (10 ms) of input. These frames are generated in front end processing, and the model data stored in the HMM corresponds to a specific acoustic model element. As such, the score generated by the probabilistic scoring phase may be directly applied to current search paths through a given node, modifying the probability of future paths with the match information. The interconnection between nodes represent probabilistic interactions between sound fragments in the language of interest (for example, the probability that two phonemes are found sequentially in a given word). The stochastic process associated with each node (or more commonly, with a group of nodes) essentially describe the likelihood that a given language construct (say, phoneme) would be generated by that node.

Knowledge base data represented in HMM format provides a number of straightforward solutions to some of the general problems of speech recognition. Consider the problem of time warping. The solution to time warping when traversing a HMM is simply to associate the appropriate probabilities to the cyclic edge leading from a given node back to itself. Considering the search methods on HMMs discussed in the previous chapter, the effect of such cyclic edges is to place the current node (and the acoustic properties associated with it) back on the candidate list until the acoustic match with other nodes becomes more likely. In other words, the search process has the ability to 'sit' on a given node for as many (or as few) frames of the input data as necessary. This solution is not only comparable to the dynamic time warping approach, but also has the potential to consume less memory and

provide a more detailed model infrastructure.

The problem of co-articulation presents a slightly different challenge. In this case, a common solution is to treat a given region of the model as representing not a single phonetic unit, but a set (often three, but more generally N). Thus, while the first and last phonemes do not model co-articulation effects, all of the internal phonemes do. The downside of this approach is that the number of HMM states must increase to accommodate all combinations of N phonemes, instead of just all single phonemes.

2.3 CMU-Sphinx Speech Recognition Engine

Sphinx is a speech recognition library developed at Carnegie Mellon University [36, 55, 56]. It provides a framework for speaker independent recognition, meaning it does not require extensive training against an individual speaker before use. As the recognition engine itself is configured as a library, it is possible to envelope Sphinx in a number of wrapper programs that provide everything from pre-recorded utterance recognition to 'online' continuous speech recognition.

2.3.1 Data Representation Summary

The knowledge base data required by Sphinx to perform speech recognition can be divided into two components: acoustic model data, and linguistic model data. The acoustic model data is used in the Sphinx implementation of probabilistic scoring (in this case, a Gaussian scoring system), and the linguistic model data (along with the result of Gaussian scoring) are utilized by the linguistic evaluation phase (a viterbi beam search over a linguistic HMM).

Acoustic Model Data

The acoustic model data utilized by Sphinx is contained in a number of data elements. Sphinx–2 uses the vector quantization technique previously discussed to reduce the data size and runtime computation required during Gaussian scoring. As such, Sphinx utilizes

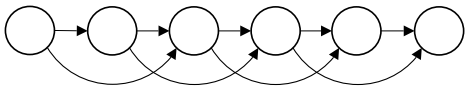


Figure 2.7: Baseline Sphinx Phonetic HMM Model

semi-continuous Gaussian probability distribution codebooks. In this case, the codebook for each required distribution is limited to 256 entries. All told, this represents around 8–10 MB of runtime data, and is provided with the Sphinx distribution. It is not modified at runtime.

The other major component of acoustic model data needed by Sphinx is the set of actual acoustically trained models for sound components in the language. In order to handle co–articulation effects, the Sphinx knowledge base is trained on all high probability combinations of two phonemes (called "senones"). There are 23355 such senones in the model data used in this research. Each senone is given a unique numeric ID number, and the acoustic feature and Gaussian distribution data for each senone are stored in an array of such information under this unique ID number. This trained model data (hereby referred to at static model data or SMD data), is also provided with the distribution, is also not modified at runtime, and represents about 5 MB of the runtime knowledge base.

Linguistic Model Data

Linguistic model data is made up of two conceptual components. The first is the phonetic modeling data used to build HMMs for each individual word in the knowledge base. The second is linguistic data used to describe interactions between words in the language.

Phonetic HMM Model

The Sphinx–2 distribution includes a set of pretrained 7-state HMM model states for phonemes of the English language. In the interests of simplicity, all of these HMMs are identical in structure, as shown in Figure 2.7. By using such low level constructs as the base of the linguistic model, it is possible to train each unit against a large number of speakers

```
A               AX
KIND            K AY N D
GENTLE          JH EH N T AX L
GENTLE(2)       JH EH N AX L
TO              T UW
FAULT           F AO L T
SOME            S AH M
KEEPS           K IY P S
GETTING         G EH DX IX NG
```

Figure 2.8: Excerpt from Sphinx Dictionary

without unreasonable effort. In this format, however, no specific training is performed with higher level language construct such as words. Rather, words are described as sequences of phonemes in a dictionary file, an excerpt of which is shown in Figure 2.8. During initialization, the phoneme sequences described in this dictionary file are used to generate a HMM network for each word in the vocabulary.

Each HMM node is represented as a structure containing path references (describing the paths of the single phone HMM model shown in figure 2.7), current score information along each path / node element, and pointers to the next HMM node, to the SMD entry containing acoustic data for this node, and to a list of potential words that this node may terminate. Note here that the connection to the SMD data is through a pointer. Rather than store acoustic data with the nodes of the search tree, this pointer based method provides a clean disconnect between the static model data and the modified search tree data, and reduces the need for replication of the much larger acoustic model data structures. As the HMM nodes maintain the current state of the search process, they are actively modified throughout the steady state search.

Word Level Model

In order to take advantage of higher level language constructs, Sphinx is also able to incorporate a word and sentence level model into its HMM framework. This model provides probabilities of association between the words in the recognizer's current vocabulary.

This association probability can be for any number n of words, and is called an n-gram model. For example, a 1-gram model would simply be a list of the words in the vocabulary associated with their probability of occurrence. A 2-gram (bigram) model is represented by a list of all two word pairs that occur with non-zero probability, and their associated likelihood of occurrence. Due to the exponential growth of possible combinations as n increases, Sphinx only extends this approach up to trigram models, and only utilizes word combinations that occur above a certain probability. The probability information itself can be generated via other toolsets, with the most common being the Cambridge Statistical Modeling Toolkit [22]. This toolkit generates an n-gram model for the 20,000 most common word associations from a corpus file of reference sentences.

The actual data components of this word level model are the dictionary and the language model. The dictionary is simply a list of all words available to the system (the vocabulary) along with word specific information such as a unique ID, a textual representation, a list of phonemes that make up the word, and a list of potential successor words in the grammar. It's programmatic representation is essentially a large "struct" for each word, and it is immutable within the steady state of the program (though the size of this structure and the language model discussed next vary with the size of the vocabulary).

The language model serves to associate probability values with inter-word transitions for all up–to–n combinations of words. In practice, the frequency of occurrence of most combinations of words in the vocabulary is very low, and is thus ignored to save space. As follow–up words are maintained in the dictionary, a follow–up word without a language model probability is appropriately considered but given low weight. This produces, in effect, a sparse matrix of probability scores. In Sphinx2, this data is represented as a series of annotated single word arrays, where the annotations are pointers to lists of higher–order (–gram) probabilities. Thus, the unigram probability information (the probability of a given word occurring at all) is stored in a single array, accessible by the word's unique ID. Each word data element in this array includes a pointer to the first entry in a second array of bigram probability values associated with this word. Thus, given the word sequence "'The

Car", the system may individually look up the unigram probabilities of each of the two words, and may find the bigram probability of the combination by following the bigram array index pointer stored with "The", and scanning forward from that point in the bigram array until it finds the word "Car" or reaches the beginning of the next unigram's list. This same procedure may be repeated to reach the n–gram probability score for any n. While this approach may appear to trade off performance for space, the orders of magnitude reduction in the memory footprint of the language model makes up for much of this through better locality.

2.3.2 Search Phase Computation and Program Flow

At a high level, the search phase of sphinx takes as input a set of feature values extracted from the front end DSP style pre-processing phase. Each value represents a specific mathematical extraction from the acoustic signal (eg: power, cepstrum, and derivatives thereof) for each 10ms frame, and is represented as a floating point number. The true output of the search phase is a probabilistic word lattice containing all of the potential words recognized during the search phase and annotated with various probability values derived from both input data correlations and language model interactions. A final post-search step scans over the constructed word lattice to extract the highest probability path through the matrix, producing the recognition result.

As the previous section on data organization alluded, the search phase can be broken into a number of steps which occur once per input frame in order. These steps are depicted in higher detail in figure 2.9. To provide some insight into the overall operation of Sphinx, and establish the fundamentals necessary to understand the modifications we make in this research, the remainder of this chapter will walk through one iteration of the search phase in detail. We will begin with the post front–end feature vector set, and consider the programmatic flow through model evaluation and work recognition.

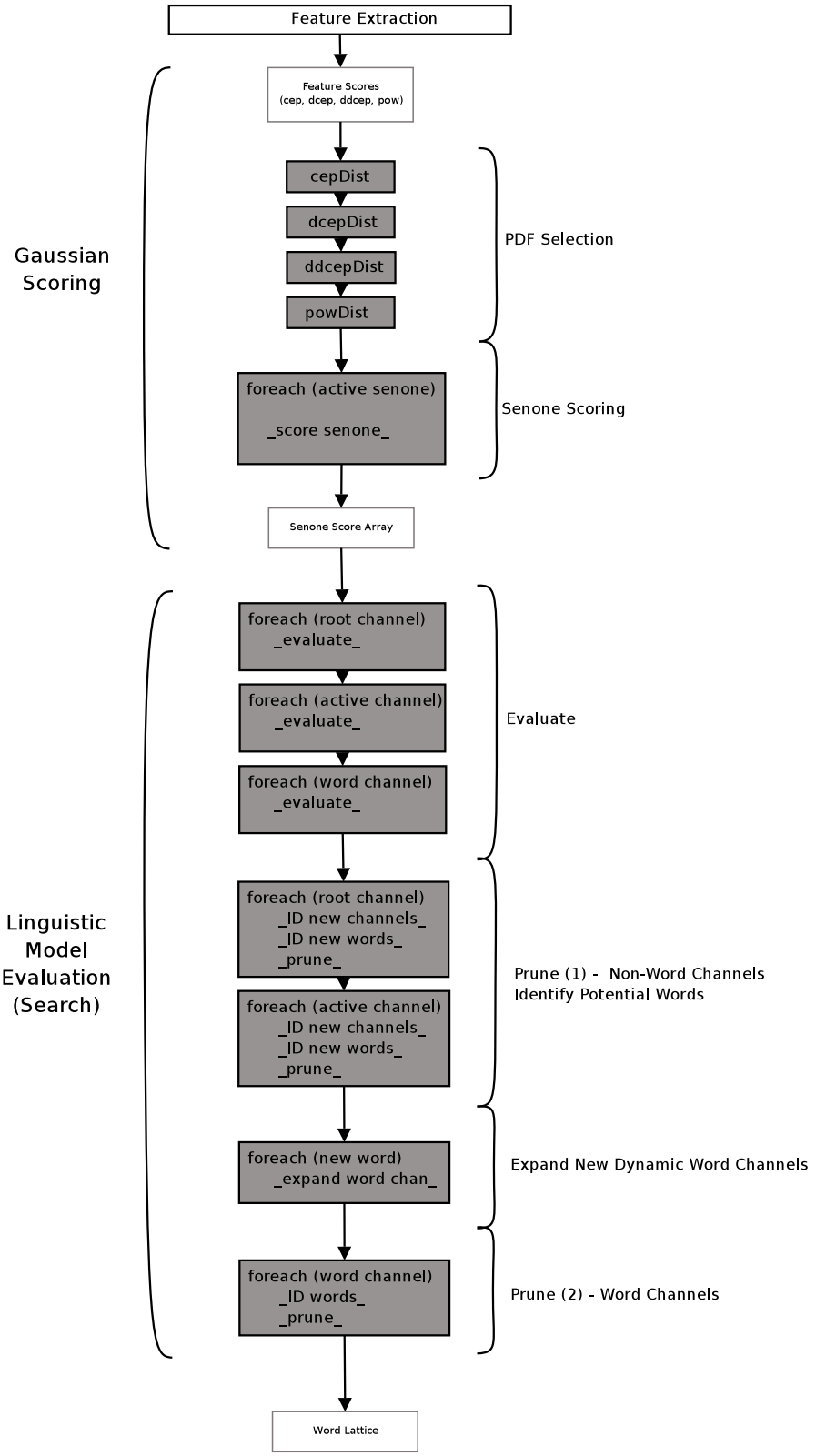


Figure 2.9: Sphinx Search Phase Programmatic Flow - This figure depicts the overall program flow for a single iteration (upon a single input frame) of the Sphinx search phase. Note that the feedback path of active senones from model evaluation back to the next iteration of Gaussian scoring is not shown.

Gaussian Scoring

The first step in a search iteration (PDF identification) it to process the value of each feature in the current frame, using mean, variance, and distance computations (relative to previous data points) to identify the appropriate codebook entries to apply to the current input frame. This phase requires significant floating point computation, but is relatively short in comparison to the remainder of search. Upon completion of this phase, the algorithm will have produced an appropriate integer representation of the floating point feature value (while the specifics of this conversion are beyond the scope of this discussion, it suffices to note that a logarithmic representation that may be manipulated through integer calculations is used in much of Sphinx to reduce the need for floating point computations), and will have identified which 256 entry codebook array should be used in the scoring of each feature (also through a formulation beyond the scope of this discussion). Our analysis indicates that this specific set of computations constitute 4–5% of search phase execution time, though this percentage will vary with search parameters (e.g. larger knowledge bases or wider search beams will increase the total time spent is search, decreasing the proportional time in this code).

The second per–frame processing step is (the actual Gaussian Scoring) involves utilizing the generated features score and selected codebooks to produce a distribution score set for each active senone in the vocabulary. This corresponds to the Sphinx implementation of the Probabilistic Scoring step discussed in earlier chapters. In this case, Sphinx assumes that the probability distribution associated with each acoustic model element can be represented by a standard Gaussian curve, and the form of that curve is represented in its Gaussian codebooks through vector quantization. The primary computation performed in this phase is a series of normal and logarithmic additions (utilizing a precomputed logarithmic addition table).

Active senones are identified as those senones pointed to by a currently active HMM node which, in turn, is active if a current path through it is of sufficiently high probability. Implicit in this description is the existence of a feedback path from the model evaluation

phase of computation. Though the use of logarithmic representations, Gaussian codebooks, and integer arithmetic significantly reduces the overall computation required in the Gaussian scoring phase, it still constitutes a substantial runtime effort. The use of this feedback path allows the algorithm to ignore all senones that, based on the current state of the search, will not be considered in this frame of input. As we will see later, however, this feedback path enforces a degree of serialization into the search phase that may not always be necessary.

Our evaluation shows that this senone scoring phase represents 20–30% of the search phase execution time. When the feedback path is removed, and all senones must be evaluated on every input frame, this number jumps to 60–75%, demonstrating the benefits of minimizing unnecessary senone scoring. Once again, the specific contribution of this phase to overall search time varies with respect to knowledge base size and search complexity.

Model Evaluation

The array of senone scores generated in the Gaussian scoring phase is passed on to the actual search phase. Sphinx provides the capability to apply three different search algorithms to the input data. The first is a standard flat Viterbi beam search, as described in Section 2.1.2. On each iteration of this search, the current set of active recognition hypothesis is extended by scoring knowledge base reference data against the quantized input feature vectors. Once the scoring has been completed for that iteration, candidate paths with cumulative probabilities below a fixed threshold are eliminated, and the search moves on to the next input vector.

The second search capability available in Sphinx is a lexical tree based beam search. This search is a somewhat more domain specific form of HMM traversal as compared to the flat Viterbi search. It is developed from the observation that the knowledge base represents phonetic data, and is connected to form word associations. In order to reduce the overall search effort, it is possible to re-structure the transitions between nodes of the HMM in a hierarchical fashion using lexical information as shown in Figure 2.10, taken from cited works [20]. Thus, all words starting with the same phonetic unit (/s/ in this example)

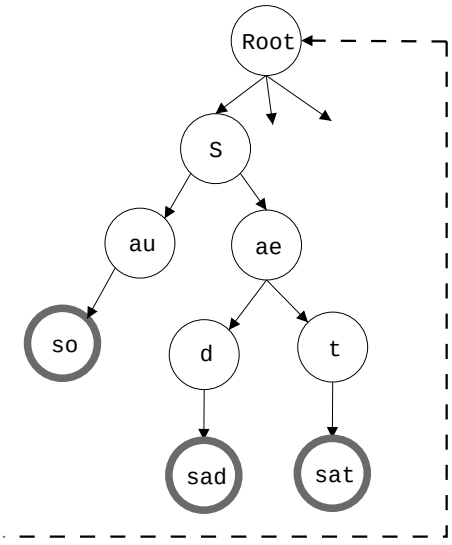


Figure 2.10: Lexical Tree Based HMM Layout

are described as part of a tree originating at that phonetic unit. While this does serve to significantly reduce the number of active search paths, it does introduce some new problem. For example, the language model information must be applied much more cautiously if a given path can be associated with a large number of words rather than just a few. Solutions to these types of problems are outside the scope of this work, but a number of possible solutions are presented it cited works [70, 12, 69, 68, 77]. The relevant aspect of these search techniques in this scope is that they do not fundamentally alter the architecture level behavior of the program.

The final search phase of Sphinx is essentially a high level language model search. Recall the discussion of language models in the previous section, and use of trigram language models to provide a more accurate picture of common word interactions. While this technique is very useful in aiding accuracy, it turns out to be very difficult to incorporate anything higher than a bigram language model directly into the HMM knowledge base structure. This difficulty is not one of theoretical capability, but rather of practical capability: in order to disambiguate such data, it would be necessary to massively replicate relevant sections of the knowledge base to the point of absurdity. The solution used by Sphinx is to apply trigram language model data in what amounts to a post-processing step. The two

beam search phases are used to build up a 'word lattice' of potential word transitions, as discussed previously. The post processing search views this information as a whole, and applies the trigram language model information to possible word transitions, adjusting recognition hypothesis probabilities appropriately.

This work utilizes the tree based search method available in Sphinx. While its characteristics are very similar to the standard viterbi search, allowing our results to be extrapolated to more general searches, its minor use of domain specific information makes in generally more accurate and less complicated than a straight viterbi search over the entire knowledge base would be. Given our target application and domain, this seems to be the reasonable option. In practice, Sphinx appears to maintain distinct lexical trees for most words in the knowledge base (as opposed to the highly condensed example in figure 2.10). The final language model level search is essentially a refinement of an existing hypothesis, not an independent search capability, and thus is not considered in this work.

The tree based lexical search can be divided into two internal steps: model evaluation, and pruning. During the model evaluation phase, the algorithm steps through a list of all currently active phones (referred to in Sphinx as "channels", these are simply one complete 7–stage phone HMM model). The evaluation of each phone model involves a comparison of each transition within the model against the appropriate scoring information generated previously. Highest probability paths are then propagated one step forward (leaving the entry node "empty" and destroying the data at the exit node). This procedure is replicated for all root channels (those nodes that begin a word, found at the head of a word tree), all intermediate channels, and all word channels (those representing the last phone of a potential word, discussed in more detail shortly). As one might expect, the primary operations performed during this phase are related to selection and sorting. As such, comparisons and conditional moves predominate. This model evaluation phase constitutes around 30–40% of execution time in our evaluations.

The final major operation involved in an iteration of the search phase, and the second component of the actual model evaluation, is node pruning based on the newly annotated

probability data, and activation of successor states. This begins by scanning over all active root and intermediate channels, checking the current score values. If *no* score within a channel is of high enough threshold to warrant continued evaluation, the entire channel is marked inactive and all paths currently terminating in it are discarded. If the current exit score meets the threshold requirement, its value is propagated to the (now empty) entry state of all successor channels. Otherwise, the current exit path is ignored, and thus pruned when overwritten during the next evaluation phase. After successor phone activation, the current exit score is evaluated against a "word recognition" threshold. If this threshold is met, it indicates a potential word recognition.

At this point, it is necessary to consider word recognition in slightly more detail. As discussed earlier, the purpose of senone based training is to help capture co–articulations that occur due to vocal inflections in human speech. This same effect occurs at inter-word boundaries. Thus, simply stringing together words and applying language model level scoring will not allow accurate modeling of the expected input sounds. Unfortunately, the solution is to produce extra channels to represent transitions between the last phone of the first word and the first phone of the next word for all possible word combinations within the knowledge base. Even if this expansion is limited to those combinations with definite probabilities in the language model, the result is still a significant (orders of magnitude) increase in the size of the knowledge base. Such extra channels would, however, be very rarely used, as they only become relevant if the word threshold is met *and* only for the inter–word transition that activates the successor word with the highest probability. In order to alleviate this problem, Sphinx utilizes a form of dynamic modeling. The static search tree is constructed without channels representing the last phone of a word. When a potential word is recognized, and if it provides the highest probability entry path into its successor for the current input frame, the necessary inter–word channels are dynamically generated. Thus, when a "word identification" threshold is met during pruning of a root or intermediate channel, its list of successor words is added to a list of such successors for the current input frame. Once all static channels have been evaluated, this list is used to

determine which inter–word channels must be generated.

Once all inter–word channels for the current frame have been generated, the entire set of currently available inter–word channels are pruned. If the exit state of one of these channels meets the threshold requirements, it is used to activate the root channel representing the successor word. In order to maintain dynamic context modeling, certain acoustic data pointer information is also propagated to the activated root channel. If no score within a inter–word meets the threshold, it is deleted (added to an internal free–list to minimize memory allocation and deallocation).

A successful word transition also leads to the addition of the word to a word lattice. This lattice contains all recognized words, and backward pointers to help identify the appropriate sequence of words upon conclusion of the search. Unlike the other phases of the search process. This global manipulation constitutes a very small portion of the overall execution time.

The series of steps discussed in this section constitute the programmatic details of the sphinx search phase. Each of these steps is repeated for every input frame of data. As we consider techniques to expose concurrency in this application domain, we will refer back to these phases of program execution.

2.3.3 A Word About Sphinx-3

Since the beginning of this research, the CMU Speech Group has moved to the next generation of their speech recognition library, Sphinx–3. The most significant difference between Sphinx–2 and Sphinx–3 is in the Gaussian scoring phase. As discussed previously, Sphinx–2 uses semi-continuous Gaussian model codebooks, and a logarithmic data representation format to minimize the use of floating point computation at program runtime. In contrast, Sphinx–3 computes senone scores by applying a continuous Gaussian distribution. Where the feature input data in Sphinx–2 is used to generate an initial score and select a codebook for later use in senone score computation, in Sphinx–3 it is used to generate parameters for a continuous Gaussian distribution and senone scores are computed directly.

The move to continuous Gaussian distributions is intended to increase accuracy, and has the effect of expanding the need for floating point computation through senone scoring. It does not, however, affect the representation of the actual senone score, nor does it affect the degree of concurrency available in the following search phase.

CHAPTER 3

Performance Analysis of Speech Recognition

In order to better motivate the optimizations explored throughout the remainder of this work, we analyze Sphinx performance characteristics across a number of parameters and workloads. The analysis presented here is generated by use of the SimpleScalar toolset, focusing on memory system performance, and utilizing the SimpleScalar front-end to the cheetah cache simulation library to explore a wide variety of configurations [6]. All studies presented are based on the forward tree search algorithm. Analysis of the other search options available in Sphinx-II show very similar memory system behavior (in terms of cache miss rates), with the bestpath search showing similar patterns with a lower ratio of cache misses. As the bestpath search traverses only a subset of the knowledge base (as identified by prior search phases), this result is entirely expected. The speech recognition engine was compiled for the ARM ISA, allowing for raw performance computations that are tailored to represent a standard embedded system configuration. We employ a baseline configuration consisting of 32-byte cache blocks for all cache configurations, a 11447 word dictionary file, (representing about 35 MB of runtime memory) and default search parameters to Sphinx-II in this study. Evaluations are performed by manipulating individual parameters from this base model. The data presented in this chapter appears in the Proceedings of the Workshop on Memory Performance Issues (2002) [49].

Figure 3.1 shows instruction per miss and cache miss ratios for a number of L1 cache configurations on the baseline system. We observe in this result the relatively high miss

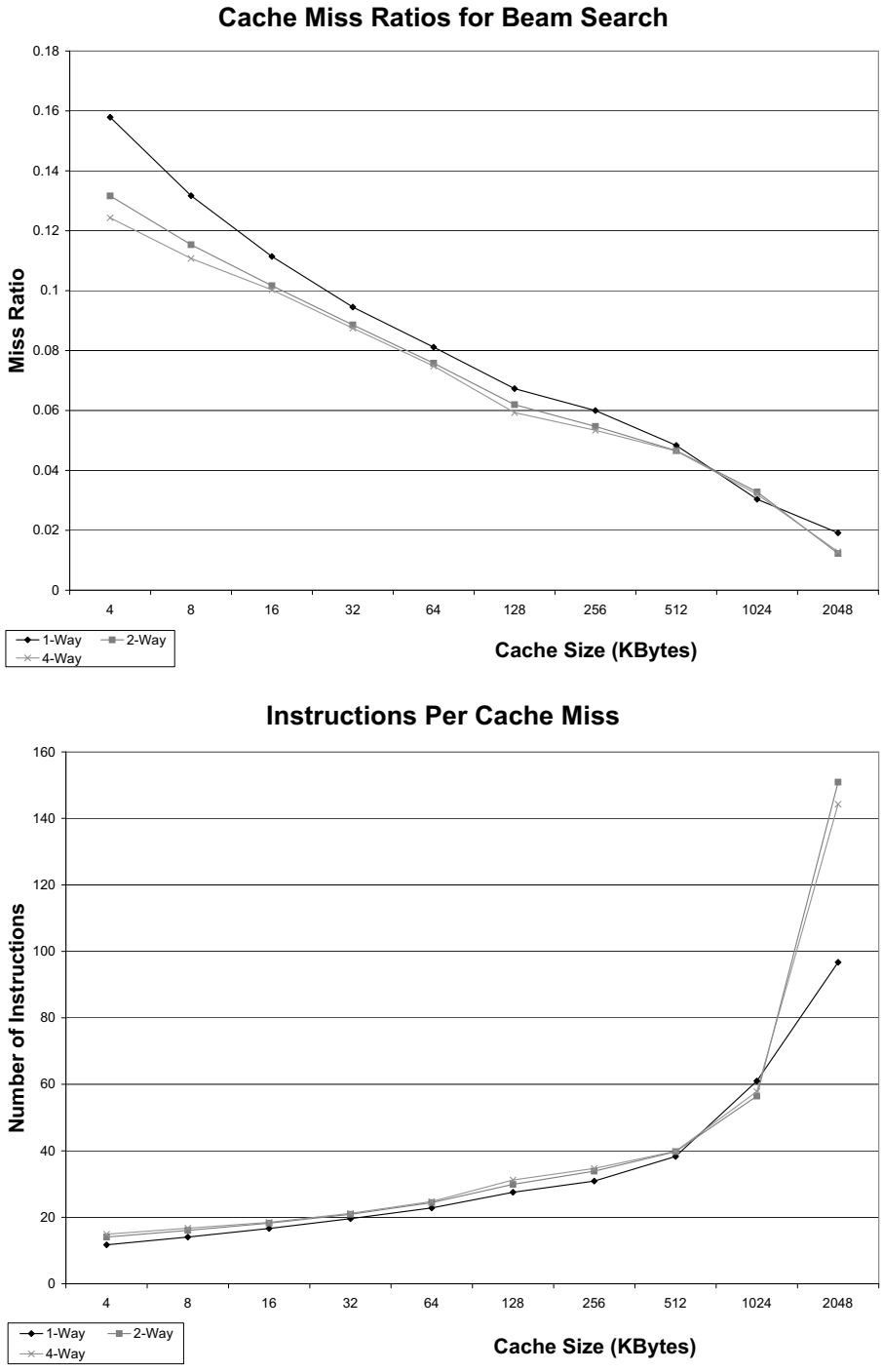


Figure 3.1: Memory Characteristics of Baseline Configuration - Average number of instructions per miss and cache miss rates for beam search on baseline configuration.

rate characteristic of speech recognition software, and described in previous works. As point of comparison, the gcc benchmark of spec2000 demonstrates a miss rate of approximately 780 instructions per miss for a cache configuration similar to the 32KB data point [17]. We also recognize, however, that this particular selection of metrics does not really consider the fact that the speech recognition workloads naturally specify a more intuitive unit of 'work' than that of most software applications. For example, in order to be useful, the system must ideally achieve a rate of recognition that matches the rate of input sounds such as syllables or words, typically 150 words per minute for non-excited speech. We discovered that without considering this inherent unit of work, the results of experimental trials were often misleading. Consider the effect of increasing the accuracy of the search process. This has a direct result of increasing the number of instructions executed as well as the number of memory references and corresponding cache misses. As it turns out, the rate of increase of instructions and memory references is often comparable to or slightly higher than the rate of cache misses, leading to a result of unchanging or slightly decreased cache miss ratio for increased search accuracy. This would lead one to believe that higher accuracy searches do not have a substantial impact on overall performance, clearly an inappropriate conclusion. Thus, in order to present a more accurate picture of program performance, we present all of our results in this chapter as metrics relative to the average duration of time of a spoken syllable (e.g. 'misses per syllable'). The computation used to arrive at this metric simply involved totaling the number of actual syllables in the test input speech sequences and normalized all associated data against this value. This formulation was chosen to provide both intuitive human understanding (syllables being much more intuitive than the 'frame' time-slices used internally by the software) and relatively similar unit sizes (variation in length of syllables is clearly much smaller than variations across entire words). We begin our evaluation by investigating the effects of a more comprehensive knowledge base.

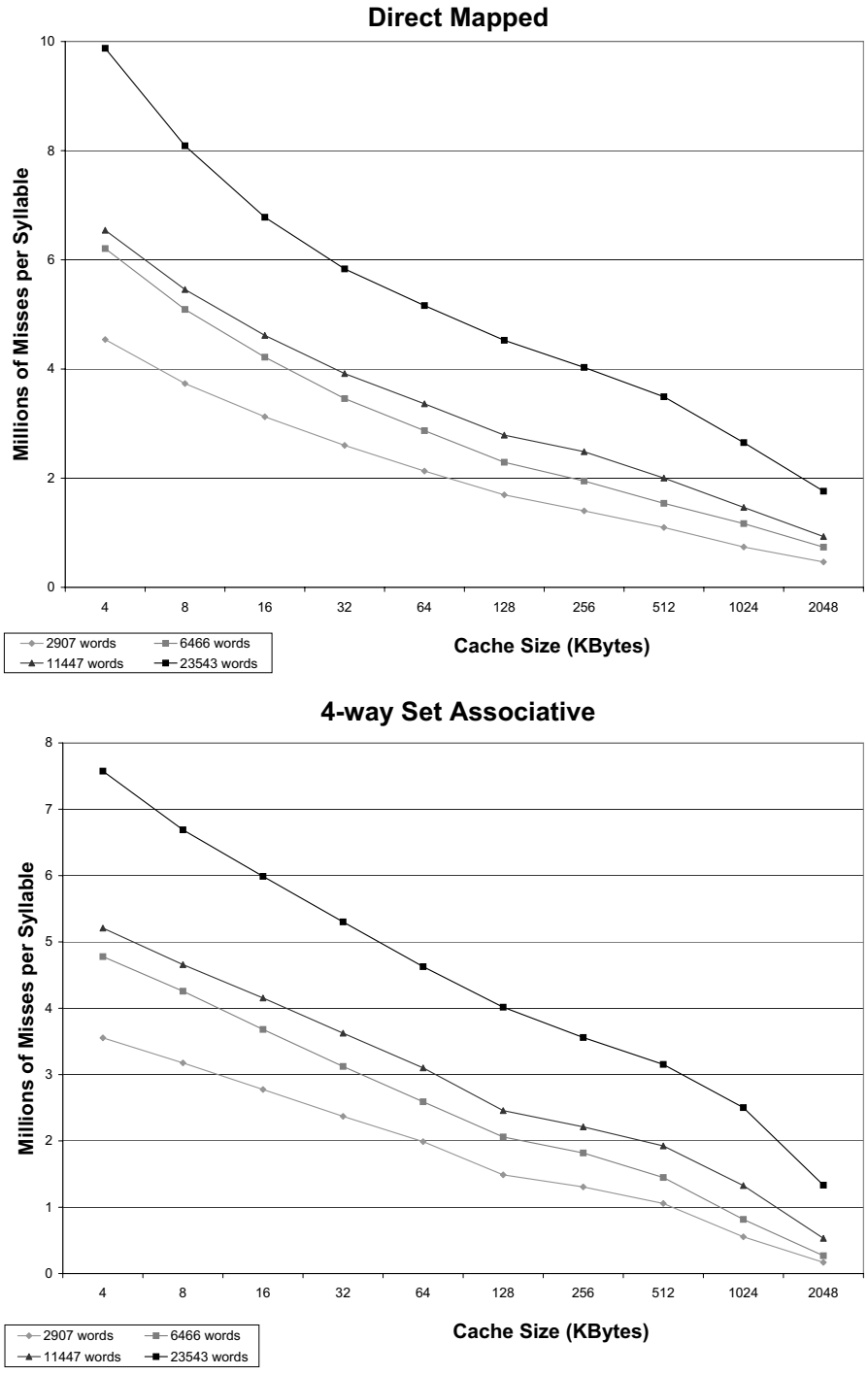


Figure 3.2: Memory Impact of Knowledge Base - Average number of cache misses per syllable for varied dictionary sizes

3.0.4 Knowledge Base Size

We consider cache miss rates for an array of knowledge base sizes. We expect larger knowledge bases (larger dictionaries coupled with more comprehensive language models, generated from larger sample data) would result in larger memory structures, leading directly to a higher cache miss ratio. The results are presented in Figure 3.2 and demonstrate an interesting pattern. While we do see an increase in the number of cache misses per syllabic time unit, the increase is not at all comparable to the relative increase in dictionary word count. This feature is most directly attributable to compression techniques applied to the knowledge base by the software. While the 2907 word dictionary (and corresponding language model) represented approximately 30MB of runtime memory space, the 23543 word dictionary (and corresponding language model) represented an increase of only 12 MB (the 6366 word dictionary and 11447 word dictionary resulted in increases of one and five megabytes respectively). Thus, while there is a high initial cost for establishing knowledge base data in memory, data compression clearly serves to mitigate the effects of adding further data. A second possible explanation recognizes the fact that our knowledge base creation software bases dictionary and language model constructs on a large input text file [22]. Thus, a larger dictionary also corresponds to a language model which is built off of a much more comprehensive sample of the language, potentially allowing the search algorithm to more quickly identify high probability paths and prune out lower probability candidates. This corresponds to observations made in previous works [95]. It should be noted that the actual cache miss ratios did not vary substantially across dictionary sizes.

3.0.5 Cache Block Size

In order to provide some intuition into spatial versus temporal relationships in memory accesses, we investigate the effects of adjusting cache block size. The results are presented in Figure 3.3, and clearly demonstrate the presence of spatial locality, as seen from the decrease in cache misses for larger block sizes with the overall cache size held fixed. The benefit of spatial locality is also seen to diminish with increasing size. This trend corresponds with

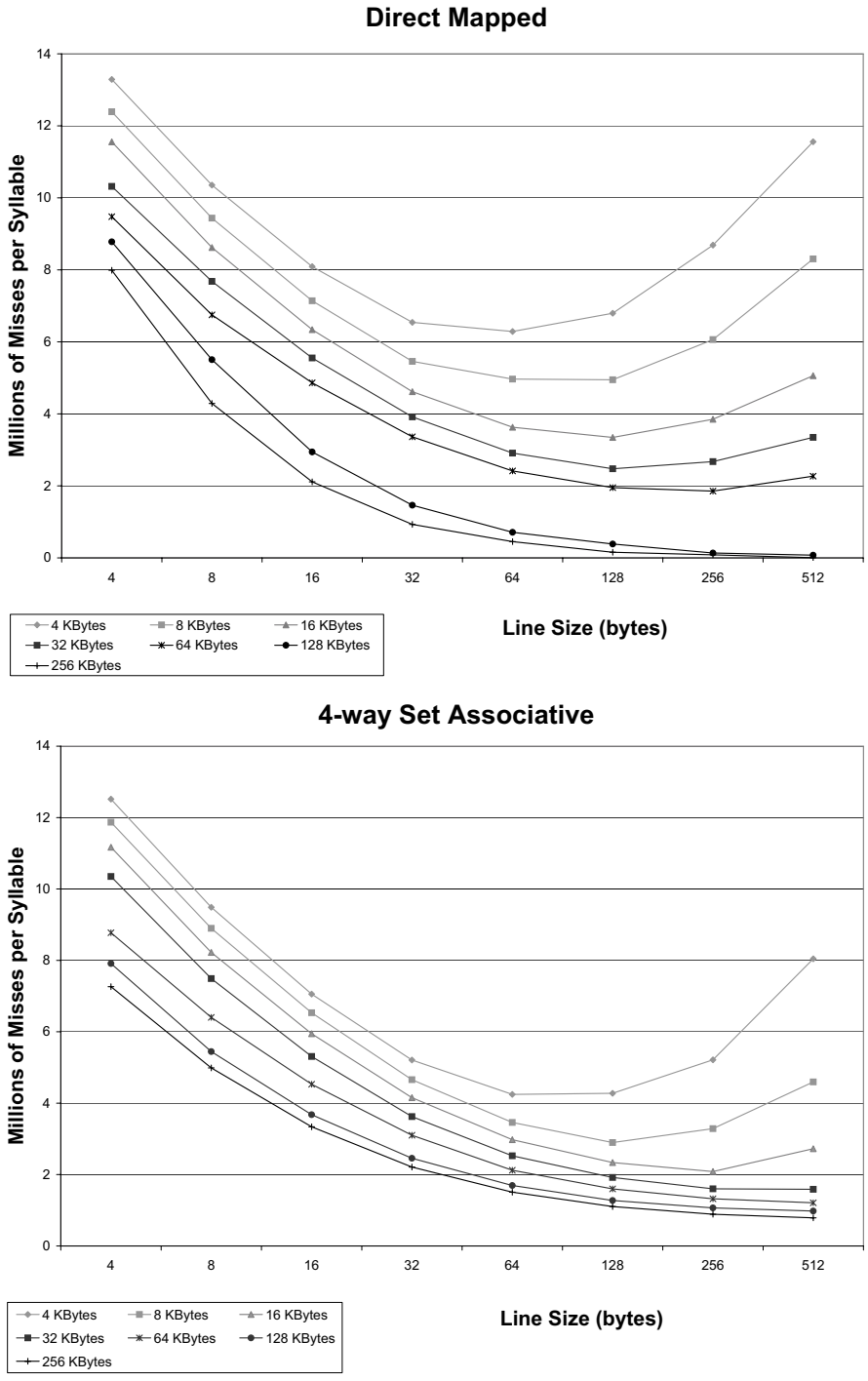


Figure 3.3: Memory Impact of Cache Line Size - Average number of cache misses per syllable for various cache sizes for a range of cache line sizes.

the notion of traversal of a linked data structure, evaluating various data points at each node before moving to the next. Thus, the pattern seen in the figure can be related (at least in part) to the size of a node data structure, which ranges from approximately 76 to 100 bytes (depending on specific node type). Clearly, the ideal approach by which to exploit this observation would involve coordination of cache block size with knowledge base node size, and alignment of node data with block boundaries.

3.0.6 Search Beam Width

The width of the search beam (effectively the pruning threshold for search channels), serves as a primary means for establishing a trade-off between accuracy and speed. Wide beams allow the search algorithm to explore a greater number of low probability paths in the search graph, increasing the probability of an accurate match, but simultaneously increasing the computational effort required to complete the search. Small beams prune out a number of high probability paths, sacrificing potential correct matches for a smaller search space. To evaluate the effects of beam width on the forward search, we perform simulations with three width configurations. The 'live' configuration corresponds to pruning thresholds suggested for live mode operation; the 'default' configuration represents default sphinx parameters used in the other evaluations in this paper (approximately one half the pruning threshold of 'live' mode), and the 'extended' configuration represents an even wider beam configuration (slightly less that half the pruning threshold of 'default'). Results of these evaluations are shown in Figure 3.4 and show that the increase in number of misses per syllabic unit actually correlate fairly well to decreased pruning threshold. This corresponds to the expected exploration of a greater number of possible paths, with relatively little locality seen in the added work. As such, this also corresponds to an increase in the working set size of the program.

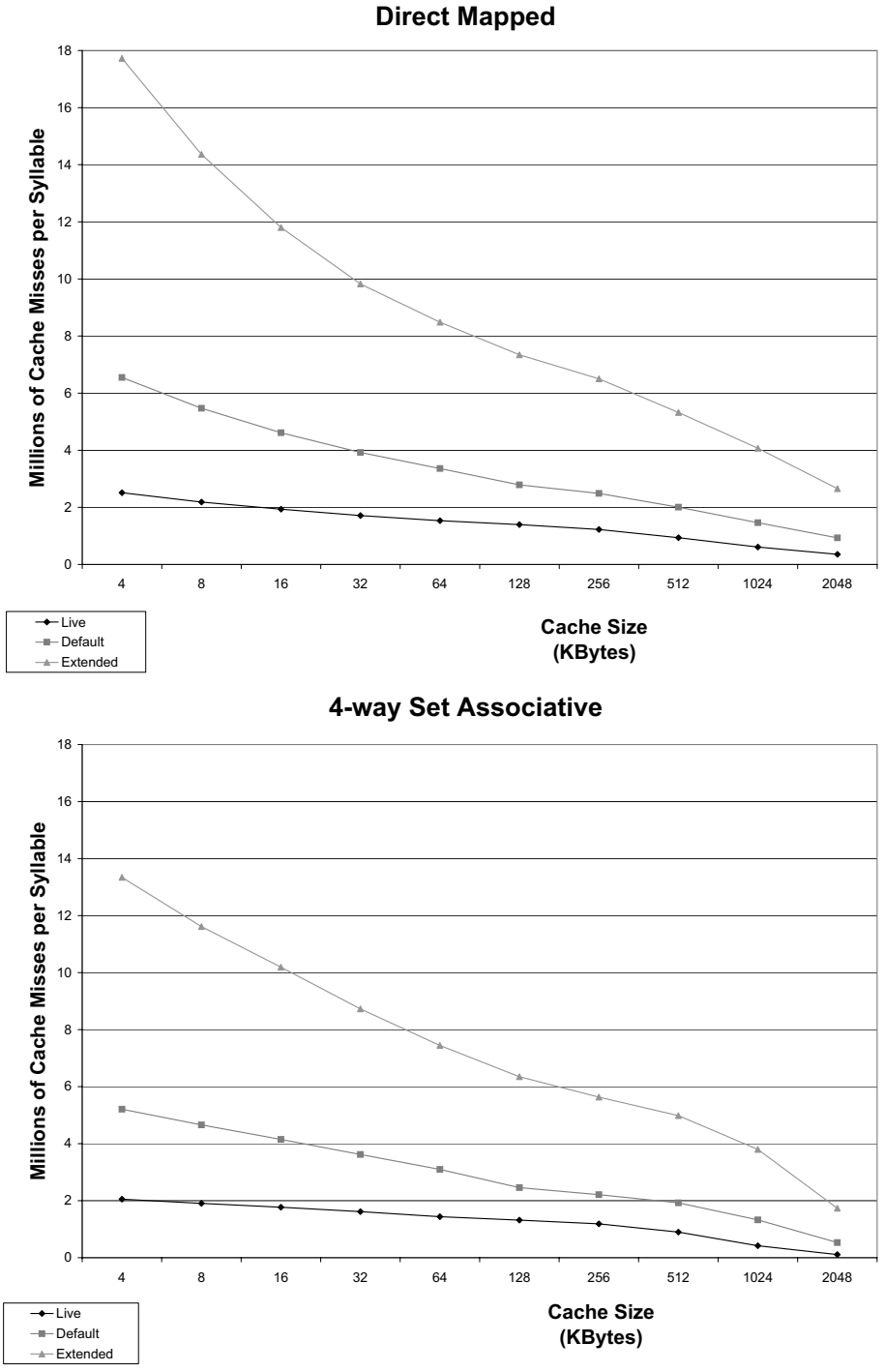


Figure 3.4: Memory Impact of Search Threshold - Average number of cache misses per syllable for various forward search beam thresholds.

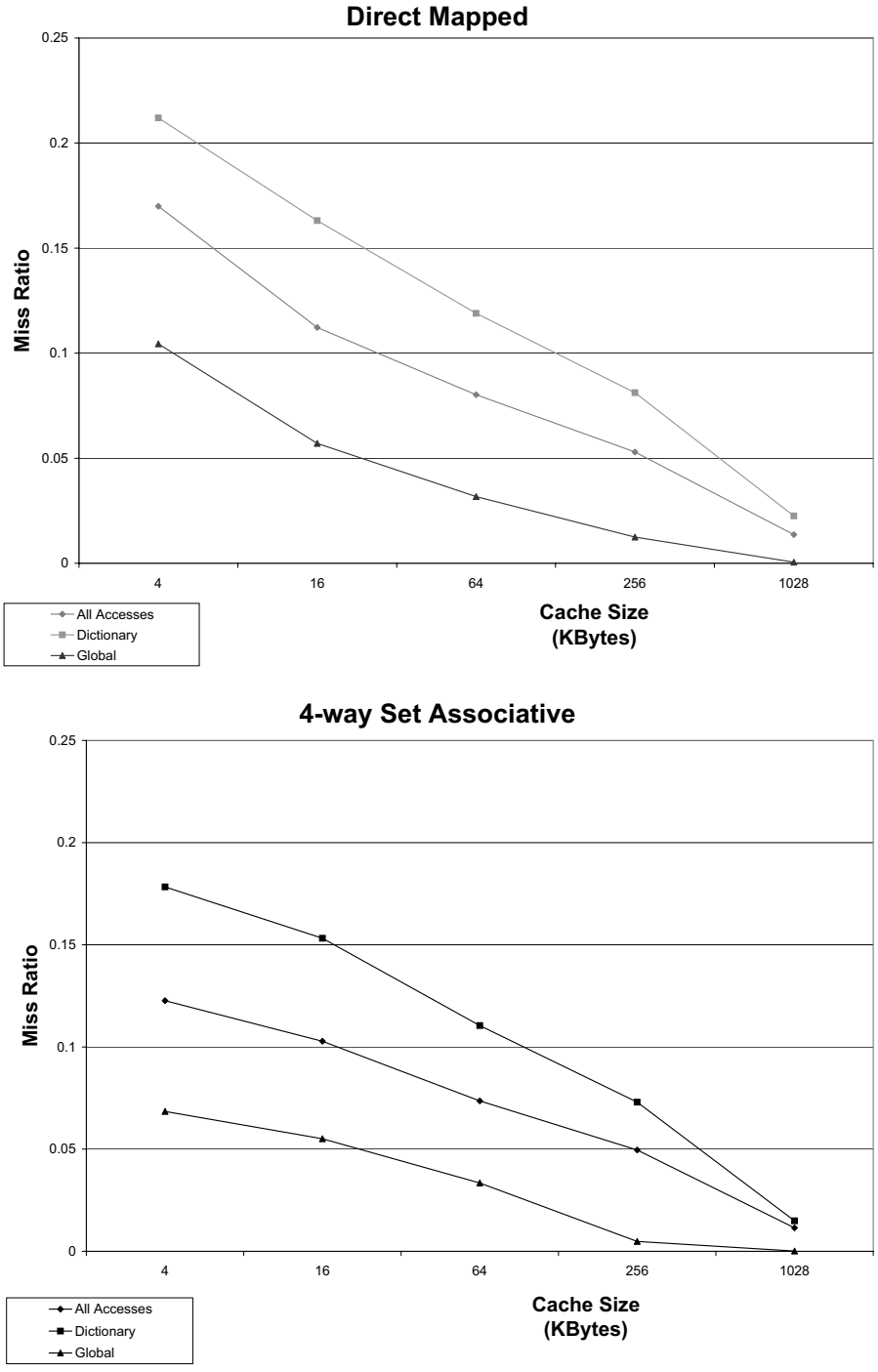


Figure 3.5: Memory Impact of Partitioned Reference Streams - Straight cache miss ratio for forward search, partitioned by reference type.

3.0.7 Data Partitioning

An interesting feature of this application domain is the presence of a large, easily identifiable memory structure to which a substantial portion of memory accesses are directed, the recognition knowledge base. As previous work has suggested that the poor memory performance observed in this domain is due largely to accesses to this knowledge base, we wish to specifically investigate the behavior of memory accesses to this region as opposed to other regions of memory. The results presented in Figure 3.5 show a clear partition in cache miss ratios to various regions of the memory space, and demonstrate that, while constituting approximately one quarter of the total number of memory references, traversals of the recognition knowledge base contribute significantly to the overall number of cache misses. Partitioning the memory space to isolate accesses to the knowledge base from other data accesses may provide the opportunity for significant improvement in memory system performance. A review of the other parameters evaluated in this paper shows this memory space division is consistently observed in all cases except the runs evaluating the best-path search algorithm. While the distinctive variations in miss ratios are still observable in this search phase, the difference between knowledge base and other accesses is far less dramatic and knowledge base accesses cannot be considered the primary source of cache misses.

3.0.8 Other Parameters

While the results discussed previously cover notable observations in our result data, a number of other parameters were investigated. Variations in these parameters, however, showed no affect on cache performance. Among these, speaker selection and input selection. Speaker selection was evaluated by presenting the recognition engine with identical phrases spoken by a number of different individuals (three male, one female). Results of these evaluations showed no major effect of speaker on cache miss ratios. A similar absence of effect is observed for various input phrases. Interestingly, an input sample constructed of two words repeated (the file was concatenated with itself multiple times to ensure identical repeating units) showed cache miss ratios similar to phrases constituting extended

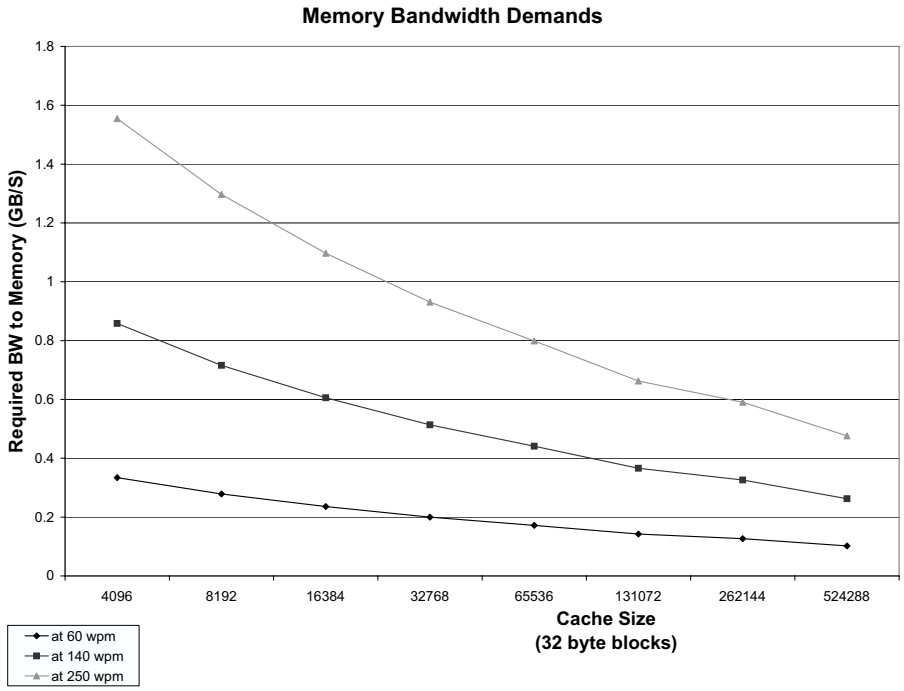


Figure 3.6: Memory Bandwidth Demands - Overall memory bandwidth demand of forward search across a variety of L1 cache configurations.

sentences with detailed structure. Me must note that neither of these evaluations could be constructed in a particularly quantitative manner. For example, it is unclear how one would quantify the intonations generated by a particular speaker in a metric that is relevant to speech recognition. Similarly, it is equally difficult to quantitatively represent the differences between speech inputs. Thus, it is possible that our variations in speaker and input were insignificant given the processing methods used by the speech recognition software. In future work, we hope to use techniques such as entropy analysis to better quantify the 'complexity' of such input variations.

3.0.9 Bandwidth Considerations

While the locality and cache characteristics of speech recognition search algorithms present some significant insight into the problems faced by these applications, they are uninteresting without some notion of overall memory bandwidth requirements. Thus, we consider memory bandwidth requirements between L1 cache, and main memory for the

cache configurations previously considered. In order to provide a better picture, we present a set of scaled bandwidth demands required to achieve various speech recognition rates. Results of this analysis are shown in Figure 3.6, and demonstrate that a fairly high bandwidth memory system is required by speech recognition software. While these demands are not at all unfeasible for current generation memory systems (current generation RDRAMs are capable of 1.66 GB/s), such form factors are not likely to be seen in handheld and low power devices. These results indicate, however, that the problem lies not in the available memory bandwidth, but in how the bandwidth is used and latency tolerated.

3.0.10 Power Considerations

As a final consideration, we wish to evaluate the amount of power required to access an average memory system at the rates required to a maintain continuous speech recognition, particularly with regards to lifetime on an average battery. This analysis was performed using the system power estimator available from Micron Technologies for their baseline SDRAM product [4], and assuming a 3.0v power supply and a 64bit 133MHz bus. The memory system was assumed to consist of three 16MB SDRAM modules, with references divided equally among them. The results of this power consumption estimate demonstrate that, given a 32KB cache, the resulting power consumption due to accesses to main memory required to maintain a 250 wpm speech rate would drain two Duracell "AA" batteries in forty-five minutes to one hour. While this time-frame appears rather short, it is important to consider that this is one hour of continuous recognition of 250 wpm speech, which is truly a worst case scenario. Furthermore, there are a number of potential techniques (such as the use of ROM memory for immutable knowledge base data) which could substantially reduce this overall power consumption estimate. Thus, what this analysis really demonstrates is that even using commodity parts in a worst case environment, the power consumption of the memory system does not present a substantial problem in efforts to produce low power speech recognition hardware.

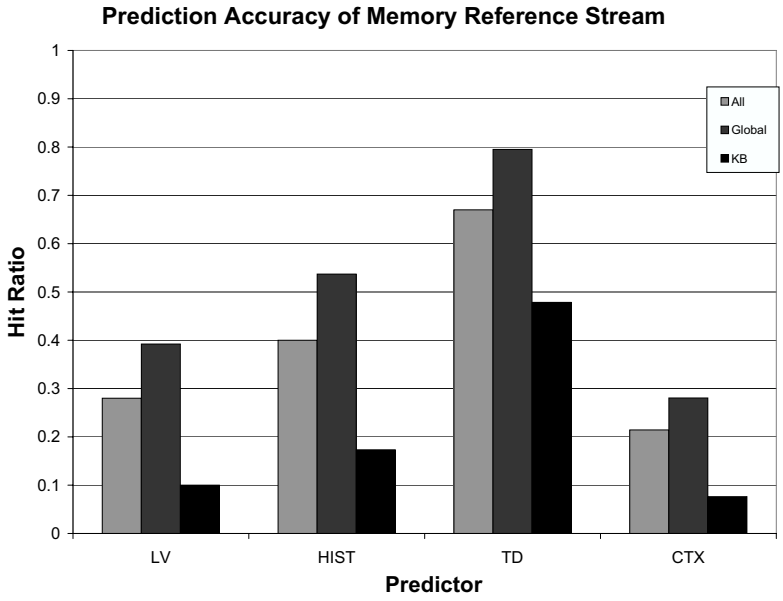


Figure 3.7: Reference Stream Predictability - Overall predictability of memory reference stream generated by forward search.

3.0.11 Reference Predictability

In order to consider the potential effectiveness of hardware pre-fetching in the memory system, we consider the predictability of memory references against a number of standard predictor types. We evaluate four types of predictors, each with generous resources (for example, 64k-entry tables) in order to observe best case value. The results of this analysis on the raw memory reference stream is presented in Figure 3.7. The "LV" predictor represents a standard last value predictor [61]. The "MARKOV" configuration represents an LRU list based history predictor[41]. The "TD" configuration represents a 2-delta stride predictor [16]. Finally, the "CTX" configuration represents a context based history predictor [85]. This analysis shows that, while there is clearly some predictability in the stream, the majority of this predictability comes from global and stack structures, while dictionary accesses show poor predictability in the general case. Only the 2-delta stride predictor demonstrates even marginally reasonable accuracy. A further analysis of the predictability of the cache miss reference stream shows that even the previous result is a rather hollow victory. The prediction accuracy for references out to main memory (cache misses) for

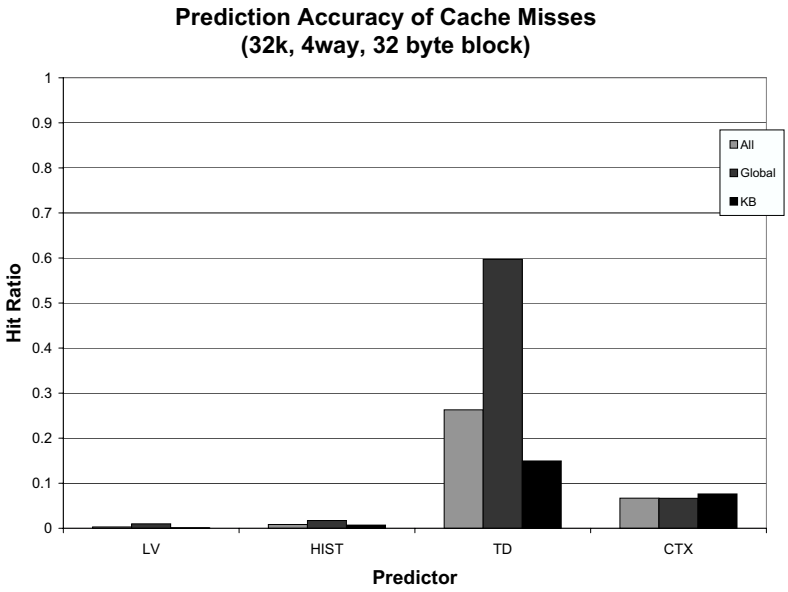


Figure 3.8: Cache Miss Stream Predictability - Predictability of reference stream after filtering by a small L1 cache.

a set cache configuration (32k, 4way, 32 byte lines) is shown in Figure 3.8. Once again, the 2-delta predictor performs the best, but even in this case prediction accuracy for the dictionary accesses are extremely poor. It must be emphasized that this truly is a best case scenario for prediction. Not only do we assume a very large number of predictor table entries, but we take no account of timing effects (effectively assuming pre-fetching can be performed instantaneously). In an actual system, a accurate prediction may be needed tens to hundreds of cycles before the data itself is required.

3.0.12 Algorithm Threading

Another key feature of this application space is the inherent parallelism in the search process. The standard search techniques maintain a number of candidate paths through the internal speech model, representing the most likely recognition solution based on the input processed thus far. Each of these candidate paths (or channels) is conceptually independent of the others. Herein lies the potential for extensive parallelization. A brief study of serial executions of Sphinx-II showed that, for a number of the 'hot spots' in the program flow,

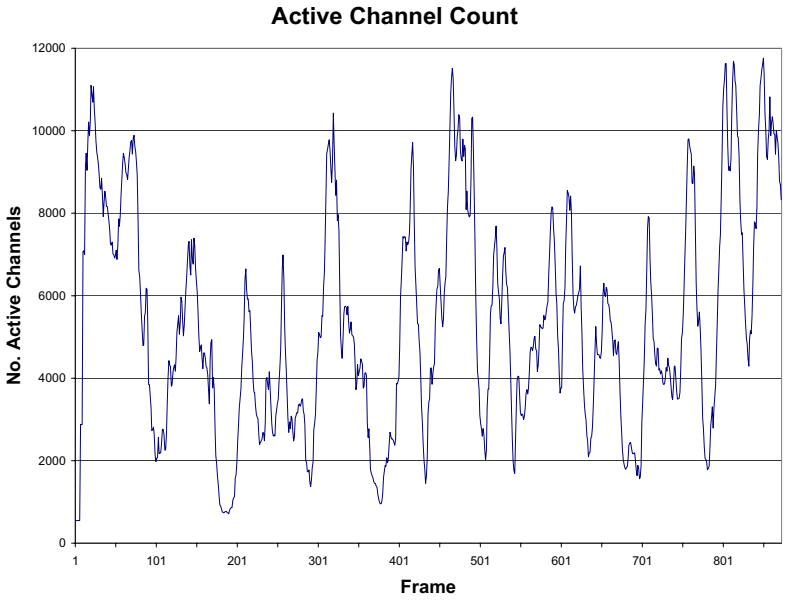


Figure 3.9: Active Channel Count - This graph depicts the number of active paths being explored by the forward search phase through a sample execution, demonstrating the availability of potential parallelism.

the average potential for parallelization ranged from 15-20 potential concurrent processes to over 2000. The available parallelism during beam search for a sample run (similar to a graph is cited works [9]) is shown in Figure 3.9 and clearly demonstrates the opportunities for parallel execution in this domain. In fact, algorithms to better exploit parallelism in certain search phases have already been studied [18, 76]. We observe, however, that exploiting such parallelism will naturally place higher demands on the underlying memory system, and concurrent execution is useless if memory bandwidth is unavailable. In order to investigate the impact of parallelization, we perform cursory modifications to Sphinx-II to operate with multiple concurrent threads during the forward search phase. These modifications involved no major algorithmic modifications, and demonstrated a 2-3x improvement is overall performance. This correlates directly to a comparable increase in memory bandwidth demand (a comparable number of memory accesses in half the cycles). Interestingly, however, the 'per-cycle' increase in memory requests is not substantial. Rather, the concurrent processes end up offset sufficiently that the vast majority of execution cycles only see one explicit memory request (that is, memory requests due to loads and stores, as opposed to instruc-

tion loads and such). As such, simultaneous concurrent parallelization appears to stress the number of outstanding requests allowed in the memory system, rather than the actual per-cycle bandwidth.

CHAPTER 4

Parallelization of Speech Recognition

The characterization of speech recognition considered in the previous chapter demonstrates the need to improve performance, and the potential constraints incurred by the memory demands of the application. It further demonstrates the potential for large amounts of thread level concurrency, which might be exploited to improve performance and tolerate some of the latencies incurred during execution.

This chapter will consider the potential for concurrency described in the characterization of Sphinx at an algorithmic and programmatic level. We will begin by discussing the general principles by which we approach our parallelization effort. We will then discuss how various sections of the search phase can be parallelized, and what resource would be useful in such efforts. We briefly consider the performance of a first round parallelization effort, and discuss modifications to the original model deemed necessary to improve performance. We conclude with a review of intuition to apply to the architectural model discussed in later chapters. It should be noted that the parallelization strategy in this chapter is presented at a fairly high level, glossing over many of the programmatic nuances of the actual software (such as specific data elements and the tricks necessary to ensure correctness in concurrent accesses to them). Such low–level artifacts that result in a performance affect or lead to specific architectural features will be discussed at their points of relevance.

4.1 General Principles

As discussed in previous chapters, parallelization of speech recognition is not a new idea in itself. We will therefor leverage techniques from previous work in our parallelization effort [76]. We diverge from such existing work in the programming philosophy we adopt to expose the concurrency available in this domain.

Previous efforts at parallelization of speech recognition applications were focused purely on performance improvement, and targeted toward large, multi-processor systems. As such, they employ standard parallelization techniques in which nearly all aspects of runtime concurrency are managed by the programmer, and the entire effort is tailored to a specific architecture configuration. We do not believe this is an adequate model for our architectural constraints. Rather, we adopt the philosophy that the programmer should expose all available (reasonable) concurrency to the architecture, and allow the architecture to dynamically manage the amount of exposed concurrency based on runtime constraints.

Adopting such a philosophy requires new methods of exposing the concurrency in the application space. Rather than specific directives to force parallel execution, the programming model must provide "suggestions", or delineate points where execution may be performed in parallel, but provide a means for serial execution as well. It must be possible to manage this in a fine grain, low overhead manner so that the resource needed to provide this information do not overwhelm the potential benefits. Finally, the process of exposing such parallelism must be convenient and intuitive to the system programmer. While numerous tools exist for automatically extracting potential parallelism in applications, the software programmer with a full knowledge and understanding of the software under development remains the best person to expose maximum amounts of useful thread level concurrency.

We believe that the most direct approach to providing such capability is by modeling the function call semantics common to current architectures. In such a concurrency model, special function call semantics could be used to denote "thread functions" which may be executed in parallel with other instruction streams. Information required for the new thread is entirely contained within the argument passing constraints of normal functions, and

undesirable concurrency can be ignored by performing a standard function call instead of spawning a new execution thread. Overall, this approach to parallelization can be thought of as a fork/join style concurrency model. The specific ways in which this model diverges from a standard fork/join implementation will be discussed in greater detail in later chapters.

This approach makes a number of trade-off in the requirements described previously. The very natural function call semantics aid intuitive programmer use, and the inherent ability to map such spawn commands to function calls allows dynamic, fine–grain, runtime control over the amount of actual concurrency exposed. On the other hand, most opportunities for concurrency are found through such operations as loop unrolling. Applying function call semantics constraints to such an operation, while neatly packaging the data needed for each separate execution thread, also has the potential to incur substantial overhead as a result of register save / restore and stack related operation performed by the compiler. While it may be possible to recover some of this overhead is certain situations, the requisite ability to dynamically map a thread spawn to a function call demands that such operations be present in the general case.

While the function call approach provides a method for exposing parallelism, a number of functions in Sphinx operate on or manipulate global data structures in one form or another. As such, some form of global locking capability is necessary to prevent race conditions. As such facilities are very common to parallel programs, we believe a simple hardware lock management system (structured to function in a generic way, as with pthread style mutex operations) would be beneficial to any programs running on such an architecture. As such, we assume the existence of some fixed number of hardware locks to manage mutual exclusion as we discuss Sphinx parallelization.

4.2 Gaussian Score Evaluation

As discussed in the last chapter, the Gaussian score evaluation phase of Sphinx operates in two steps that can be separated from each other. The first involves identification of the appropriate set of Gaussian distributions for the current feature vector set. The second

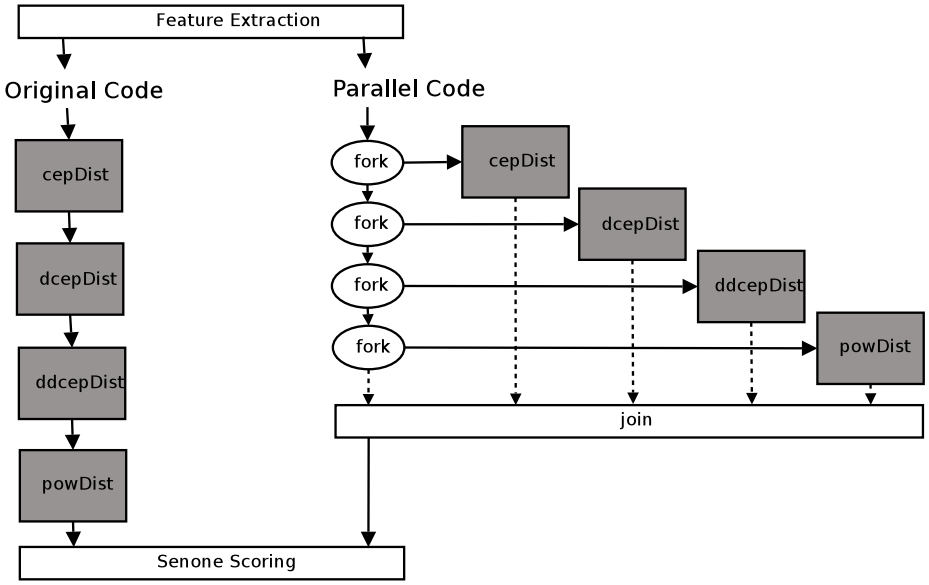


Figure 4.1: Parallelization of Gaussian Scoring PDF Selection - PDF selection for each feature vector type is performed through a series of transformations and computations. The necessary computations for a given feature may be performed concurrently with those of other features. This example depicts the overall parallelization of this phase utilizing the fork/join aspects of our programming philosophy.

involves processing the acoustic information for each of the currently active senones in the trained SMD set, and applying the selected Gaussian distribution functions to the input feature vector to produce a probabilistic match score for the input vector against each active acoustic entry. These two steps must clearly be ordered with respect to each other, though opportunities for concurrency exist in each individually.

The key opportunity for thread level concurrency in the PDF selection phase of operation comes from the set of four feature vectors utilized by Sphinx. The transformations performed upon the input data within this selection process for a given feature involve a number of floating point manipulations and distance computations which may expose the opportunity for some instruction level parallelism, but no segments of independent code large enough to warrant the generation of a new thread. The transformation computations on each feature set, however, are independent of the computations of the other three. Thus, given an architecture with processors capable of the necessary floating point computations, it would be possible to process and select PDF distributions and perform necessary

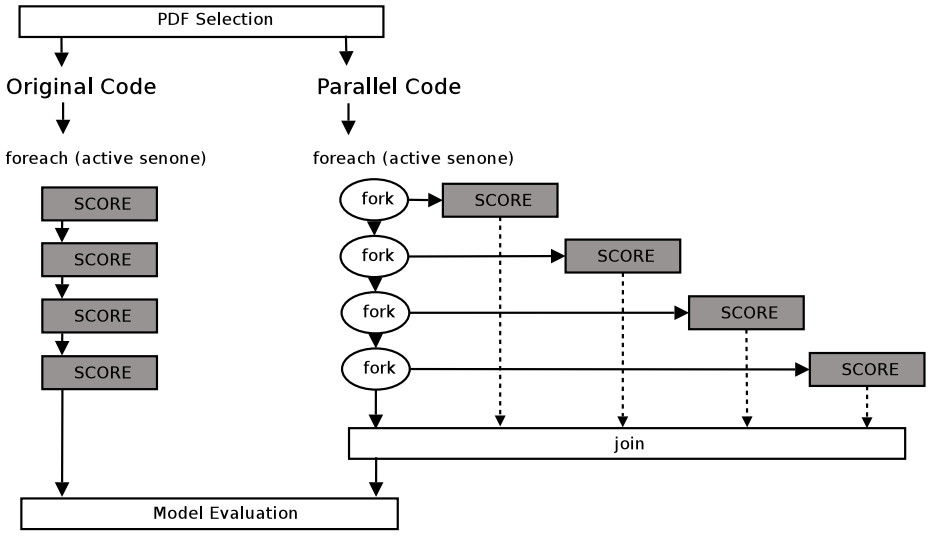


Figure 4.2: Parallelization of Senone Scoring - Once PDF selection is complete, all active senones are scored against the new input data. The evaluation and scoring of each individual senone is conceptually independent from that of other senones, and thus may be performed concurrently (note that the parallelized version shows conceptual parallelism in program flow, not a timeline of overlapping computation).

transformations on each feature concurrently. This parallelization strategy is depicted in Figure 4.1

The actual senone scoring phase of Gaussian scoring is essentially a for–loop which iterates over all currently active senones. As discussed previously, a senone is marked as active if any currently active search tree node references it. As acoustic scores are independent of search tree parameters, a senone need only be scored once, irrespective of how many active search tree nodes refer to it. Furthermore, with PDF selection complete, there is no dependency between scoring operations for individual senones. Each is considered with respect to the original (transformed) input feature set in turn. Thus, it becomes a fairly trivial matter to expose the concurrency of this operation by spawning a new thread for each senone that must be evaluated. This strategy is depicted in Figure 4.2.

4.3 Linguistic Model Evaluation

As linguistic model evaluation requires the complete senone score array for the current iteration, it is necessarily a disjoint operation from the scoring of senones for the current iteration. It should be noted that linguistic model evaluation of a given iteration can successfully be overlapped with Gaussian scoring operations on future frames of the search input. Such an approach would, however, require breaking the feedback path of "active" senones generated during linguistic model evaluation, as such information would not be available for future frames. As a result, all senones would have to be scored, dramatically increasing the computational demands of the Gaussian scoring phase. While this is a viable strategy under certain circumstances (Mathew et al utilize such a strategy in the design of their Gaussian scoring ASIC component for Sphinx–3 [65]), it is impractical for the more generic architecture we focus on in this work. As such, we assume a barrier between the Gaussian scoring and model evaluation steps, ensuring that a complete score array is available before linguistic model evaluation begins.

Evaluation of the linguistic model in Sphinx–2 occurs in a slightly disjointed fashion due to the need to produce and evaluate correct inter–word context channels. Recall from the last chapter that these dynamically created channels allow modeling of the last phonetic elements of words for co–articulation purposes, but are not maintained in the static search tree due to the substantial size burden incurred by static expansion of all such channels relative to their infrequent use. Furthermore, the evaluation phase passes through distinct steps of active channel evaluation and active channel pruning. The pruning threshold utilized during this second step is selected based on the best score seen during the evaluation phase, maintaining a "beam" of constant probabilistic width, regardless of the actual hypothesis probabilities currently seen in the search process.

The result of all of these algorithmic constraints, and our general goal of avoiding creation of concurrency that results in massive data collision / communication problems, is the introduction of three concurrency barriers in the model evaluation phase. The first separates model evaluation from pruning, ensuring that all active channels have been evaluated

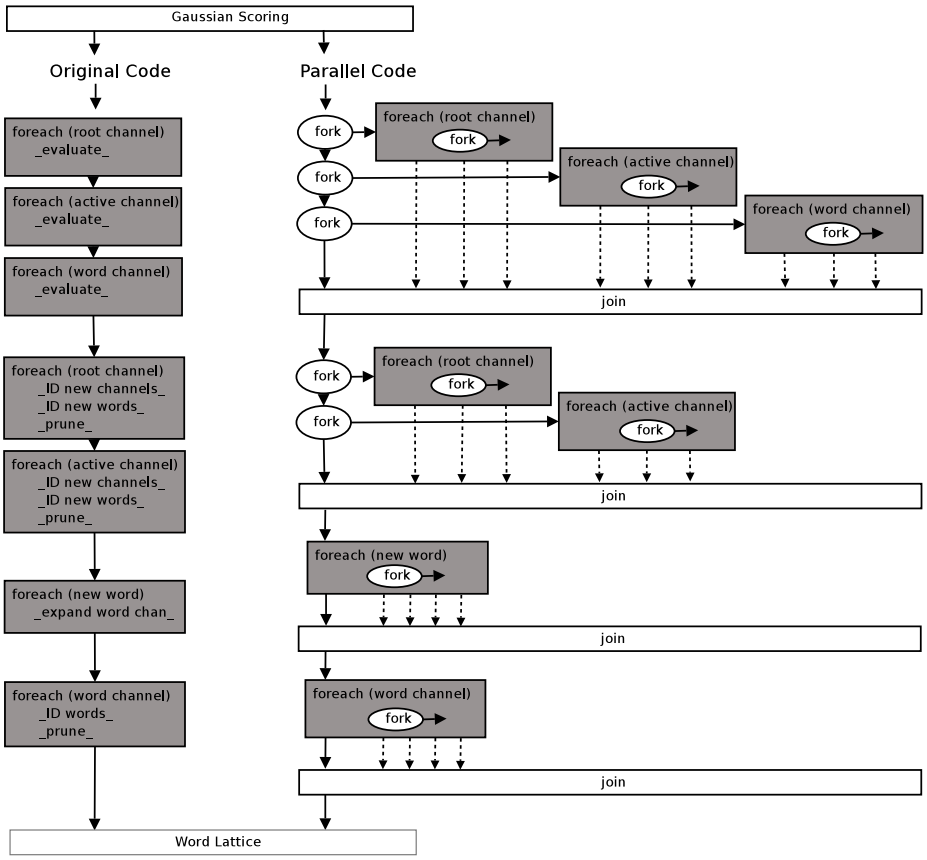


Figure 4.3: Parallelization of Linguistic Model Evaluation - This figure depicts the overall concurrency strategy for the linguistic model evaluation phase of search, relative to the sequential model as depicted in earlier figures.

and the current best probabilistic score is accurate before the pruning threshold is select. The second occurs at word–channel expansion. Once again, due to the nature of Viterbi search, new inter–word channels must be expanded before word–channel pruning can take place, as a newly identified potential word may alter the current score of an existing inter–word channel. The third actually occurs as a side–effect of, and between the first two. As potential new–word identification occurs during the pruning of non–word channels, and as all such potential new words must be identified before expansion (again, to capture best paths through newly created word channels), the first pruning phase must be completed before word channel expansion can be performed. Note that, depending on the particular workload distribution and parallelization technique used, it may be necessary to perform memory synchronization activities at these concurrency barriers.

Given these constraints, the linguistic model phase may be parallelized as depicted in Figure 4.3. The evaluation portion of the computation may be parallelized almost completely across individual channels. As each channel's internal evaluation process need only occur once during this step, much of this work can be performed without concern for race conditions or data hazards introduced by concurrency. The need to acquire a global best score for a given iteration, however, does demand some amount of coordination between potential threads. Channel evaluation is concluded by a global join operation, forcing the system to await complete evaluation before proceeding to pruning, and ensuring that the global best score is correct before setting the pruning threshold for the next stage.

The first pruning stage is managed in a way very similar to evaluation. In this case, however, there are a number of subtle data contention issues that must be observed. The most obvious of these comes from the need to track newly activated channels and newly activated words. This identification process requires modification of global state (active lists for the next iteration), as well as other potential colliding references (setting the best score for a channel in the next iteration). The primary issue here is manipulation of the active channel and new word lists, which must be performed as atomic operations from the viewpoint of a thread to ensure correctness.

Once these steps are completed, the "new word channel expansion" and "word channel pruning" phases parallelize fairly directly. The channel expansion phase contains a number of global data accesses, global list updates, and other such tracking operations that have not been discussed in detail here. Much of this comes from the need to track candidate words across multiple frames of search in order to identify the best possible word transition point, and thereby the most likely choice of actual words. As a consequence, this phase of computation incurs the potential for significant lock management and passing overhead. The word channel pruning section, by contrast, is very similar to the previous pruning steps, and hits many of the same problems.

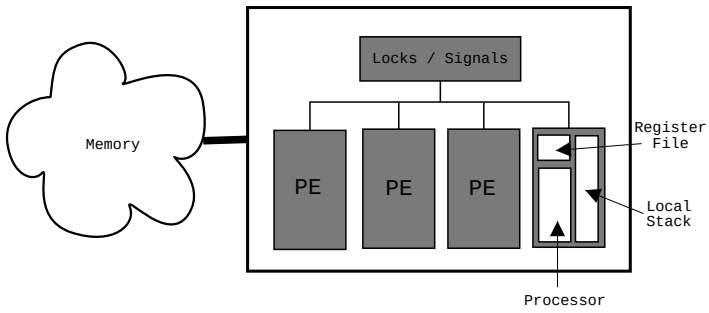


Figure 4.4: Simple MP-on-chip Model - We utilize this basic model for our initial evaluation of parallel speech recognition performance. Each "Processing–Element" constitutes a basic in–order SA-1110 core with possibly multiple register contexts and some local stack space. Communication with memory and global locks / signals is performed through a single bus–like interconnect.

4.4 Performance Implications

In order to explore the trade-offs and problems with this concurrency model, we perform a first–round evaluation with a basic architecture. This section considers the performance of sample speech recognition runs with Sphinx, parallelized using a partial form of the concurrency model discussed thus far (but drawing significantly from existing techniques for managing concurrency), and utilizing a basic multi–processor, multi–threaded architecture. The results of this first cut evaluation are used to refine our model and develop intuition for the actual architecture and programming model implemented in this work.

4.4.1 Architectural Model

In order to develop some general intuition regarding the effectiveness of our programming model and architectural philosophy, we perform this initial analysis on a very straightforward architecture with little specific optimization (depicted at a high level in Figure 4.4). We begin with a standard Intel SA-1110 processor implementing the complete ARM instruction set, with the specific inclusion of FPA style floating point extensions. This provides a good, current generation embedded processor architecture and ISA from which to perform evaluations.

To provide resources for concurrent computation, we expand this single processor model

in two ways. First, we replicate the base processor, assuming a generic interconnect model to allow multi-processor computation. At a very basic level, this provides duplication of resources to take advantage of available concurrency. As a second enhancement, we add a fixed number of hardware thread contexts to each processor, and assume that each processor is able to dynamically select a context to execute from for each given cycle. Due to the in-order nature of the execution core in our infrastructure, these multiple thread context do not provide for simultaneous execution, but rather are intended to help tolerate system latencies. For this baseline model, we assume a per–cycle round–robin selection process between all actively running contexts on a given processor.

We provide for the need to manage critical sections, inter-thread signaling, and mutual exclusion through a series of hardware locks and signals accessed by ISA extensions. The "hardware locks" are generic in nature, identified by an ID number, and may be constructed with a set of small flag registers (to maintain the current lock state), a bitfield to track thread contexts currently waiting for each lock, and some simple selection logic to transition the lock between requesters. Signals may be constructed with similarly unsophisticated hardware, providing the ability for a given context (or set of contexts) to stall and await an activation message. This signaling infrastructure may be easily modified to allow a "synchronization" operation mode, providing a hardware facility for global barrier synchronization operations. Access to these global locking and signaling systems is centralized and serialized, but we assume no other overhead (communication, bus contention, etc.) for their use. We assume 15 locks and 15 signals, matching the available instruction bit field size used for register accesses on the ARM ISA.

Finally, the memory model of this evaluation system is a simple fixed delay of 30 processor cycles. Given a processor clock rate of 200MHz, this constitutes a fairly slow, high–bandwidth memory system, and is sufficient for the resolution of detail required at this stage of the evaluation process.

4.4.2 Parallelization Model

For the purpose of this initial study, we adopt a concurrency model that is somewhat less aggressive than the model discussed in the previous chapter, and more in line with conventional multi–processor parallelization schemes. Rather than explicitly exposing all concurrency to the processor, we adopt a "processor ID based workload distribution" model akin to what one would use on a normal shared memory multi–processing environment. For example, rather than simply mark potential concurrency, we spawn a fixed number of threads to match available processors and hardware contexts, and then use synchronization and coordination systems to dynamically distribute work to each thread in a deterministic way. This parallelization will be described in detail here.

We begin once again with the Gaussian scoring computation phase. As discussed in the earlier chapters, PDF selection on each feature type can be performed concurrently. Thus, the initial phase of iteration processing begins by spawning off threads to perform these operations, placing each task on a single processor if possible, and on multiple contexts if not. Thus, for a two processor model with two contexts, each processor would process two of the four features, one on each context. A four processor by two context model, however, would distribute the work to each processor, leaving each with a empty context. Finally, a two processor, single context model would simple require each processor to compute two features sequentially.

Once PDF selection has been completed, senone scoring is performed by distributing active senones across available processing resources for evaluation. This is done essentially statically through standard programmatic techniques such as having each thread context start at an offset in the list of active senones determined by it's context ID, and skip *num_contexts* entries in the list to select the next senone for evaluation until reaching the end. As this phase of computation is naturally distributed, it is possible to perform much of this without any locking systems.

The model evaluation steps are similar to senone evaluation, but require a slightly higher degree of sophistication. Evaluation itself may be performed nearly concurrently, utilizing

the same node selection / work distribution approach used for senone selection. The one major difference is the need to track the overall best score, and have that score available at the end of the evaluation process. As this operation is not ordering dependent, we eliminate the need for explicit locking by maintaining a local best–score value for each context, and gathering all such values to determine the global max in a post–processing serialized step.

Node pruning presents a somewhat more challenging set of obstacles, particularly for the data locking model utilized by this architecture. The primary problem here is that successor nodes could potentially be activated my multiple ancestor nodes, and only the highest probability entering path should be kept. The same is true of potential new word identifications, which must not only reflect the highest probability activation, but are maintained in a global, dynamically allocated data structure for later per–frame processing and word channel expansion. With a limited number of hardware locks, providing the fine–grain locking capabilities needed for this phase of computation proves rather challenging. We are forced to employ a two stage locking solution, where the hardware lock is used to serialize accesses to a new global data structure that maintains individual software lock states for each node in the search tree, achieving the desired exclusion zones at a high locking and communication overhead.

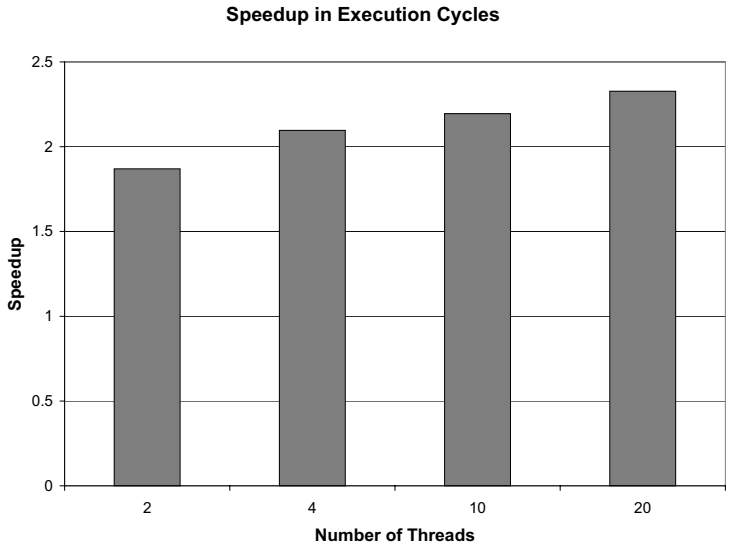
The expansion of new word channels presents problems comparable to channel pruning. This phase of computation not only tracks potential candidate words for this frame, but also tracks potential start frames for these words, and tracks starting and ending points across multiple potential frames of input. The procedure for processing this data first involves sorting and re-organizing from a format amenable to modification by the search phase to a format amenable to analysis across multiple start / end frames. The only natural data partitioning evident in this search component occurs along individual words. Thus, this section is parallelized by allowing each context to acquire work based on a list of relevant word–IDs, and performing all the processing necessary for each word. Unfortunately, substantial overhead must still be incurred through data locking and serialization in order to manage some of the fine–grain exclusion aspects of this phase.

Once data ordering and sorting has been completed, the actual expansion of new potential candidate words may proceed using the same word–based partitioning used during the sorting process. It requires considerably less modification of global data, and thus can proceed more freely in a concurrent environment.

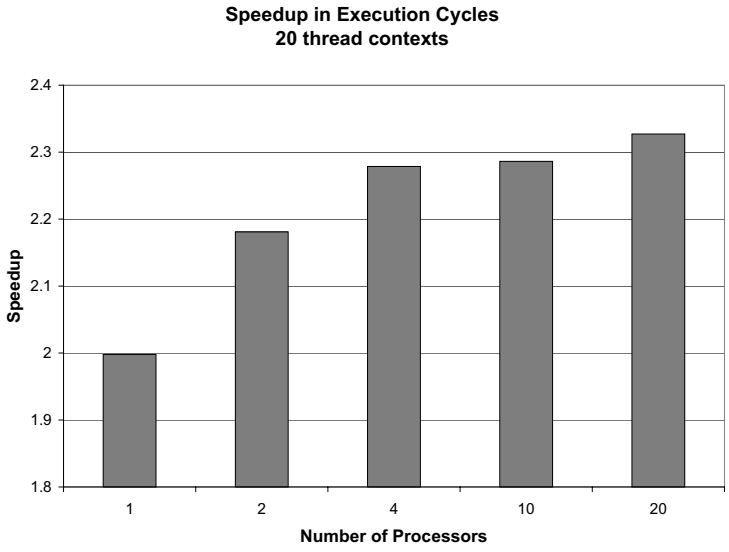
Pruning of all word channels (performed after expansion) achieves goals similar to previous pruning steps, but the nature of these word–channels is conceptually different from others in one important respect: any word channel for a given word that reaches the required word identification threshold potentially activates that word. Thus, it is necessary in one sense to maintain the partitioning along word based constraints, and it is further necessary to essentially collect a "per–word" best–score value (the actual path to activate a word in a given iteration is the one that activates it with the highest probability). Furthermore, these activated words are entered into a global word lattice and back pointer table, which are used post–search to backtrack words for the highest probability path (to identify the recognition hypothesis). As such, the demand for atomic access to global data structures is much higher in this pruning phase than in previous pruning sections. We attempt to alleviate some of these issues by allowing each context to maintain it's own "word identification" list for a given frame, and collecting all activated words into the global data structure in a post–processing step. While this reduces lock contention during concurrent execution, it adds considerable extra computation and overhead at the end.

4.4.3 Performance Implications and Derived Intuition

We perform the overall evaluation by implementing the described architectural model in a parallel implementation of SimpleScalar/ARM, and utilizing ISA extensions to incorporate instructions necessary for thread management and computation. High level performance results for sample runs of this architecture and parallelism model are depicted in Figures 4.5(a) and 4.5(b). Figure 4.5(a) shows the performance effects of multiple contexts on a single processor. We see that the addition of a second thread context results in a nearly 2x performance improvement (over a single context), but that the benefit of sub-



(a) Varied Contexts



(b) Varied Processors

Figure 4.5: Speedup vs. Unparallelized System - These graphs consider speedup in execution cycles versus the original unparallelized implementation for a varied number of hardware thread contexts (a) and complete processor pipelines with 20 hardware contexts each (b). The performance points are related to interactions between available concurrency, tolerable latency for a single processor, and thread scheduling.

sequent context additions is fairly minimal compared to this initial improvement. Similar performance improvement patterns are shown in Figure 4.5(b) for a fixed number of per–processor thread contexts, but varying the number of actual available processors. Overall, neither set of results demonstrates a performance improvement comparable to the amount of added processing resources.

A more detailed analysis of the underlying components leading to this data, however, reveals a number of very important performance affecting factors that we will use to improve the concurrency and architecture model for the complete architecture developed in this study. We identify a number of particularly important architectural properties suggested by these results, and consider each one in the remainder of this section.

Fine–Grain Locking

One of the key observations from our parallelization efforts and from the underlying components of the performance numbers presented here is the need for low–overhead, fine–grain locking support. One of the major constraints found in the performance results of this evaluation is the time spent acquiring and passing global locks (despite the low overhead hardware locking infrastructure provided). This problem is particularly evident in the pruning phases, and points to a need not for a generic locking mechanism, but rather some specific functionality for providing extremely fine–grain mutual exclusion on access to specific nodes or specific data. Furthermore, this mechanism must be inherent in the concurrency model, as explicit lock identification, acquisition, and release lead to significant overhead in program execution.

Static Data Partitioning

Related to the problem of fine–grain locking is that of generic data partitioning. The competition among all running processors / contexts for global resources such as global lists and score values, and the exclusion techniques needed to manage access to such values was also a major contributor to the overhead of concurrent execution and the poor perfor-

mance. While standard parallelizing techniques were employed to statically distribute data to hardware resources, these programmatic techniques were generic, and did not really take into consideration some of the natural opportunities for static workload division available in the application. A better static partitioning, followed by distribution of global resources, may do much to reduce contention and improve performance.

Thread Scheduling

There are two components to how work is scheduled onto hardware resources in this trial architecture that demonstrate a significant impact on performance. The first is inherent in the programmatic workload partitioning methods used. As a result of these techniques, it was not possible to account for run–time workload imbalances by redistributing work that was "supposed" to be performed on one particular thread context or processor onto another one. Indeed, the static functionality of the architecture did not even allow it to adapt concurrency resources to match the available concurrency in the application at any moment in time. This inflexibility is particularly detrimental to a non-deterministic, input driven application such as speech recognition, because it is impossible to tell at programming time how the workload should best be distributed (the best distribution being highly related to the current state of the search process). This suggests the need for a more flexible architecture and programming model, giving credence to our original goal of designing a system in which all concurrency is exposed, and the architecture dynamically select the amount of exploited concurrency based on run–time constraints of the system.

The second major flaw discovered in this thread scheduling algorithm was the use of round–robin scheduling of thread contexts. While this seemed to be a logical starting point (and had the advantage of requiring minimal control logic), it does not achieve the intended goal of effectively hiding system latencies incurred by any of the specific threads. Rather, this thread scheduling policy causes all threads to operate in lockstep (and they are all operating predominantly on the same code). As a consequence, all threads work through their computational sections at nearly the same time, and all threads issue their memory

requests at nearly the same time. Thus, rather than distributing memory demand and utilizing the computational section of one thread to mask the memory access (or other) latency of another thread, this scheduling model caused massive bursts in memory demand, and left all thread contexts simultaneously awaiting servicing of memory requests or other latency causing operations. This problem suggests the need for slightly more sophisticate thread selection logic that allows a single thread of execution to flow until it reaches a long–latency operation, and then switches to another one.

ISA Revisions

Our analysis reveals that, except for the PDF selection portion of Gaussian Scoring, the remainder of operations performed by thread contexts in our model utilize only a small fraction of the functionality available in the ARM ISA. For example, they make no use of floating point computations, and due to the translation to logarithmic computation do not even make use of integer multiplication and division operation. Indeed, an analysis of instruction frequency for other components of the search process suggest that integer arithmetic, logical, memory access, and control operations would be sufficient for most of the program execution. Earlier theoretical analysis of stochastic search applications suggests a strong motivation for such simplified computation due to the inherent complexity and computational demand of the application space, so other stochastic search applications are likely to show similar characteristics. This, and the realization that complete duplication of the main system processor in not a likely configuration for a hand–held or portable device, lead to the conclusion that parallel resources should likely be provided as more of a "master–slave" or "processor to co–processor" framework, rather than the hardware parallelization scheme used in this implementation. They further suggest that the processing elements of this "speech co–processor" could be significantly different, and notably less sophisticated than the computational resources of the main processor.

Beyond this potential ISA simplification, there are a number of ISA operational extensions that would likely be very helpful in performing many of the global operations executed

in Sphinx. For example, it is clear from the repeated "bestscore" identification processes that the ability to perform efficient global "MAX", "MIN", and "ACCUMULATE" type operations in an atomic fashion would not only improve program performance, but would also ease parallelization and reduce demand for locking resources.

CHAPTER 5

Architectural Models

Based on the programming model, parallelizability of Sphinx, and first round of derived intuitions, we begin making revisions to our basic parallel architecture to improve the performance of speech recognition. In this chapter, we will present two such revisions. The first includes some basic enhancements from the simple model considered in the previous chapter, including some scheduling and ISA enhancements. We will then consider the performance implications of these modifications, and introduce some first round power estimates to begin gaining some intuition with respect to overall energy consumption.

We then once again utilize the knowledge gained from this implementation to develop a final architectural revision. This final model will be evaluated in the following chapter along a number of architectural constraints, and will be the basis of the remainder of evaluation in this work.

5.1 First Architectural Revision

This first revision of our architectural model adopts some of the intuition developed in the previous chapter with our evaluation of concurrency in the Sphinx speech recognition system. We adopt a programming model more consistent with the "expose all concurrency" approach previously mentioned, introduce revisions to support more fine–grain locking and mutual exclusion support, and incorporate some capabilities for atomic operations useful

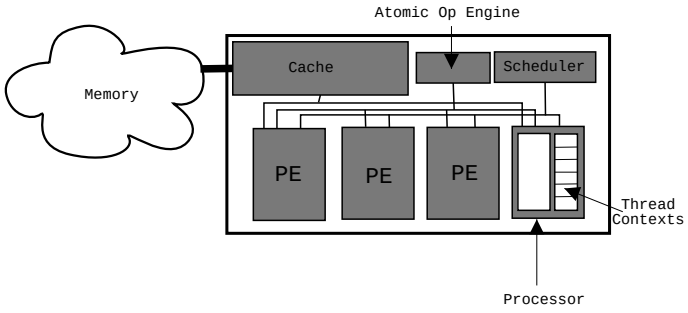


Figure 5.1: Overview of First Architecture Revision - This figure depicts an overview of our first set of revisions from the simple parallel architecture discussed in the previous chapter. Main variations are the move from fully replicated processors to a unified cache, unified global resource model. We incorporate a global thread scheduler to dynamically manage exploitation of concurrency, as opposed to the static, programmatic distribution used in the previous model.

to the program.

5.1.1 General Organization

An overview of our first architectural revision is depicted in Figure 5.1. In this model, we move to a more current generation Intel XScale processor [2] model running at 400MHz, but drop the assumption of complete processor replication inherent in the previous model environment. Rather, we assume replication only of datapath and functional units, moving to centralized cache and other global resources. The other primary differences of this architectural model over the previous implementation are the introduction of a more sophisticated hardware thread scheduling infrastructure, and the addition of functionality to perform certain global atomic arithmetic operations.

The introduction of the global thread scheduler is based on a more sophisticated concurrency description framework. In this model, we move to the full function call based approach to exposing parallelism described in earlier chapters, and utilize the global thread scheduler to determine if new thread spawn requests should be sent to available thread contexts or converted to function calls and executed on the existing running context. The thread scheduler adopts a load balancing policy across the execution pipelines, distributing new work to those processors that have the fewest active jobs at request time.

The basis of this advanced workload distribution system is a new concurrency model with inherent support for fine–grain mutual exclusion, a quality deemed vital to improving performance based on previous analysis. In this model, one field of the function–call–esque thread spawn instruction is reserved for a "exclusion ID" (EID). The scheduler ensures that no two active threads of execution have the same exclusion ID value. As such conflicts should be fairly infrequent, this implementation stalls the spawning context if an attempt is made to spawn a thread with an in–use exclusion ID. In practice, this ID value can be any register–sized value, allowing the base memory address of individual data elements to serve the purpose. This provides a very natural means of ensuring that access to any given node or data element in the knowledge base is strictly ordered in some way (though it places no specific constraints on what the final ordering will be). In order to support this capability, both the individual thread contexts and the global scheduler maintain a copy of the ID field for all active contexts. When a thread completes and terminates, it issues a termination message to the scheduler, freeing the ID for future use. At such time, any contexts stalled due to ID conflicts are released (one at a time). It should be noted that we maintain a special exclusion ID value of "0" for threads that do not require exclusive data access or are inherently safe from potential race conditions. Thus, multiple threads with an exclusion ID of zero may be in–flight simultaneously.

The other major addition is the "atomic operation unit", which is responsible for handling global synchronization operations (such as locks and signaling as from the previous architecture), as well as certain global arithmetic operations. While lock and signal management was discussed with the previous architecture, the arithmetic operation manager is new. It essentially represents a set of special global registers that can be set with values during initialization, and may then be updated over the course of parallel execution through special ISA extensions. These extensions consist for the moment of a "ADD" and "MAX" operation. The new instruction sends a single register value to the atomic operation unit along with a global register ID and a control code to select the operation. The operation engine then performs the requested operation on the selected register in an atomic fash-

ion, eliminating the need for much of the lock passing that constrained performance in our baseline implementation.

In the more general sense, this revision of our architecture moves much further toward a dynamic Chip–Multi–Processor/Multi–Threading (CMP/MT) system, in which explicit programmatic control of concurrency is replaced with a more responsive hardware scheduling system. The intent behind this architecture is to follow the insight found in other works [84, 43] that such combined systems can provide higher performance improvement for a given degree of energy consumption that conventional architectures. This performance / energy benefit comes from improved utilization of execution resources and a better match to the underlying program requirements. Thus, our revised architecture attempts to improve efficiency by dynamically utilizing extra thread contexts to tolerate processing delays, without committing fixed quantities of work to each context and incurring the potential penalties of workload imbalance that occur.

5.1.2 Programming Model

In order to take full advantage of the capabilities presented by our revised architectural concurrency model, it is necessary to re-evaluate our programming model and algorithm parallelization efforts. The inherent support for fine–grain data locking by the exclusion ID field of the spawn instruction in this architectural model allows us to present concurrency in the application much more aggressively, with substantially less need for programmatic lock management.

We begin by revisiting the Gaussian Scoring phase of computation. It remains necessary to partition the first PDF selection phase of this section along feature vector lines. Thus, the parallelization of this stage is essentially the same as in the previous implementation. This time, however, rather than a static thread assignment by the programmer, the main thread spawn each feature vector computation as a new thread with an EID of zero. Thus, the actual location of execution of each of the four feature vector threads (new context vs. new processor vs. function call mapping) is determined at runtime by the set of available

hardware resources and the thread scheduling policy. The senone scoring segment of the Gaussian Scoring phase, as discussed previously, is trivially parallelizable due to natural algorithmic partitioning of data accesses. While assignments were performed statically in the previous programming model, in this instance we once again simply spawn off new threads (with EID's of 0) for each senone scoring operation, allowing the dynamic scheduler to manage resource allocation.

The linguistic model evaluation phase is where the benefits of our revised programming model are best demonstrated. In general, due to the inherent fine–grain locking support, it is possible to issue evaluation / pruning of all channels simultaneously, utilizing the memory address of the channel in question as the EID of the relevant thread. By this approach, the architecture has some ability to work around conflicts by having substantial quantities of exposed parallelism from which to operate, the scheduling model ensures consistency in access to data structures from multiple threads, and the programmer can by in large ignore the issue entirely. The addition of the atomic operation unit further assists the programmer, allowing easy tracking of runtime variables such as the current best probabilistic score without the need to programmatically maintain locks.

Unfortunately, this model does not completely eliminate the need for global data locking. Pruning and word expansion operations that involve such steps as updating lists of active channels for future iterations must still be performed on global data, and are not represented in a format amenable to the fairly simplistic functionality provided by the atomic operation unit. Thus, while we are able to spawn threads almost indiscriminately, letting the architecture manage consistency issues to data nodes, many of these threads (particularly in the pruning phases) must still perform global locking operations as they manipulate such global data structures.

5.1.3 Performance and Power Evaluation

As with the initial parallel implementation, we consider the implications of our architectural additions by evaluating performance on a simulated infrastructure. Once again, we

develop a simulator based on a multi-processor implementation of the SimpleScalar/ARM toolset, and utilize ISA extensions to access the added functionality available on our system. As this evaluation model uses a more practical architecture than full replication of the processor, we are able to further extend our analysis by incorporating power / energy considerations. Since a full–system implementation of this architecture is unavailable, we generate workload energy estimates by combining energy data from a number of data sources, prorating based on usage data from simulation. We approximate power for the processor core by performing area scaled power estimates of execution resources of a base XScale processor (using an XScale die photograph and worst case power figures from the PXA250 datasheet [2]). Cache energy and thread context register file energy are estimated using the Cacti cache modeling library [89]. Memory system energy estimates are generated utilizing the SDRAM system power estimator available from Micron Technologies [4]. This power estimator considers active, background, precharge, and refresh energy for a single chip. The estimates were then multiplied across the number of DRAM chips required to meet size / bandwidth assumptions made by our simplified fixed delay memory model.

We will begin this evaluation by considering raw performance numbers, and evaluating the effectiveness of the new architecture and programming model. We will also consider sources of performance loss in this infrastructure, and consider methods of further improving overall performance. We will then introduce the workload energy estimates, and consider the impact of improved performance on overall energy consumption.

Multithreaded Performance

Raw performance speedups for a number of processor / context variants relative to a single processor / single context implementation are depicted in Figure 5.2. The performance results shown here clearly demonstrate the improved performance achievable as a result of our architectural revisions. This data also depicts the potential performance improvements achievable due to latency toleration across multiple thread contexts, and the peak performance achievable before overhead becomes a limiting bottleneck (at around

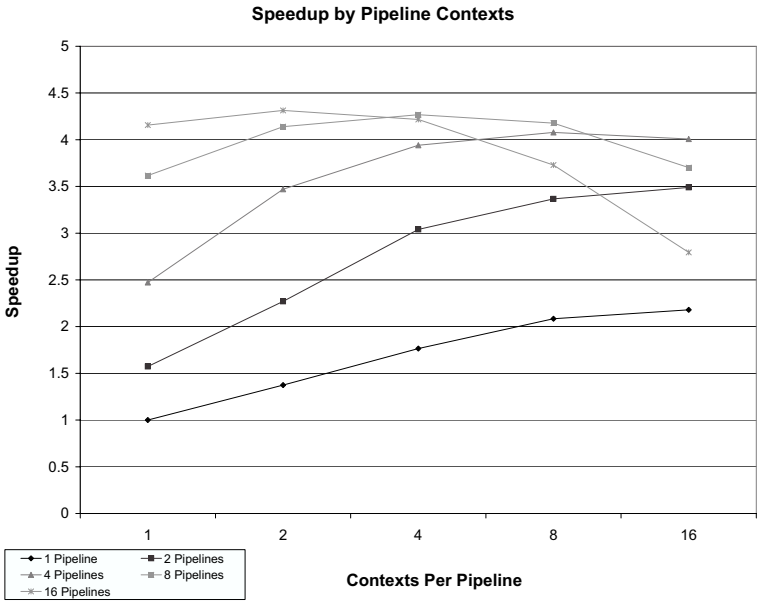


Figure 5.2: Speech Recognition Performance on Revision 1 Architecture - This figure depicts raw performance (relative to a single processor, single context) for an array of processor and context implementations. The potential performance benefits of multiple execution pipelines (for pure parallelism) and multiple execution contexts (for latency tolerance and better utilization) are clearly visible. The performance limits of this design are also apparent, as overheads due to communication, synchronization, and scheduling activities once again limit realized gains.

4.5x). We also observe that doubling the number of execution pipelines (which is a pure doubling of execution resources) leads to an approximately 1.5x improvement in realized performance, suggesting substantial overhead in our implementation. A more detailed analysis of the sources of overhead in this model reveals that concurrency production rate and concurrency management are major contributors.

Concurrency production rate constraints are a product of our programming model. While "worker" threads are distributed quickly across processors, our current model does not efficiently distribute the job of spawning such worker threads. Thus, the overall concurrency available at any given moment in time is constrained by the rate of execution progress along the context that primarily spawns threads. Considering that this context may be competing for pipeline time with other worker threads running on other contexts on the processor, the resulting limits on concurrency can be substantial.

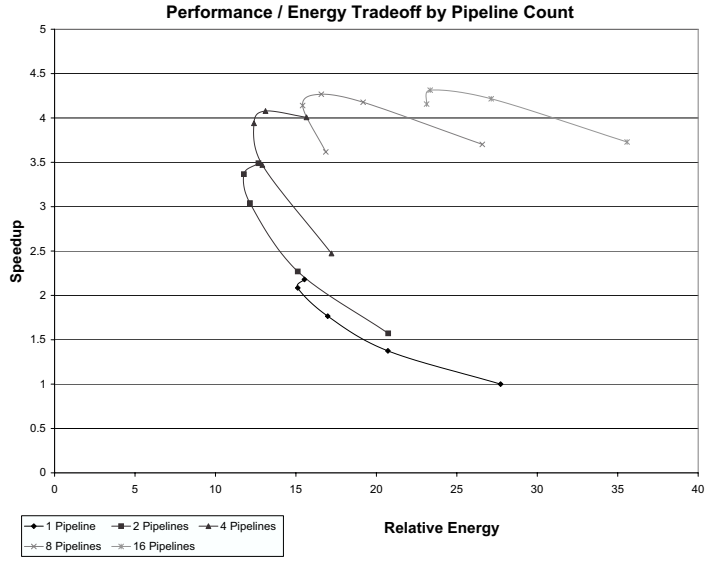
The major contributor to concurrency management overhead comes from the global

nature of the task scheduler. In order to guarantee fine–grain mutual exclusion across the entire architecture, all thread spawn attempts require communication out to the global thread scheduler, a decision by the scheduler, a response from the scheduler, and if necessary a third communication to send the new thread to the context upon which it will execute. While a few programming modifications were able to alleviate some of the "concurrency production" issue, the act of doing so only exacerbates this problem. Thus, these two overhead contributers in some sense trade off with each other, limiting performance despite better programming efforts.

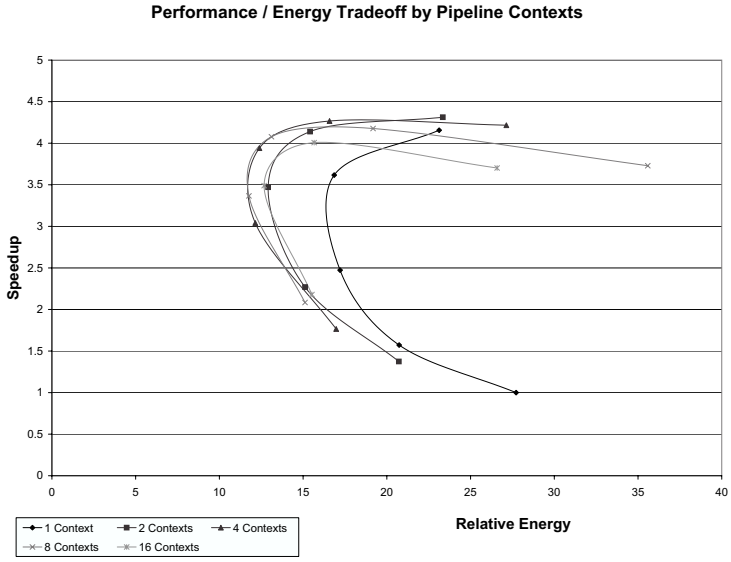
Performance / Energy Tradeoff

While the performance data discussed thus far demonstrates the potential of well managed concurrency in improving performance, it is important to keep the target application domain of low power portable devices, and the power / energy constraints that go with them, in mind. To this end, we consider overall system energy consumption for our architectural variants, in order to gauge the practicality of this approach. The energy estimates presented here are based on data sources discussed in previous sections. We recognize, however, that a comparison of instantaneous power consumption would be uninteresting. Increasing the hardware resources of a processor will, by necessity, increase the power consumed. A more telling metric for our purposes is the amount of total energy consumed in order to complete a given speech recognition task. While this does not directly address factors such as total running time in the same way that power does, it gives a much clearer picture of how much speech recognition can be performed within a fixed energy budget. In the end, this seems the more relevant metric.

To this end, we estimate total energy consumption for a fixed speech recognition task across a varied number of execution pipelines and a varied number of execution contexts. Both considerations are important, as adding an extra context has a much smaller impact on power than adding a full pipeline, but also results is less performance improvement. Since overall energy consumption is related to both power (how much are you using) and



(a) Tradeoff by Pipeline



(b) Tradeoff by Contexts

Figure 5.3: Performance / Energy Tradeoff for Revision 1 - The data point associated with each pipeline (a) or context (b) represent the number of contexts (a) or pipelines (b). The data point at the left terminus of each line represents one. The next represents two, then four, and so on. Note in (a) that due to reduction in runtime, the overall energy consumption for the same speech recognition task often with some addition of hardware resources. It is clear from (b) that 4-8 contexts per pipeline provides the best performance improvement relative to energy consumption in most all pipeline configurations.

performance (how long are you using it for), it is important to explore this tradeoff.

The results of these evaluations are presented in Figures 5.3(a) and 5.3(b), and depict performance as it relates to relative energy consumption. As we are utilizing energy and power estimates from a number of sources, we feel it is inappropriate to claim actual energy consumption, but that relative energy between experiments computed by the same means will depict an accurate general picture of our energy effects.

The most obvious and interesting characteristic of both of these result graphs is that added computational resources leads to distinct reductions in overall energy. To reiterate, this does *not* correspond to reductions in power consumption. The effect we see here is directly related to increased efficiency of computation. Despite the use of higher power computational elements, the far shorter time for which these components are active leads to a reduction in the overall energy consumed to compete the task. Overall, this data seems to show that a configuration of around 4 processor and 4–8 contexts provides the best performance relative to energy consumed, and the performance increases beyond this do not offset the added computational power required.

Observations and Analysis

We once again make a number of observations regarding the performance of this new model, and what techniques should be considered to improve its performance. The most fundamental observation at this stage is the need to better distribute work. While this architectural model begins implementing the lessons regarding workload distribution taken from the previous evaluation, it is clear that a more aggressive approach is necessary. A technique that provides some baseline workload distribution, allows individual processing elements to identify segments of the work load that belong to them with minimal computational effort, and better distributes the actual work of exposing concurrency is necessary to eliminate some of the global constraints that seem to limit performance here.

The underlying data demonstrates that use of an atomic operation type system for max and accumulate operations is quite effective, however our implementation leads to

communication and resource contention. In keeping with the theme of further distribution of workload, we believe operations that can be performed piece-wise through scatter / gather operations (as "add" and "max" can be) should be performed as such. This provides further partitioning of work, and reduces contention on global resources.

As we further evaluate the power and performance data, and consider once again the analysis presented with the original evaluation regarding potential ISA and pipeline optimizations, we see that a better energy tradeoff may be possible by reducing the actual computational resources of the added execution pipelines. As discussed previously, Sphinx does not use functionality beyond basic arithmetic and logical operations for much of its execution, suggesting that floating point capabilities, and even multiplication capabilities may be unnecessary for many of the added execution contexts.

This leads us to a look at the practical implications of our architecture. It seems fundamentally unrealistic to believe that the main processor of a handheld system would undergo extensive modifications to support our concurrency model in a real world environment. The far more likely scenario is that of a "speech coprocessor" tied to a standard embedded master processor with a few specific extensions necessary to support the added hardware. The power analysis suggests that such an architecture, by reducing the ISA and capability complexity of the processing pipelines in the co–processor, could further benefit from the performance / energy dynamic depicted here.

5.2 Final Architectural Model

This section presents the final architectural model we will employ throughout the remainder of this work. It is based on intuition and knowledge gained from the previous two evaluations of parallel speech recognition on embedded system architectures. The most fundamental paradigm shifts in this model are a "master–slave" configuration of a main system processor and a speech coprocessor, and the use of profile–based static data partitioning in order to address the work distribution issues seen in the previous model. This chapter will present an initial performance / power analysis, as was done in the previous section.

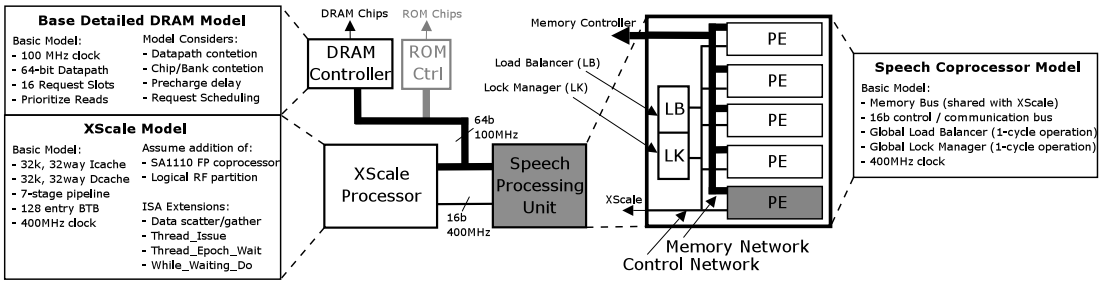


Figure 5.4: Overview of Final Architectural Model - A top level view of our final architectural model is shown in this figure. Key components are the use of a baseline XScale processor which is separated from the speech processing unit through a main–processor / co–processor model. The speech co–processor itself is made up of a number of processing pipelines and some centralized resources for dynamic load balancing and global locking operations.

Further detailed architectural analysis will be performed in following chapters, once we have established the benefits of this model configuration.

5.2.1 General Organization

A high–level view of our overall proposed system architecture is presented in Figure 5.4. The major features at this level of organization are a main system processor, a speech co–processor containing a number of parallel processing elements, a memory system interface, and a number of interconnects. These divisions are logical only, and do not necessarily represent chip boundaries.

The main system processor in our evaluation is modeled once again after a standard Intel XScale 400MHz processor [2], a reasonable model for a modern high–end embedded system. The XScale processor is based on the StrongArm ISA, the key features of which are a set of RISC style instructions, extensive addressing modes, and the ability to predicate any instruction based on the results of previous conditional tests. The basic XScale processor does not, however, possess floating point capabilities. While most of the floating point operations performed by SPHINX are during the DSP phase, the initial feature score generation phase of search requires floating point capabilities. As such, we assume the inclusion of an FPA style floating point coprocessor [40] similar to that used for the StrongArm class of embedded processors. While it may be possible to perform the necessary floating point

computations utilizing existing integer operations without a major impact on performance (due to our model, discussed shortly), we do not consider this option in our work.

Besides a set of ISA extensions to manage the speech co–processor, the only major modification to the main processor is the inclusion of a second thread context. As will be revealed shortly, our new model removes floating point capabilities from the speech processing unit, forcing all such computations (in the PDF selection phase) to be performed on the main processor. This reduces the proliferation of unnecessary resources, but also potentially serializes a number of execution steps. As we will discuss later, we utilize this second "background context" to allow the main processor to perform many of these computations ahead of the actual search phase being handled in the speech co–processor. By this approach, we are able to overlap execution, but allow these extra computations to take a back seat to the main concurrency producing thread.

Two communication infrastructures connect the XScale system processor to the speech coprocessor and to system peripherals. The first is a relatively small, low speed control bus connecting the XScale and the speech coprocessor only. This bus is primarily used to start speech processing, collect results, and perform a small amount of inter-PE communication within the coprocessor. The second is a larger, more sophisticated memory bus, interfacing to a standard DRAM memory system. This organization is developed from the understanding that relatively little inter-PE communication is required in our new programming model, while performance evaluations show that memory throughput is a key bottleneck.

The speech coprocessor consists of a set of execution pipelines (Processing Elements or PEs), which will be discussed in detail in the next section. It also contains a few resources to aid in parallelism management. These extra resources consist of a dynamic load balancer and a small number of global locks, neither of which require sophisticated logic.

One of the major modifications in programming style in this revision of our architecture is an initial static workload partitioning. This partitioning is performed upon the linguistic model tree utilizing profile based usage data to achieve a workload balanced set of partitions matching the number of processing elements available in a given version of our architecture.

This initial workload partitioning turns out to be quite effective. The need for the dynamic load balancing mechanism arises from the fact that static partitioning can only equalize overall utilization, allowing the potential for substantial imbalance within any single concurrent execution section (say during a single iteration). A sufficient accumulation of such small single instance imbalances can have a substantial overall effect on performance. Thus, the dynamic load balancing system monitors activity on each PE, and utilizes special commands to migrate certain classes of jobs from an overworked processor to an idle processor as imbalances arise. Busy / Idle information could be passed to the job scheduler over the command bus, but in our model we assume the addition of an extra interconnect line to pass this information. More precisely, the only relevant information is that a processing element can accept a job. Which specific PE is not relevant to the scheduling decision. As such, we assume a single global signal line that can be activated by any processing element, to indicate that a job acceptor is available. As we will discover in later evaluations, a shared bus is more than sufficient for the control network, and as job migration data is passed over this shared network, it may be possible to snoop migration requests and eliminate the job scheduler as an independent device entirely (allowing each PE to utilize local information and bus snooping to decide if a job should be sent out).

In keeping with the philosophy of distributed control, each PE contains facilities for performing fine–grain data locking (utilizing the same exclusion ID mechanism as in the previous architecture). These mechanisms allow for an easy method of achieving mutual exclusion on data elements within the scope of a processor's partition. The cross–partition exclusion provided by static partitioning ensures that this exclusion is global in effect. Certain operations such as global memory allocation or global data manipulation, however, still require the ability to enforce mutual exclusion across the entire architecture but are not accommodated by the partitioning. Thus, similar to the need for a global dynamic load balancer despite the initial static workload distribution, a global locking mechanism is needed despite fine–grain locking support. Once again, this infrastructure is used minimally, and thus has low performance pressure. Furthermore, the logic required to provide this

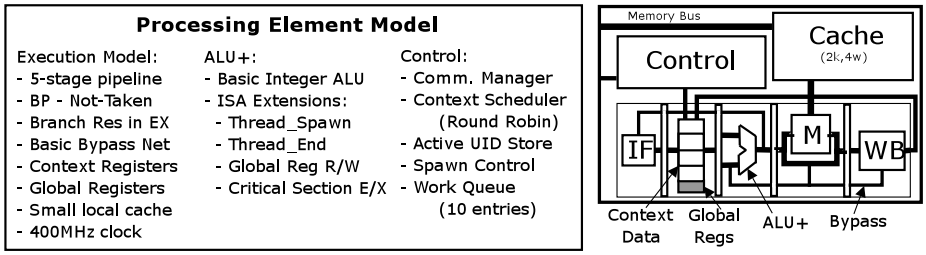


Figure 5.5: Details of a single Speech Processing Element - This figure depicts the details of a single PE from the previous figure. The actual execution resources of this pipeline are minimal, constituting a simple, in–order integer pipeline, some thread selection control logic, and a small cache.

functionality is little more than a set of registers (manipulated as bitfields) to indicate what processors have requested a particular lock, and which processor currently holds it. In our evaluation infrastructure, all communication with the global lock manager is performed over the communication bus by use of special ISA extensions on the PEs.

5.2.2 Speech Processing Element Details

The execution resources of each PE (depicted in Figure 5.5) are very simplistic. Due to the availability of extensive thread level concurrency, our model is able to eliminate many of the power consuming resources utilized to improve performance on modern microprocessors. For example, the processing elements have no ability to expose ILP, the scheduling and issue logic of which accounts for more than 15% of chip power on modern processors [15]. Furthermore, the inherent complexity of stochastic search programs provides strong impetus to use integer representations of data and avoid long latency computations. For example, the search tree evaluation phase of Sphinx issues only integer arithmetic and logical operations (along with loads and stores) excluding even multiplication and division operations. As a result, these pipelines consist of little more than an ALU and sufficient logic to handle control, memory, and basic mathematical operations. The basic execution model is of a standard 5-stage, in-order pipeline with branch-not-taken prediction and branch resolution in the execute stage.

The hardware thread contexts consist of a set of integer and control registers (for the

ARM instruction set employed by the XScale processor, approximately seventeen 32-bit registers), and the selection logic required to identify a ready context from which to execute. Each pipeline also contains a bank of "locally global" registers (accessed using ISA extensions) used to hold parameters and data that are global to the PE and must be accessible by any context, and a small "work queue" used to buffer a few work elements for future assignment to a context. A small cache (2k, 4way in the baseline case) is included to capture basic spatial locality and improve integration with the memory system. No cache coherence protocols are employed between PE caches as data consistency is maintained at the software level through partitioning and locking. Finally, the operations previously performed by the atomic operation unit (predominantly "add" and "max") are now performed on each processing element as ISA extensions accessing the "locally global" registers. The main processor, through ISA extensions, can perform global scatter/gather operations to access this data.

5.3 Programming Model

As we adopt a more partitioned approach in the architectural model, so too do we follow a partitioned approach to the programming model. While previous architectural revisions treated all processors as equal and all thread spawn attempts as equal, the performance results of those evaluations demonstrated that this was a naive approach. This new model employs many techniques similar to our method of exposing concurrency in the first revision, but wraps these techniques in a higher level global behavior that appears somewhat more like the standard shared memory multi–processor style utilized in our first evaluation.

This two stage approach to concurrency is most obvious in the interaction between the main processor and the speech co–processor. Rather than arbitrary thread spawn instructions being assigned to available thread contexts, our new programming model utilizes a much more explicit fork/join model from the viewpoint of the main processor. In order to start a section of parallel execution, the main processor executes a special "ISSUE" instruction, which contains a single program counter value. This instruction and PC value are

broadcast along the communication network, and cause all processing elements to simultaneously begin execution at the specified address. The main processor may then proceed to perform other work. Within some reasonable amount of time, the main processor should issue a "EPOCH" instruction. This has the effect of stalling execution until all active threads on the speech co–processor have terminated, and provides the same effect as a synchronization barrier. In an actual system, the main processor may perform a context switch and perform other work during the time spent waiting for an epoch to complete, allowing our infrastructure to blend seamlessly into a multi-tasking environment. Presumably, once an epoch has completed, the main processor will begin the next phase of computation by performing any necessary serial operations and then executing another "ISSUE" instruction.

While this provides a very straightforward programming model, in the interests of efficiency we add one further programmatic aspect. Recall from the architectural description that we include a second thread context on the main processor itself. This contexts is accessed through a second special set of instructions, and is used for essentially scheduling work to be performed during epoch waiting periods. This is conceptually similar to a context switch, but with far lower overhead as a full register spill / fill need not be performed. We utilize this low overhead context switch to our advantage by "scheduling" tasks within the speech recognition work itself that require access to facilities only available to the main processor (such as floating point computations). By configuring the context selection process on the main processor to switch back to the original execution thread whenever an epoch occurs, we are able to perform substantial amounts of work in the background on the main processor without holding up the issuance of new parallel sections of work for the speech coprocessor. Thus, at a conceptual level, the behavior of the main processor during a single iteration of speech recognition now amounts to scheduling work in the background thread, issuing parallel sections and awaiting epoch events, then finally issuing a special instruction to ensure that all needed work on the background thread has completed before moving on to the next iteration.

The programming model for the individual processing element is much more akin to the

general programming model used in revision one of this architecture. Due to the statically partitioned nature of the data, however, a more structured partitioning of the workload becomes possible. Rather than maintaining single global lists of active channels and other similar structures, each processing element is now configured to maintain its own. This is possible because the partitioning guarantees that a given data element will be handled by a given processing element. This almost completely eliminates the need for centralized global data locking. Continued use of the "exclusion ID" model with thread spawn instructions on processing elements allows for distributed fine–grain locking that ensures global serial access to individual data elements. Furthermore, the task of spawning new worker threads is distributed to each processing element, resolving the workload distribution problem and data passing problem seen in the previous architecture.

To place this into better perspective, consider the operation of a processing element through a single parallel section. Work begins when the "ISSUE" command is received from the main processor. At this point, the processing element loads the program counter passed with the ISSUE instruction into its first thread context and begins executing. This initial thread may make use of partition specific data stored in the "locally global" registers discussed previously to determine the memory location of active lists or other such information for this partition. It then begins spawning threads as necessary. In this case, the thread spawn event is entirely local. The new exclusion ID is compared against the IDs of other active contexts, and the appropriate action is taken. In the event of an ID conflict, we take advantage of the newly added work queue facilities, and store the spawn request away if possible, allowing the executing thread to continue. This, once again, becomes a practical optimization because all decisions and data passing may be done locally. Once all threads that have been spawned finish executing, the processing element generates a signal which, once all other processing elements have also finished, generates a "epoch" interrupt on the main processor. It should be noted that this completion signal could be as simple as a cascade chain of AND gates and does not require sophisticated communication on the interconnect. By this approach, we have achieved nearly complete distribution of the work

involved.

5.4 Application of SPHINX to Architecture

As we discussed the application of programming models to Sphinx for our previous architectures, this section once again steps through the phases of Sphinx and discusses how they are implemented in this architectural model. The most significant initial difference in this regard is the static partitioning of the search tree. We create these partitions by performing profile runs of Sphinx with training input, and annotating usage characteristics to the search tree nodes. The results of this profile are written to a data file, and the hMetis graph partitioner [42] is used to generate equal weight partitions, weighing the nodes based on usage. We find that, due to the nature of the search tree, all of these partitions can be generated without creating any edge cuts and still maintaining a minimal imbalance. Specific partitioning details for our 11440 word evaluation vocabulary are presented in Appendix D. We assume that in an actual architecture, this partitioning information would be used to place data nodes in specific memory locations, allowing easy identification of which node belongs to which partition.

This partitioning of search tree data nodes is most relevant because it partitions the primary read–write data accessed during search phase processing. It leads directly to individualized channel lists for each partition, further reducing the need for global communication. It is also theoretically possible to replicate all of the remaining immutable data as necessary, and using this tree partitioning to partition other aspects of the search as well. This is most relevant as it relates to the Gaussian Scoring phase of computation. Recall that the set of currently active senones which must be scored by this phase is directly related to the set of currently active search tree nodes. One could conceptually imagine that each processing element may now perform scoring for the subset of active senones related to the set of active search tree nodes within its partition. We quickly discover, however, that replicated effort leads to massive overheads from this approach, as many search tree nodes in many different partitions may point to the same senone information. Thus, as we discuss next, the senone

scoring operations are simply distributed by number to available processing elements.

With this general understanding of the static data partitioning aspect of this programming model, we proceed with our presentation of Sphinx parallelization. We once again begin with the Gaussian Scoring phase. Due to the new features and constraints of this architecture, we make a fairly substantial departure from the parallelization philosophy utilized thus far. Specifically, PDF selection is performed concurrently with the actual senone scoring (and linguistic model evaluation) phase by use of the background context available on the main processor. This is possible because PDF selection does not depend on the feedback path used to reduce workload during senone scoring. Thus, it is possible to frame shift this segment of computation, and perform PDF selection one frame (or iteration) ahead of the remainder of the search algorithm without incurring any additional overhead. Recall also that this background main processor context can be assigned work, and interrupted whenever a parallel execution phase on the speech coprocessor terminates. Thus, we schedule PDF selection for a subsequent search frame, and ensure that it completes by the time that search frame is actually evaluated by the core search engine. Since PDF selection is only a small contributor to overall workload, but requires floating point facilities no longer available on the speech processing elements, we find that this concurrent execution almost always hides the selection computation behind the latency of model evaluation without the added physical floating point resources required in previous implementations.

The senone scoring segment, as discussed previously, could not be partitioned along the same lines as the search tree. The incurred overhead of replicated computation was substantial. Instead, we employ a very straightforward approach, partitioning senones by contiguous group and assigning them to processing elements during initialization. While this may not result in the most balanced partitions over the course of the entire application, the impossibility of maintaining per–iteration balance makes this a non–issue, and the contiguity of memory references through such an allocation is helpful. Since we maintained a shared memory model view in this architecture, a cache flush at the end of senone scoring ensures that correct score information is available to all partitions, regardless of where the senone

scores were computed.

Given the static partition (and the partitioned channel lists that are naturally derived from it), high efficiency parallelization of the linguistic model became a very straightforward problem. The essence of our concurrency technique is basically the same as the previous revision. In this architecture, however, each processing element traverses its own active channel lists, and spawns off new threads. As discussed previously, all scheduling decisions are local, and overall global communication is minimized. Furthermore, our partitioning scheme resulted in an intrinsic by–word partition (due to the natural lack of inter–word edges in the static search tree resulting from dynamic word channel expansion). Thus, word channel evaluation, expansion, and pruning steps could also be performed without the need for substantial global communication. While this architecture supports transmission of threads between partitions (for example, partition one attempts to spawn a thread that belongs on partition two), the need for this communications is also very limited.

One aspect of the programming effort that has been repeatedly implied but never discussed in detail is the need to acquire information on the current partition (such as the start of the current partition's active list). Clearly some degree of initialization is required to set up the necessary information for each processor, and some means of acquiring this initialization information is required on each processing element. This is, predominantly, where the "locally global" processing element registers come into play. The main processor has the ability to set individual registers on individual contexts through both specific and scatter/gather type operations through a set of ISA extensions. Thus, during initialization, the main processor distributes and sets many of these locally global registers, making the necessary per–partition information immediately available to each PE.

On the other side of processing, once an iteration of speech (or relevant phase of an iteration) is completed, it is often necessary to generate best scores or other max/accumulate operations necessary later in processing (for example, the best score from evaluation is used to set the pruning threshold). These operations, performed by the atomic op unit in the previous architecture, are now performed in a distributed fashion on each PE. As

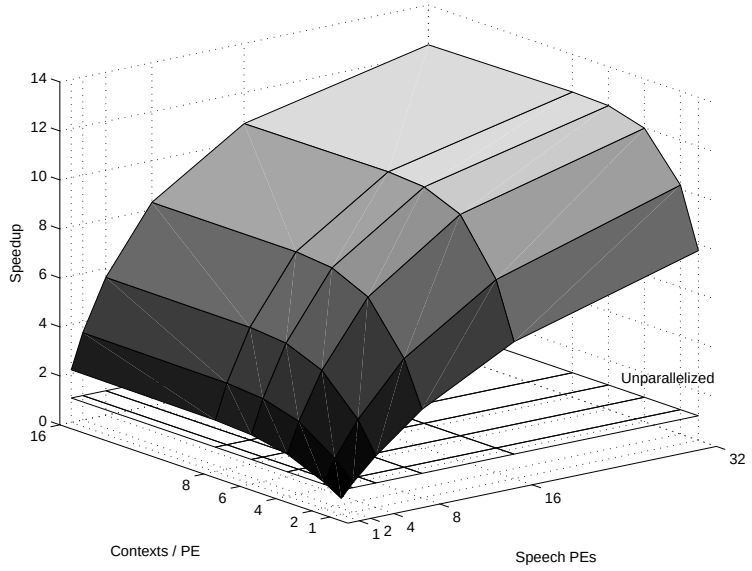
part of the scatter/gather facilities of the main processor, however, are a "gather_max" and "gather_accumulate" operation, which automatically pass data between processors and utilize processing element execution resources (which are idle when these operations are performed) to quickly provide the necessary global result. Thus, collection of a true global value is postponed until the last possible moment, allowing the best use of concurrency for the remainder of the execution time.

5.5 Performance Analysis

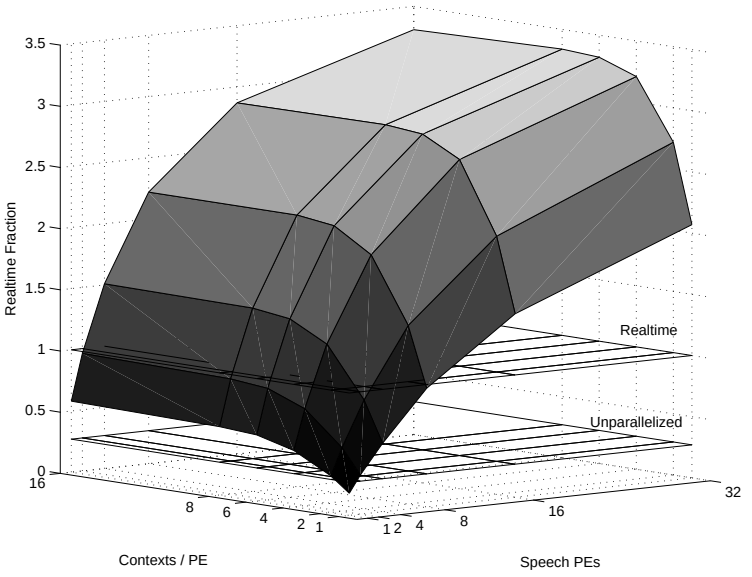
This section explores performance as well as power tradeoffs for our final architecture model to verify that it meets the concurrency opportunity available in the speech recognition domain. Once the effectiveness of this model is established, later chapters will perform more detailed architectural and bottleneck analysis.

As with previous evaluations, the results shown in this section are based on a multi–processor implementation of SimpleScalar/ARM which models the architecture described. We assume a 400MHz XScale main processor with the necessary floating point extensions, and assume that the speech co–processor and system interconnect also run at 400 MHz. For this architectural evaluation, we assume once again a fixed latency for all requests out to the memory system, setting the delay to 100 processor cycles. A more detailed breakdown of our simulation infrastructure, particularly as used in later true memory system analysis, is presented in Appendix A.

While later sections will evaluate the effect of constrained resources in some detail, for this initial evaluation, we assume a simple interconnect that models a 16 bit bus, and no internal delay in global resources such as the dynamic scheduler and global lock manager. We further assume no delay to spawn a thread onto a new context on the current processor (thus incurring only the standard programmatic function call overhead). Each processing element is assumed to have a 2k-4way L1 data cache, with instructions stored in a local ROM. The latency of a cache hit is a single cycle, while the latency of a cache miss is the fixed memory access latency.



(a) Relative to Unparallelized



(b) Relative to Realtime

Figure 5.6: Performance of Final Architecture - These figures show the performance of our final architectural implementation relative to the unparallelized code running on a single XScale processor (a) and relative to a realtime constraint (b).

The raw performance speedup relative to unparallelized code running on a standard XScale processor and relative to realtime constraints are depicted in Figure 5.6. As these graphics clearly demonstrate, this new distributed concurrency approach finally captures the performance improvements achievable through concurrency. Based on the programming model discussed above, we determine that around 92% of the search phase computation is executed on the speech co–processor. Of the remaining 8%, around half is represented by the PDF selection phase and other components that can be executed concurrently with co–processor operation. Given these figures, the performance speedup for increased numbers of processors tracks rather well with ideal expectations, and much of the lost performance is regained through the addition of a few thread contexts.

If we examine these numbers in greater detail, further nuances become apparent. First, there is rather substantial overhead to our concurrent approach. The single–processor/single–context implementation sees a 40% slowdown relative to the unparallelized code. This despite the 4% PDF selection phase, which still runs concurrently with search phase code on the main processor. The primary source of this overhead is found to be the result of exposing concurrency. Many operations that occurred within "for" loops in the original code are replaced with thread spawns in the parallelized code. As these spawn instructions model function calls, the lack of any parallel resources for them to execute on means that the system incurs the full programmatic function call overhead, including register fills and spills, but in unable to reclaim the time spent on these extra operations.

Second, as we analyze sources of latency and inefficiency in the system, we find that the average number of "active" cycles for any given processor pipeline (cycles of actual execution, as opposed to stalls) decreases at an almost precisely ideal rate as the number of physical pipelines is increased. This suggests that any fall–off from ideal performance (beyond the initial coding overhead of parallelization) is due not to inefficiencies in the parallelization effort, but rather due to imbalances in concurrency management. Put another way, if the number of active cycles for all processors decreased along an ideal curve, the only potential source of added latency is a mis–alignment of those active cycles such that one

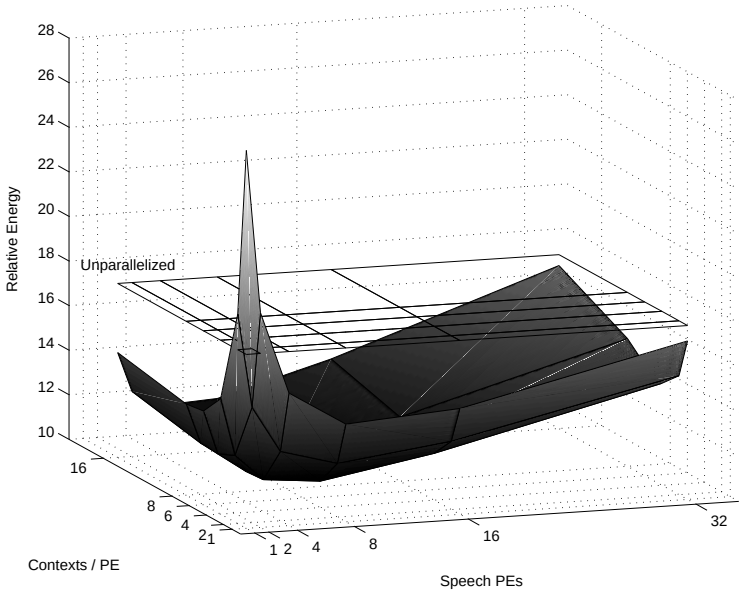


Figure 5.7: Energy of Final Architecture Relative to Unparallelized - This graph depicts the relative energy of various processor / context configurations for our final architecture. This data depicts once again the "reduced energy by reduced execution time" effect seen in our revision one energy estimates.

or more processor are inactive while others are active. Further analysis reveals that much of this imbalance occurs at the end of parallel sections (during synchronize operations). As alluded to previously, the profiled static partitioning does an effective job of distributing work (leading to very similar active cycle counts), but is unable to affect these imbalances on a per–iteration bases. If, in every iteration, at least one processor in required to evaluate two more channels than any other due to the current state of the search, the overall imbalance at the end of the speech input can become quite substantial. This is the primary effect we see detracting from ideal performance in these initial architectural studies.

We finally note that the realtime graph depicts substantial extra performance over the timing constraint. It is important to note that this performance potential can be converted into increased accuracy through relaxed search tree pruning, or increased power savings through voltage and frequency scaling techniques. As such, this performance overhead is not simply unnecessary excess, and must be evaluated in more detail, and in relation to other system–wide goals and constraints.

The relative energy estimates of various processor / context–per configurations of this

final architecture are depicted in Figure 5.7. This graphic once again depicts the workload energy reducing effects of improved performance, similar to the characteristics seen in our revision one results. This effect is greatly enhanced by the dramatic performance improvements achieved in this architecture model. Note, however, that despite continued performance improvements, energy benefits still trail off at 4–6 contexts and 4–8 processors. The improved performance, once again, is insufficient to make up for the added power demand for such large systems. Unlike the previous architectural revision where this fall–off was primarily due to lack of performance improvement, however, in this architecture it is a combination of performance factors and size factors. For example, each added pipeline in this model incurs a much larger energy cost due to the presence of a per–processing–element cache.

5.6 Related Solutions

Hardware based approaches to the problems inherent in the speech recognition domain are not new, and a number of parallel architectures and algorithms have been investigated in other works. The primary difference between these works and this research is that our focus is not only on performance, but also on constraining power to allow speech recognition in a very confined domain, dramatically changing the characteristics of the problem. Hon explored several approaches to hardware speech recognition, including AT&T's Graph Search and ASPEN Tree Machines and CMU's PLUS architecture [34]. These machines all consider the same basic problem we are looking at here, with a focus on pure recognition performance on large, multiprocessor systems. These are also on older architectures and thus consider far smaller vocabularies.

Anantharaman and Bisiani develop a custom hardware accelerator for speech recognition [11]. Their work is primarily focused on transformations from the algorithm leading to a custom hardware implementation and thus acquires a number of algorithm specific characteristics. Our architecture, in contrast, exploits only the most general aspects of the speech recognition domain, achieving substantial performance improvements through a architec-

tural model that is applicable to a number of problems within the stochastic search domain. Chatterjee and Agrawal consider connected speech recognition on the MARS pipelined processor [18]. Their primary focus in this work is implementation and performance on their existing architecture and do not consider architectural arrangements to maximize recognition performance or power considerations.

Ravishankar [76] presents a parallel implementation of the SPHINX beam search heuristic. Many of the techniques described in his work are utilized in our parallel implementation, with modification for our programming model. He is able to demonstrate 3x performance improvements on a 4-processor SMP system, tracking well with our experimental results in low power, embedded environment. Our work also demonstrates the effectiveness of added SMT support it improving performance.

Mathew et al present a low power accelerator for the Gaussian scoring phase of SPHINX–3 [65]. They find the memory bandwidth requirements of their infrastructure exceed 800 MB/sec, far greater than the available bandwidth on even high end hand–held platforms. They manage memory constraints by altering the reference patterns of the Gaussian phase to eliminate feedback from the linguistic model evaluation, resulting in consistent, stream style reference patterns. They then take advantage of the consistent computation performed in this phase and pack evaluations of multiple input frames into single passes of the knowledge base data, as well as reduce the accuracy of the floating point computation to reduce bandwidth demands. In our architectural model, breaking the feedback from linguistic model evaluation lead to prohibitively large increases in workload during the Gaussian phase due to the less ASIC style approach employed. The HMM phase of speech recognition also does not perform straightforward, well understood mathematical computations (or rather, the computations performed are more likely to vary with recognizer generation and algorithm) making an ASIC approach much less feasible.

With regards to SMT and SMP, recent work has also shown potential benefits of hybrid system similar to ours. Sasanka et al consider chip multiprocessors and SMT facilities on out of order processors for multimedia benchmarks and find that CMP processors provide

the best energy efficiency, and hybrid systems show the ability to accommodate both high performance and high energy efficiency [84]. Kaxiras et al observe similar trends on mobile phone DSP workloads [43]. While the search phase of speech recognition that we address is not DSP-like in nature, the basic principles of power saving through exploitation of concurrency remains the same. Numerous works (for example: [53, 92]) have also shown that low–overhead multithreading can be employed to achieve near peak computation rate in the presence of low ILP and high cache miss rates, often at relatively modest hardware cost [92].

The programming model itself is similar in concept to a number of prior approaches. It is based on a pthreads style approach to describing parallelism, but has evolved into a scheme more in line with Futures [33], Promises [62] or Active Objects [54]. Unlike those works, however, our system makes no effort to track threads once they have been issued, assuming all data is returned through shared memory. This reduces system and algorithmic complexity significantly, and is feasible because of the nature of concurrency in this domain.

By contrast, our approach to achieving mutual exclusion (the exclusion ID field associated with a thread spawn) is somewhat different from standard approaches to achieving exclusion and synchronization on general purpose platforms. Solutions such as Speculative Synchronization [64] (in which the system runs speculatively past lock/barrier conditions until an explicit error is discovered), Optimistic Synchronization [78] (in which a linked load of a lock value and a conditional store are used to optimistically assume lock acquisition until the store attempt discovers that the data has changed), and Adaptive Replication [79] (in which data under a lock is replicated, and independently modified versions are later merged as necessary), all attempt to reduce the observed overhead of synchronization and mutual exclusion operations. Our approach is philosophically identical to the Parallel Dispatch Queue proposed by Falsafi and Wood [28], in which special synchronization key values are used to describe the resource demands of messages in fine–grain parallel software communication protocols. These keys, used identically to our exclusion IDs, allow their system to synchronize messages in the queue prior to dispatch, allowing execution of protocol handlers

in parallel without race conditions. The nature of resource (data node) demand by individual threads in our speech recognition task fit very well this this model of mutual exclusion. Combined with static partitioning of nodes to distribute exclusion ID's across processing elements, this allows us to avoid long synchronization latencies rather than attempt to mask them.

CHAPTER 6

Architectural Evaluation

The previous chapter developed our final architectural model through a series of recursive design steps. We further demonstrated that, given a somewhat idealized environment, this architecture and programming model are capable of demonstrating near–ideal performance improvements for our recognition system. This chapter will now move away from developing a working architectural model, and begin instead a more thorough analysis of the final model. We will explore a number of programmatic and architectural variations, and reach conclusions regarding the actual constraints these architectural metrics place on design.

The base system configuration used in this chapter is the same as that developed at the end of the previous chapter. That is, we assume a base XScale system processor running at 400MHz, and a separate speech co–processor for parallel code. We begin by assuming an ideal interconnect between these devices, and a fixed 100 cycle memory latency on all requests to main memory from either the main processor or speech co–processor. Furthermore, each processing element (on the co–processor) is assumed to have a 2k, 4–way set associative cache, and a small work queue to buffer jobs. Each thread context represents a complete duplication of the general purpose registers available to the arm instruction set (around 17 registers) and each processing element contains one set of 32 "locally global" registers. It should be noted that a real implementation of this system, taking advantage of the more static nature of the memory space, could reduce the number of "locally global" registers required substantially by hard–coding certain memory segments which are currently

dynamically allocated.

As most of the architectural variants considered in this chapter involve small hardware or programmatic changes, we will predominantly focus on performance. Later chapters will continue our power analysis in more detail as we consider architectural variants that have a more significant impact on overall power / energy consumption.

6.1 Register Pressure

One of the major aspects of this design is the inclusion of multiple hardware thread context resources on each processing element, and a second thread context on the main system processor. Taken in sum, the physical space needed for these hardware contexts, and the extra routing and switching logic needed to access them can add significantly to size / power dissipation. Thus, we wish to explore the actual register pressure of this application space to determine if a reduction in the number of physical resources in each hardware context is practical.

We perform this analysis by re–compiling Sphinx utilizing the –ffixed flag to the GCC compiler to explicitly remove registers from the compiler allocation pool. It should be noted that the GCC register allocation scheme is generally considered poor in its efficiency. The results of this analysis can thus be considered a fairly worst case estimate of actual application register pressure.

It is important to note that the ARM instruction set exposes a number of operational registers in the general purpose register space. For example, the program counter is actually register 15. Thus, in reality, there are closer to 12–13 usable general purpose registers at any given point of program execution. We find that, when exploring search phase code executed by the speech coprocessor, elimination of five registers leads to approximately a 5% slowdown in overall performance. Elimination of a greater number of registers prevented successful compilation of the program. While a 5% slowdown does not appear to be significant relative to the performance effects we see in other areas, it does suggest that removing a large number of registers (say, cutting the size of each hardware context in half)

will not be possible. The elimination of these five registers may also have a negative impact on the ability to run other programs on this architecture, leading us to maintain a standard register configuration for the hardware thread contexts in the remainder of this work.

A more interesting result is found when considering the effect of register elimination for search phase main processor code relative to performance. Restricting the compiler to 2 general purpose registers and 2 floating point registers (out of fifteen and seven respectively) results in a slowdown of less than 0.5%. This is certainly in part related to the relatively small amount of work the main processor has to do during a search iteration relative to the speech co–processor, but it does suggest the very intriguing possibility of eliminating the "second hardware context" entirely, and instead utilizing a logical partition of the existing register file to the same effect. This would require substantially less hardware, and substantially fewer changes to the actual processor design. Non–search phase portions of code that require a greater number of registers could simply remove the logical partition and proceed as they normally would. An important motivation to the feasibility of this approach comes from the fact that we were able to restrict the number of registers to well below half, leaving enough room for duplication of state registers such as the program counter as well.

6.2 General Latency Tolerance

One of the primary motivators in our architectural designs was the ability to tolerate latencies in the overall system. Indeed, maintaining high processor utilization despite long system latencies is the main reason for developing a hardware multi–threaded processor model. Thus, before we begin a more specific analysis of individual potential contributors to latency and other unmaskable latency sources, we wish to explore the general ability of the architecture to tolerate delays. The most obvious and easily manipulated source of latency in this system is the time required to make off–chip memory accesses into the DRAM. We thus evaluate the ability of our architecture to tolerate latency by considering the relative performance of a system with a fixed 50 cycle memory latency versus our base system with a 100 cycle memory latency. It should be noted that a 50 cycle memory latency is much

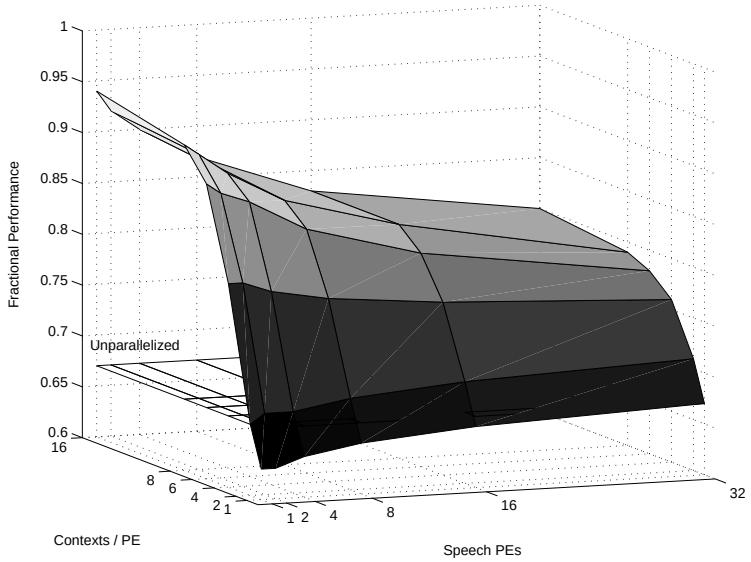


Figure 6.1: Fractional Relative Performance of 100 cycle memory vs. 50 cycle - This figure depicts the fractional performance of a given processor / contexts–per configuration with a 100 cycle memory latency as opposed to a 50 cycle memory latency. A value of 1 on this graph would mean the 100 cycle experiment performed as well as the 50 cycle experiment, while a value of 0.5 would mean the 100 cycle experiment performed half as well (or took twice as long). This graph clearly demonstrates the latency tolerating effects of added hardware contexts and multi–threaded execution.

closer to the actual latency of a 100MHz SDRAM system with low resource contention.

The relative performance of a 100 cycle memory latency with respect to a 50 cycle memory latency is depicted in Figure 6.1. As this figure clearly shows, the addition and utilization of hardware thread contexts can act to substantially mitigate the performance loss of increased system delay. While the unparallelized code performs at around 68% of its 50 cycle delay performance when the memory delay is doubled to 100 cycles, a single speech processor with 8 contexts runs at almost 95% of its 50 cycle performance. We further see that this effect is directly related to the addition of thread contexts. Indeed, the relative performance benefits decrease with added processors, as workload imbalance and other effects (such as limits on exposed parallelism for a more finely distributed workload) become factors in limiting performance. It is important to note that this graph depicts relative performance for same–configuration experiments. A 16 processor implementation still performs substantially faster than a 2 processor implementation on either the 50 cycle or 100 cycle delay. This graph simply shows that added thread contexts do not reclaim as

much of the performance loss with 16 processors as they do with 2 processors.

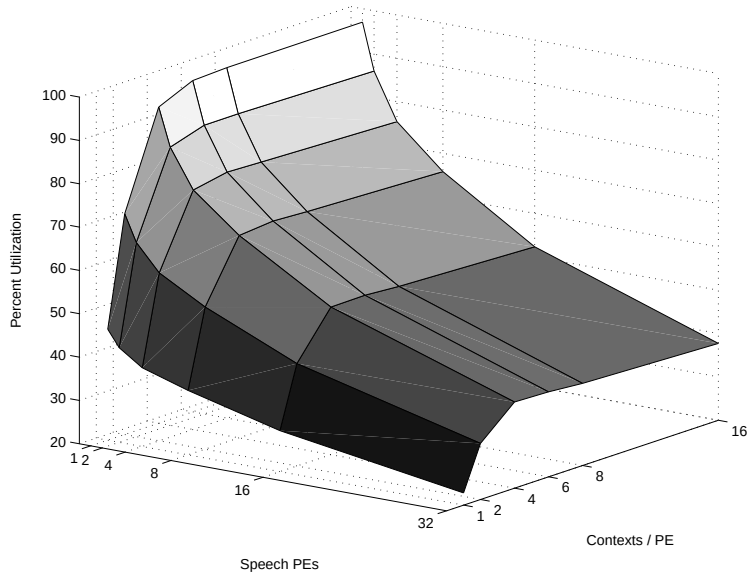
We investigate one further aspect of performance effects as they relate to system latency by considering the average processor utilization of systems with 50 cycle and 100 cycle latencies in Figure 6.2. These graphs both demonstrate a fairly dramatic rise is processor utilization as we begin adding hardware contexts. The bottom figure (depicting 100 cycle memory latency) also shows a slightly extended ramp up before leveling off, accounting for much of the latency tolerance seen in the previous relative performance graphic. These graphs also demonstrate how added physical pipelines, while providing more raw processing resources and thus improving performance through concurrency, also lead to lower average processor utilization due to workload distribution and imbalance effects.

In general these results demonstrate the effectiveness of this architecture at tolerating arbitrary fixed system latencies. They do not characterize the behavior of the architecture under varied delays as would be seen with bottlenecked resources. For example, a memory system bottleneck would lead to varied delays in the perceived latency of a memory request, increasing with the number of other outstanding requests awaiting servicing. We will investigate such throughput constraints in greater detail in later chapters. These results do suggest, however, that non–throughput limited components that may simply be slow due to circuit or power constraints may be well tolerated in our environment.

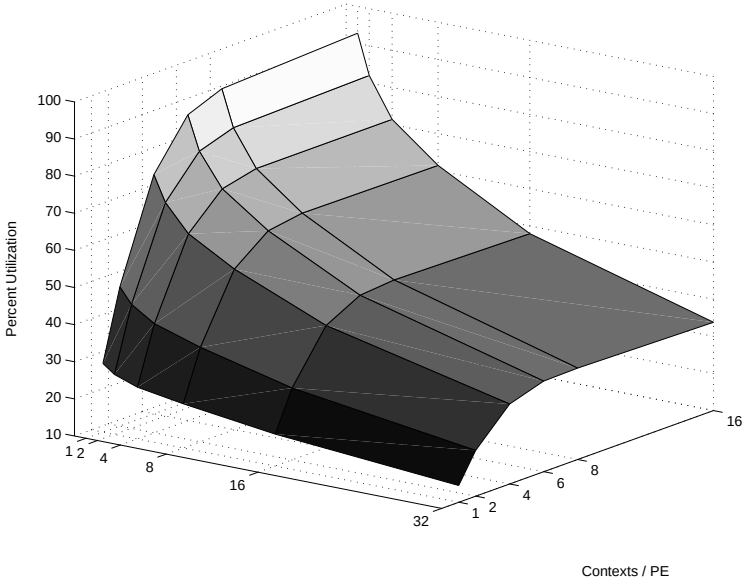
6.3 Thread Spawn Delay

Akin to latency tolerance in the motivation for our design is the ability to manage large amounts of concurrency with low visible overhead. Our programming model essentially assumes large numbers of threads can be spawned with minimal impact on performance. If the act of spawning threads constitutes a major performance bottleneck, then our approach to exposing concurrency must be re–examined.

In order to evaluate the performance effects of thread spawning latency, we consider the impact of applying an arbitrary latency to all thread spawn operations. Relative to our base idealized design, which assumes no architectural latency for spawning new threads



(a) 50 Cycle Latency



(b) 100 Cycle Latency

Figure 6.2: Processor Utilization with 50 cycle and 100 cycle Memory Latencies - These figures depict average processor utilization for experiments with a fixed 50 cycle memory latency (a) and a fixed 100 cycle memory latency (b). This further supports and confirms the effectiveness of our multi–threaded approach, demonstrating that these extra thread contexts are quite effective at improving pipeline utilization and reclaiming utilization lost to added system latency.

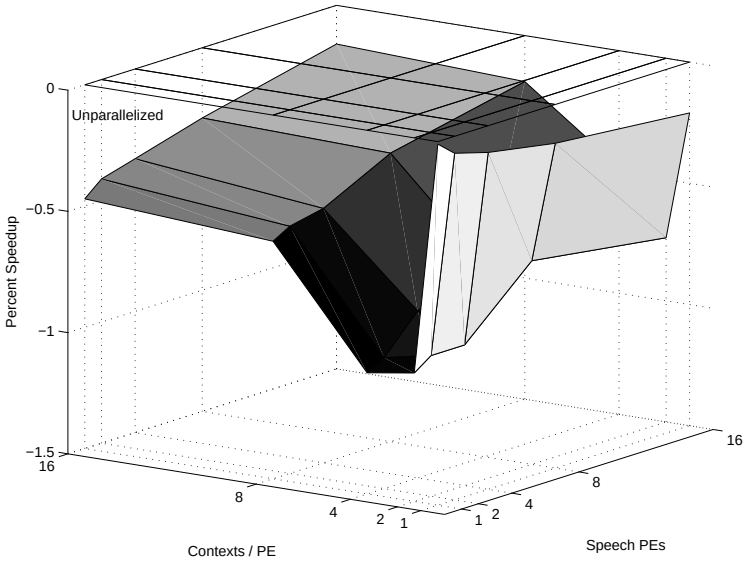


Figure 6.3: Percentage Slowdown of 20–cycle Local Thread Spawn Delay - This figure depicts the percent slowdown incurred by applying an architectural thread spawn delay of 20 cycles plus 1 cycle for every two registers to be copied. We believe such a delay far exceeds any reasonable delay that might be incurred for local thread spawn operations (data copies and such). Clearly this does not contribute a substantially to performance effects, suggesting that this operation need not require a sophisticated high performance implementation.

(only programmatic effects of the function call interface), Figure 6.3 depicts the effect of utilizing a 20 cycle spawn latency. In order to better model a real system, we actually go beyond a straight 20 cycles, and also assume one cycle for every two registers that must be copied between contexts to set up the new thread. This should be well more than any realistic latency for such a local operation. As the figure shows, the performance effects of this were insignificant, resulting in less than a 1% slowdown in all cases. The actual performance variations were too narrow to identify the source of the dip for 2–4 contexts, but it is important to note that small variations may be expected in architectural modifications that essentially alter a large, dynamic execution schedule.

6.4 Communication Network Latency

A common source of latency and contention in multi–processing environments often occurs at the communication network. Most parallel system implementations go to signif-

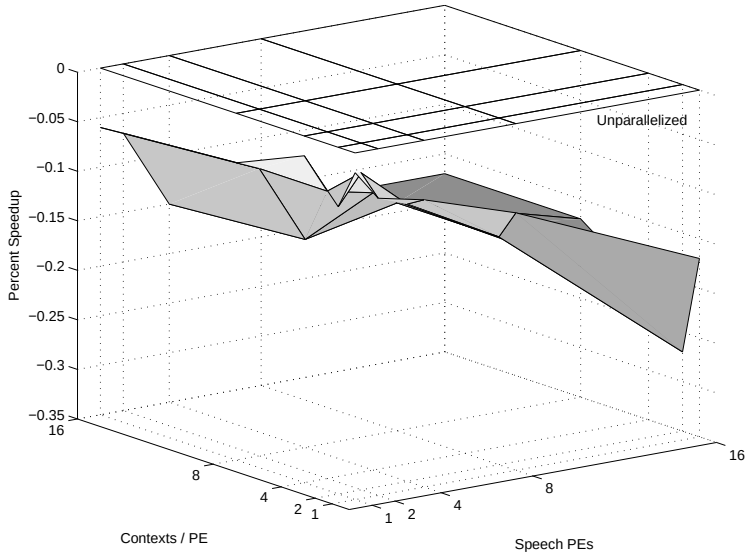


Figure 6.4: Percentage Slowdown of Highly Constrained Communication Network - This figure depicts the percent slowdown incurred by modeling the command interconnect as a 8–bit bus with a 10 cycle protocol overhead (beyond data transfer time) relative to our idealized model which assumes a 16 bit bus and no protocol overhead. Communication on this network includes ISSUE and EPOCH commands, job migration, and all inter-processor communication, including communication with the global lock manager. We assume information regarding availability of migratable jobs or free contexts is passed to the dynamic load balancer through a separate channel, as this information must be available on a per–cycle basis. This data suggests that, due to the relatively small inter–processor communication in our system, the command network can be very slow, and designed to minimize power consumption.

icant efforts to minimize inter–processor communication to avoid just this problem. The static partitioning utilized in our design serves this purpose by providing an initial workload distribution, reducing communication needs to load balancing instead of concurrency exploitation. Recall also that contention for communication resources was a major bottleneck in our revision one architecture.

Given these considerations, we wish to determine if the communication network remains a bottleneck in this current model. The demand on such a system will not only have a significant effect on performance, but may also impact power, as high performance inter–connects can add substantially to run–time power dissipation. We perform this analysis by modeling the system interconnect as a bus, and assuming various bus widths and protocol overheads to account for complexity of communication.

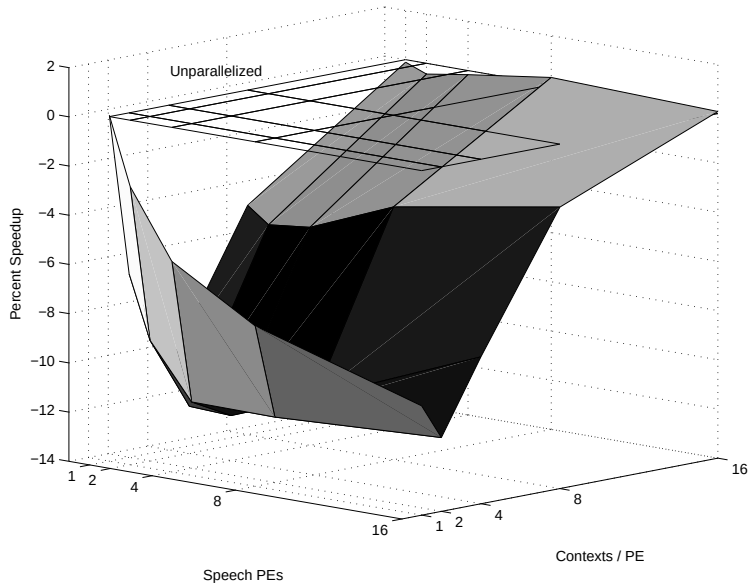
As shown in Figure 6.4, even constraining the communication network to a 8–bit bus with a 10 cycle protocol overhead on each communication (relative to a 16 bit bus with no overhead) has minimal impact on overall performance. We see the beginnings of a bottleneck on the largest configurations, but find that the performance loss in general is less than 2%. This fortunately suggests that the command interconnect can be designed in a power conserving fashion, with only minimal consideration for performance. It also suggests a safety margin under situations of increased job migration or odd search parameters leading to increased communication.

6.5 Work Queue

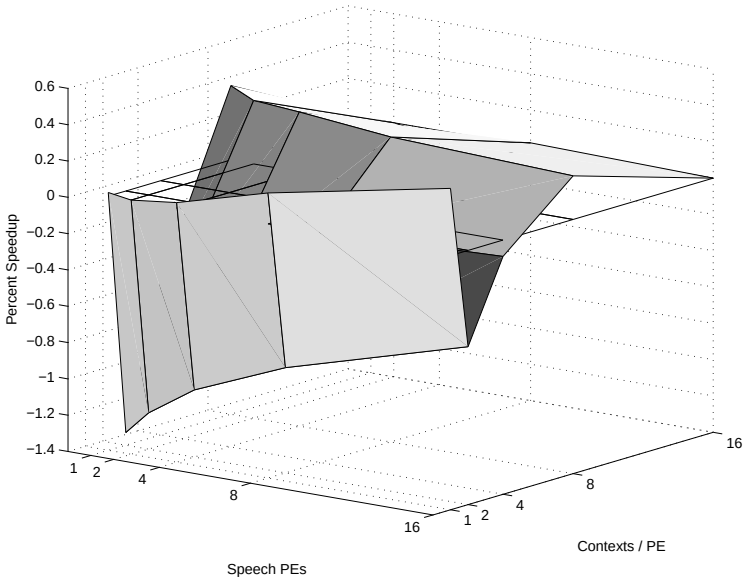
The "work queue" constitutes a small bank of register space on each speech PE which is used to buffer work elements for future assignment to thread contexts. As our parallelization algorithm converts a thread spawn into a function call when resources are unavailable, it becomes possible for a previously spawned thread context to complete its work unit but remain empty because the context running the section of code that issues spawn instructions is busy operating on a work unit of its own. As only one context may run at a time, a considerable number of cycles may pass before the PE returns to the thread spawning code. The presence of a small work queue allows the newly free context to accept a work unit from this small buffer, improving the opportunity for overall processor utilization.

We evaluate the usefulness of this work queue by varying its size from a baseline of 5 entries. Figure 6.5(a) depicts the relative performance of eliminating the work queue for all locally spawned jobs. We see that this leads to a 10% slowdown on average for smaller numbers of contexts. As the number of contexts passes the utilization point (approximately 8 in the ideal model), this slowdown disappears as potential work queue usage points are eliminated.

By contrast, Figure 6.5(b) depicts the relative performance of increasing the work queue size to 10 entries. This increase in size has a minimal impact on performance, and we once again see secondary effects in the result data. Overall, this suggests that a small work queue,



(a) No Work Queue



(b) 10–entry work queue

Figure 6.5: Relative Performance of Small and Large Work Queues - This figure depicts the relative performance of a system with no work queue for newly spawned jobs versus one with a 5–entry work queue (a), and of a 10–entry work queue over a 5–entry queue (b). While the addition of a work queue to buffer jobs can lead to notable performance improvements, we see here that a small queue is sufficient to capture the effect.

and the resulting buffer of runnable jobs, can be helpful under certain configurations, but that only a small work queue is needed to reclaim any potential performance loss.

Note that our evaluation did not consider true elimination of the work queue. Work queue entries were still used to hold thread spawn requests that resulted in a UID conflict, and to accept migrated jobs. True elimination of the work queue would exacerbate the slowdown as contexts were forced to stall on conflicting UIDs during a thread spawn, and dynamic load balancing was scaled down.

6.6 Global Locking

The global locking mechanism provides the ability to establish mutual exclusion across all running threads. This is a very slow, easily congested means of performing locking, and as such every effort is made in the software to utilize fine–grain locking mechanisms available on individual processing elements. The facility is provided, however, because certain operations such as memory allocation and other operating system commands require global exclusion.

As access to the global lock manager is inherently constrained by access to the communication infrastructure, we would expect no greater impact on performance due to the speed of this very simple hardware as due to the bandwidth through the communication network. Our baseline results show that both the delay due to cycles of lock manager operation, and delays due to contexts waiting for currently held locks is trivial relative to other performance factors. More precisely, the delay awaiting a global lock can be significant for an individual context on a sufficiently large configuration, but is easily hidden when multiple contexts are available, resulting in an insignificant impact on overall performance across all configurations (there is either low contention, or the resulting latency is hidden).

6.7 Dynamic Load Balancing

Our baseline architecture includes a global job scheduler to perform dynamic load balancing between speech PEs. While the profile based partitioning scheme used to perform

initial work distributions proves very effective in balancing speech PE usage over the duration of the program, it is unable to account for imbalances occurring during individual "SPAWN to EPOCH" segments. The result is that each PE spends approximately the same time actively computing, but the accumulation of small workload imbalances leads to a correspondingly large number of idle cycles waiting for other processors to finish and reach the next epoch. In order to mitigate this effect, the global job scheduler identifies jobs awaiting assignment to a context (waiting on the work queue) of a PE with no free context, and issues commands to migrate such job to the work queue of a PE with free contexts. Migratable jobs are identified by the program, and may not spawn new jobs themselves, or require access to data local to the home processor. This set of constraints minimizes the impact of job migration on the communication network, and allows the job to execute with highest possible throughput on the receiving PE. As these constraints basically match threads representing the inner-most loops of nested search algorithm traversals, they are quite plentiful.

The relative slowdown caused by removal of the load balancing engine is depicted in Figure 6.6. This chart clearly shows that, for a small number of contexts, the effect of load balancing can be quite substantial, particularly for a larger number of partitions. As an increased number of partitions increases the potential for imbalance, this result is quite reasonable. An interesting characteristic, however, is how quickly this effect is dissipated by the addition of thread contexts. As the load balancing engine will only migrate a job from the work queue of a busy processor to a free context on another processor, the addition of thread contexts directly reduces the availability of jobs eligible for migration. In the extreme case, free contexts are always available, and no useful job migration occurs.

To extend this analysis, we consider the aggressiveness of load balancing by placing a constraint on the number of free contexts that must be available on the receiving PE before a job may be migrated to it. The intent of this analysis is to consider the case where a migrated job takes up the last free context on the receiving processor, causing a newly spawned local job to have to wait (and possibly be migrated itself). Our results show

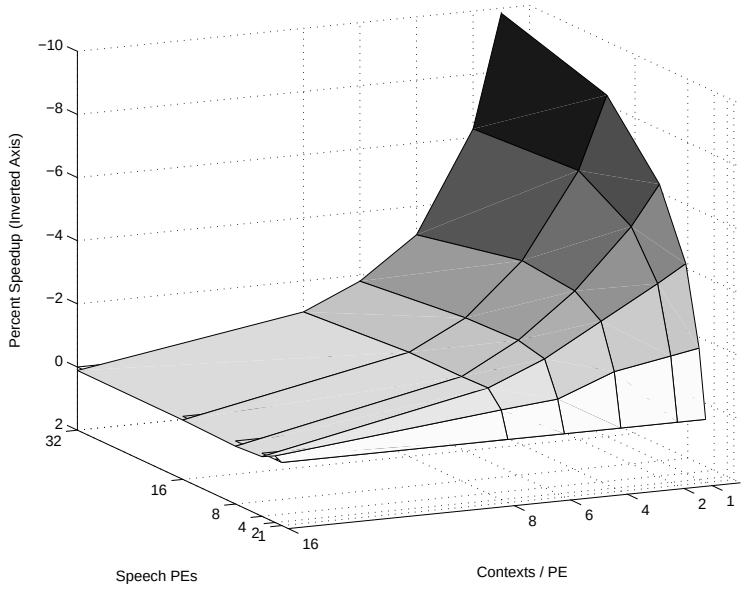


Figure 6.6: Percentage Slowdown of Eliminating Dynamic Load Balancing - This figure depicts the percentage slowdown of eliminating dynamic load balancing relative to the baseline architecture with dynamic load balancing in place. As shown, load balancing has the greatest effect when there are few contexts on each processor. As the number of local contexts directly affects the number of migratable jobs (jobs *on* a context may not be migrated), this is an expected result. We also see a much more substantial effect as the number of processor pipelines (and correspondingly the potential for workload imbalance) grow.

that, such reduced aggression (requiring a larger number of free contexts) only degrades performance, and a dynamic load balancing system that migrates jobs whenever possible achieves the best performance.

6.8 Static Partition Quality

As discussed previously, knowledge base data partitioning serves the dual purposes of aiding in fine grain locking support and providing an initial distribution of workload. The previous section discussed the need for dynamic run–time load balancing to manage per–iteration imbalances which can not be handled by the global static partitioning approach. These per–iteration imbalances, while resulting in relatively equal workload on each processor at the end of search, contribute to a large number of idle cycles and wasted time, particularly with a larger number of partitions. Recall that the static partitions used in

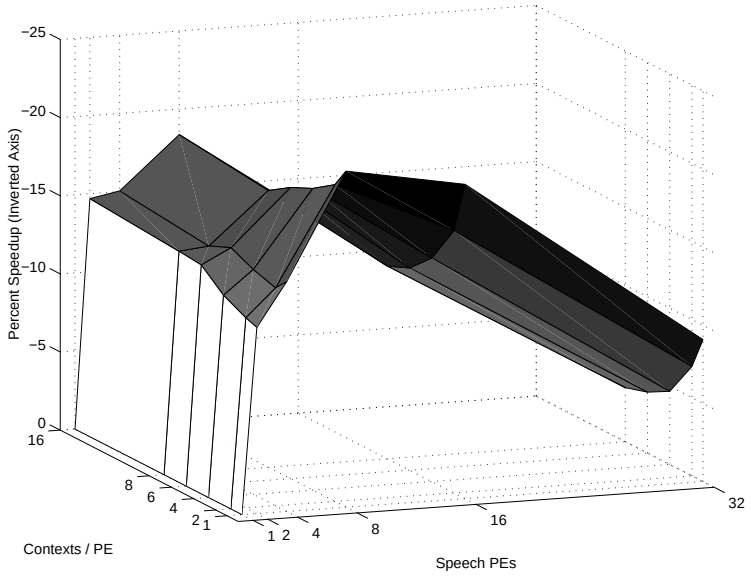


Figure 6.7: Percentage Slowdown of Naive Static Partitioning - This figure depicts the slowdown incurred by utilizing a naive static partitioning scheme as opposed to a more advanced scheme. The naive scheme in this evaluation performs no graph profiling, and partitions search tree nodes based purely on number. The more advanced scheme is the baseline from our other evaluations, and involves a profiling stage to weigh search tree nodes by usage. Partitioning is then performed to equalize weight (as opposed to just number). This figure clearly demonstrates the benefit of a profile based partition. The relative improvement in performance past 8 partitions can be attributed simply to a natural effect of dividing the workload. Since static partitioning is performed offline, there is really no reason not to perform a detailed profile based partitioning step.

these evaluations involved a detailed profiling step to achieve that equal work distribution. If dynamic load balancing is necessary to maximize performance regardless, such a step may not be necessary. We therefore explore whether profile based partitioning is in fact necessary or helpful, or whether any reasonable static workload assignment will suffice.

We consider the effect of two partitioning options. The naive approach distributes partitions based purely on number of search tree nodes, with no consideration of use. The more sophisticated partitioning scheme is the baseline used throughout this paper, weighing each search tree node by access frequency during a profiling run, and partitioning to equalize weight. The results of this evaluation are shown in Figure 6.7 and depict a fairly substantial slowdown for the naive partition scheme for smaller numbers of speech PEs (i.e. smaller numbers of partitions). This slowdown is substantially greater than the effect of eliminating load balancing, and demonstrates simply that the relatively slow job migration implemented

by the load balancing infrastructure is not as effective a method of exploiting parallelism and tolerating delays as the fast job assignment within a single speech PE. A secondary effect of the naive partition is increased pressure on the communication network as a substantially (often over 3x) greater number of jobs are migrated in an attempt to balance workload. The graph also appears to show a peak in slowdown at 8 processors, and a relative improvement in performance for a greater number of partitions. Our analysis indicates that this relative improvement derives from the fact that, given enough partitions, frequently traversed nodes will inherently end up in different partitions simply through an effort to equalize the number of nodes. As profile based partitioning can be performed offline in a "do once, use forever" fashion, we see no benefit to avoiding a detailed, profile based partitioning step.

6.9 ISA Optimizations

The frequent and repetitive nature of many of the operations performed during search phase speech recognition code suggests the potential for ISA level optimizations. By incorporating special purpose instruction to address some of these frequent operations (particularly those that may be useful in other domains as well), it may be possible to dramatically decrease the overall instruction count, and improve performance/power characteristics by once again increasing efficiency of resource use.

After careful analysis of low level program characteristics, we identify two potential instruction level optimizations. The first addresses the extensive amounts of control flow and conditional execution required during model evaluation. This comes from the need to evaluate incoming paths, and propagate only highest probability paths through the search. The source of this behavior, therefore, is algorithmic, and likely to appear in other similar applications. The second optimization is potentially somewhat more domain specific. As discussed in earlier chapters, Sphinx–2 minimizes the need for floating point operations in much of the search phase by translating floating point operations into the logarithmic domain space. As a consequence of this translation, a number of logarithmic addition operations are necessary, opening up a potential optimization point.

Unfortunately, neither of these optimizations demonstrates significant usefulness. The reason for this lack of usefulness is different for each, and is somewhat revealing as to the nature and behavior of the underlying application and base instruction set. As such, this section will explore each optimization, and discuss why each is impractical, or why other alternatives are better.

6.9.1 Compare, Select, and Sort

Recall from previous discussions of the operation of Sphinx that the evaluation phase of search functions by propagating highest probability paths one level forward through the linguistic model. In order to perform this task, it must determine the best incoming path at each node through a scoring system. Based on the results of this comparison, path specific data and history information from the appropriate path must be copied forward. In practice, this requires a series of conditional statements to sort amongst the input scores and select the best for propagation, along the lines of:

```
s1 = path 1 score;
s2 = path 2 score;
s3 = path 3 score;
if(s2 > s1) {
   if(s3 > s2) {
      path_record = path_3_record;
   } else {
      path_record = path_2_record;
   }
} else {
   if(s3 > s1) {
      path_record = path_3_record;
   } else {
      path_record = path_1_record;
```

```
    }
  }
```

We attempt to alleviate the need for some of these conditional steps, and simultaneously reduce the number of executed instructions by recognizing that compare instruction and a subsequent "select and move" functionality can achieve the same effect. We therefore implement both two and three way compare, select, and move operations, using a small extension to the existing flag register to hold comparison results. Thus, the previous segment of code becomes:

```
s1 = path 1 score;
s2 = path 2 score;
s3 = path 3 score;

/* This CMP_MAX instruction sets the two
   bit index (00, 01, 10) of the
   largest provided value in the flag register
   space */
3WAY_CMP_MAX(s1, s2, s3);

/* This SEL_MOV instruction reads the result
   of the CMP_MAX to move the appropriate value
   into path_record. */

path_record = 3WAY_SEL_MOV(path_1_record, path_2_record, path_3_record);
```

To a first order analysis, this optimization would appear to reduce the number of instruction that must be executed, and thus improve overall performance. Unfortunately, the results of implementing this ISA revision show that no performance gain is achieved, and

in some circumstances we see a small but notable performance loss instead. This result is observed despite an actual reduction in the number of dynamically executed instructions.

To understand the source of this curious behavior, we must understand a key aspect of the ARM ISA, the ability to place execution conditions on any instruction in the instruction set. As a result of this per–instruction predication ability, and the nature of the 3–way move instruction, this ISA optimization actually results in a greater number of memory requests (and potential misses), and increased overall delay. Furthermore, this predication ability minimizes the negative impacts of control flow modifications, reducing the potential for benefit in this regard.

To understand how these factors affect performance, we consider pseudo–assembly level code for a simplified two–way select, which demonstrates the intuition we discuss without excessive complexity. First, consider how the original code might look for a two way select and move operation. Thus, C-code similar to:

```
s1 = path 1 score;
s2 = path 2 score;
if(s2 > s1) {
  path_record = path_2_record;
} else {
  path_record = path_1_record;
}
```

becomes (in pseudo–assembly):

```
CMP ( r[s1], r[s2] );  /* compare registers containing s1 and s2 */

/* We assume CMP sets a flag to TRUE if arg1 > arg2 and FALSE otherwise */

[if TRUE]  LOAD r[path_record], m[path_1_record]; /* load path 1 record */
[if FALSE] LOAD r[path_record], m[path_2_record]; /* load path 2 record */
```

Note that, due to the ability to predicate individual instructions, control flow operations are essentially unnecessary. Rather, the appropriate LOAD instructions are predicated as necessary to execute only if the COMPARE results are appropriate. Thus, this ability to conditionally execute individual instructions goes a long way to addressing code efficiency in this environment. Now, consider the optimized form:

```
s1 = path 1 score;
s2 = path 2 score;

2WAY_CMP_MAX(s1, s2);
path_record = 2WAY_SEL_MOV(path_1_record, path_2_record);
```

which becomes:

```
2WAY_CMP ( r[s1], r[s2] );  /* compare registers containing s1 and s2 */

/* 2WAY_CMP sets a flag to the index of the largest value */

LOAD r[tmp1], m[path_1_record];
LOAD r{tmp2], m[path_2_record];

2WAY_SEL_MOV  r[path_record], r[tmp1], r[tmp2];
```

While this example has been greatly simplified for clarity, the critical problem with our optimization in this environment is quite clear. The architecture is, fundamentally, an explicit load/store architecture with register to register operations. Thus, in order to implement the 2WAY_SEL_MOV instruction as a register–to–register operation, the program

must first explicitly load both possible values. Allowing the SEL_MOV operations to access memory directly would require re–tooling of the pipeline in order to perform the initial comparison to select the correct memory address, then perform address generation and do the correct memory lookup. While this re–tooling is not a problem in itself, it would achieve nothing more than what conditional execution provides on the base architecture.

It is important to note that on actual compiled code (as opposed to this example), and for three–way comparisons, the ISA optimization *does* reduce the number of actual instructions executed. This reduction comes from optimization and re-use of results across multiple moves which are not depicted here. The basic problem of having to issue a greater number of memory operations, however, remains. If these extra loads miss the data cache even occasionally, any performance benefit of more efficient program flow is quickly lost.

While this optimization failed to demonstrate improved performance, it does point to one important conclusion. If conditional instruction execution were not available, and the code segments discussed thus far were implemented with a series of actual control flow operations (branch instructions), performance could suffer considerably. Thus, either conditional instruction execution, or the ISA optimization discussed here, are beneficial to this application.

6.9.2 Logarithmic Addition

A second potential ISA optimization we explore is the incorporation of logarithmic addition facilities. As mentioned previously, the nature of data manipulations in Sphinx leads to a number of logarithmic additions during Gaussian Scoring. Sphinx–2 performs these logarithmic addition by utilizing a pre–computed logarithmic addition table. Table lookups are a fairly standard way of performing such operations, and as such this behavior is not intrinsically unusual. It does present the possibility, however, of improving performance and reducing demand for memory resources by allowing small lookup tables such as this one to be stored on–chip, and/or replicated for the benefit of individual processing elements.

We evaluate the potential benefit of a high speed log_add instruction considering a

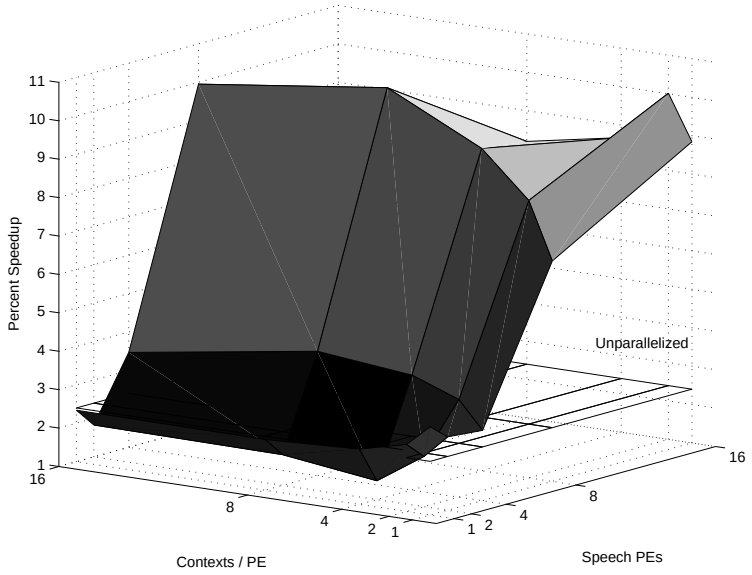


Figure 6.8: Performance Improvement of Logarithmic Add Instruction - This figure depicts the potential performance of adding a high speed LOG_ADD operation (relative to the original code), utilizing a local copy of the lookup table. This optimization is generally beneficial, and generally more so across a larger number of processors. The benefits taper off somewhat as a greater number of contexts are added, and the extra latency in the original code is better tolerated.

version of our infrastructure with such an instruction available, and no overhead for its use. The results of this evaluation (relative to the original code) are depicted in Figure 6.8 and demonstrate that this optimization is generally beneficial, though not substantially so for most configurations. As one might expect, since this reduces the amount of time needed to perform scoring, an increased number of processors shows an increased relative performance gain. By contrast, in most cases an increased number of contexts reduces the relative performance gain, as the extra latency is better managed in the original configuration.

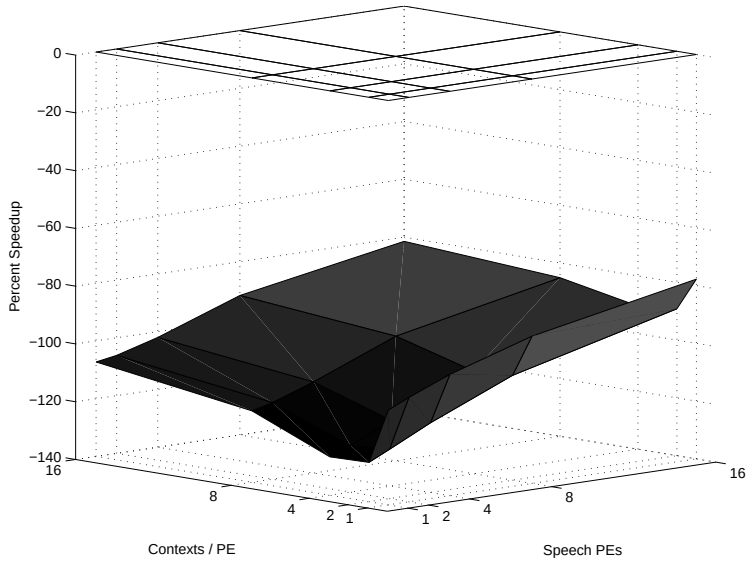
While this optimization demonstrates some promise, we determine that in is not sufficiently general for most applications. This specific operation constitutes a fairly small (if often executed) segment of code that is not likely to be seen in many other applications because it does not address an algorithmic characteristic of the domain. Furthermore, it is part of the Gaussian Scoring segment of search. While the use of vector quantization and other techniques obfuscates this issue slightly, the fact is that the Gaussian Scoring operation constitutes a fairly straightforward, well–understood mathematical translation which

is likely to be useful to other applications in the domain. This also makes Gaussian Scoring a nearly ideal target for ASIC style optimization, a fact determined and performed by other groups already [65]. Thus, given the application specific nature of this particular operation, and the fact that it can be subsumed in a more general ASIC design, we abandon it as a optimization candidate for further study.

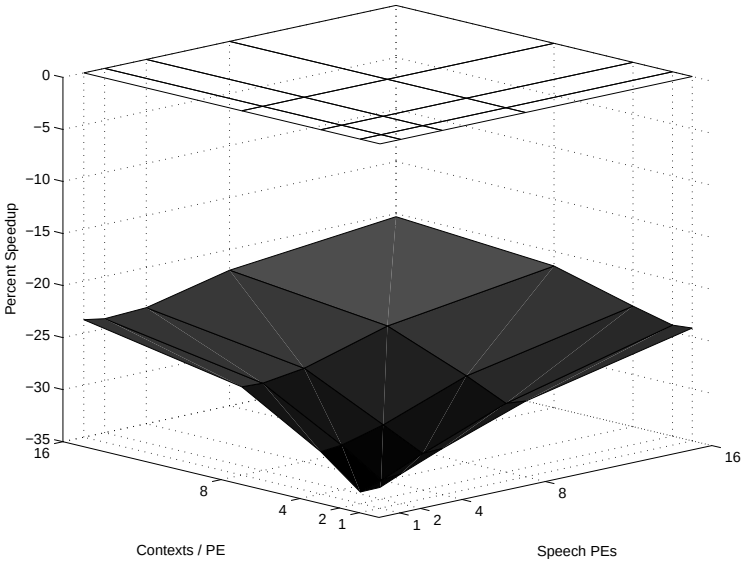
6.10 Reduced Clock Rates

One final processor architecture study we wish to consider is the potential for reduced clock rate execution. Recall the substantial potential performance surplus seen in the idealized performance discussion. This opens up the possibility of utilizing voltage and frequency scaling techniques to reduce overall power consumption while maintaining realtime performance. The degree to which reducing clock rate on the speech co–processor will affect the overall performance of the system is strongly tied to the amount of computation that would otherwise occur between existing system latencies. Thus, what we are really exploring is how useful those extra computation cycles are at masking the latency of stalled contexts

We evaluate this variant by considering simulation environments in which the speech co–processor operates at 200MHz and at 100MHz. The main system processor is always maintained at 400MHz. The results of these reduced clock rates relative to a 400MHz speech co–processor are shown in Figure 6.9. This figures show fairly substantial performance degradation at both 100MHz and 200MHz. While added processing resources and added contexts do tend to reclaim at least some of the performance loss, this effect is minimal at best. This strongly suggests that while some amount of frequency scaling may be possible, a fixed low–frequency system will incur substantial performance penalties. It further demonstrates that a considerable amount of computation is either directly exposed, or is being hidden behind other system latencies, and that extending out this computation is not beneficial.



(a) 100MHz vs. 400MHz



(b) 200MHz vs. 400MHz

Figure 6.9: Performance Loss of Reduced Clock Rate Co–Processor Core - This figure depicts the performance (relative to a 400MHz speech co–processor) of reduced clock rate designs. A 100MHz system is shown in (a), and a 200MHz system in (b). Clearly, a substantial amount of computation is exposed or can be hidden behind other system latencies in the 400MHz model, resulting in considerable slowdown when that computation is extended by slower clock rates.

6.11 Conclusions

This chapter explores potential architectural optimizations and bottlenecks of our low power parallel processing environment. We determine that our approach to program execution is very tolerant of long latencies in the system, and that added thread contexts are able to maintain high processor utilization even in the midst of such latencies (assuming no bandwidth related bottlenecks). This exploration further reveals that register utilization is very low on the main processor during parallel code execution, suggesting the possibility of virtualizing the "background context" utilized to perform floating point computations. Dynamic load balancing and detailed profile based initial static partitioning are found to be beneficial to maximum performance, and a small buffer of runnable jobs is found to result in better overall processor utilization. Perhaps most importantly, we determine that due to low utilization, a system with a highly constrained 8–bit control bus for inter–processor communication performs nearly as well as one with an idealized bus model.

Based on the analysis presented here, we focus on an architectural model in which the main XScale processor has support for virtual partitioning of its register file to allow for the execution of multiple thread contexts. We further assume that speech processing elements use a subset of standard ARM instructions (finding little benefit from specific ISA optimizations), and that they contain a 5 element work queue (pending job buffer) which is filled once available thread contexts are filled. We further assume a system with a slow, potentially distributed dynamic load balancing infrastructure, and an 8–bit control bus operating at processor frequencies. We also utilize initial static partitions based on usage weight profiled graphs, and limit further evaluations to a maximum of 16 processing elements with 16 contexts each.

CHAPTER 7

Memory System Evaluation

In the previous chapter, we analyzed potential architectural performance and on–chip bottlenecks inherent in our design. In general, we feel that this architectural and programming model effectively exploits the concurrency available in the speech recognition domain. The second major aspect of performance, exacerbated by the concurrency potential previously discussed, is the limitation imposed by memory system bandwidth.

In this chapter, we perform a design space exploration of potential memory configurations to support our processor architectural model. This essentially amounts to a bandwidth problem. A certain amount of memory bandwidth is available for a given memory system configuration, and a certain amount is required by the program. In order to match the two, one can either reduce the demand on the memory system by altering memory stream dynamics (reducing demand by caching, or distributing demand by data stream partitioning), or increase the available bandwidth (moving from a SDRAM system to a DDR system). This chapter will explore these potential design space variations in light of power and performance constraints to determine which configurations are preferable.

We begin by revisiting some of the assumptions previously discussed, such as the presence and size of the processing element cache structures. We will then expand our analysis to consider multi-level cache hierarchies, and various interfaces to main memory. From this analysis, we will develop an understanding of potential existing memory system configurations that may be useful in supporting speech recognition.

We will then extend our analysis into more advanced architectural models, still within the bounds of current technology, but not seen on our target domain of low–power systems. This analysis will include flash and ROM based designs, as well as embedded DRAM technology.

Our evaluation standard will also shift slightly as we progress through this exploration. We begin by focusing on memory characteristics of the application, and techniques to provide the necessary memory throughput to achieve performance objectives. As we reach levels of memory sophistication that provide the necessary bandwidth, the goal shifts toward minimizing energy consumption. To this end, it now becomes vital that we present a consistent and detailed model for estimating overall performance and system energy consumption.

Performance accuracy is tied to simulator accuracy. As with our previous analysis, we utilize a multi–processor version of the SimpleScalar toolset. Our modifications account for processor characteristics, communication resource utilization, and memory system characteristics. A detailed analysis of the simulation infrastructure utilized for this work, including consideration of potential simulation inaccuracies, is presented in Appendix A. To estimate energy, we employ an activity based model that attempts to account for active and idle energy dissipation on all major computation, storage, and communication components of the system. A detailed power audit and description of our estimation framework is presented in Appendix B.

Furthermore, the added energy constraints combined with the existing performance constraints complicate the task of identifying which configurations are superior to which other configurations. While we will present more general "optimal design" analysis in later sections, in this chapter we will present much of our analysis in terms of the energy–delay product (EDP - calculated as Joules–Seconds). While it is tempting to assume that the lowest EDP that achieves realtime performance is the optimal design, it is important to note that we are exploring architecture models, and not speech recognition operating parameters. As such, this casual conclusion would only be valid for the specific vocabulary size and

search parameters of our evaluation framework, and may vary substantially if one varies the parameters of the underlying application. While we will essentially make this assumption for practical considerations, and further make the assumption that while a specific configuration presented here may not be ideal for all speech recognition tasks, following the lowest EDP will point us to general optimization strategies that can be applied across the board, it is important to remember this caveat when considering the results presented.

In recognition of the variability in specific performance metrics, we will present most of our analysis as relative performance between configurations, allowing us to focus on how much a particular design parameter affects performance in a metric somewhat independent of the application. That is, we assume the relative performance of one architectural configuration over another is generalizable to all instances of the speech recognition task, even if the specific "runtime–based" performance is not.

7.1 Instruction Stream Analysis

The main bottleneck in speech recognition is access to the large knowledge base. Algorithmically, the program traverses only a small set of instructions at any given time, and instruction reference stream locality is rather high. We have, therefore, essentially ignored the effect of instruction fetches thus far, as a number of fairly simple solutions may be employed. As we begin memory system analysis and consider more realistic configurations, however, we wish to account for instruction effects. Thus, we will analyze potential solutions briefly in this section before moving on to the data stream, which is the crux of this chapter.

An analysis of instruction stream locality characteristics as cache miss ratios for a single processing element are depicted in Figure 7.1. Not only does a very small cache reduce the miss rate to near zero, but most of the misses occur at well identified program transition points. Furthermore, these program transition points are congruent with sections of parallel execution, suggesting the potential of preloading instruction data into processing elements during serial execution when memory bandwidth is not as significant a bottleneck. This

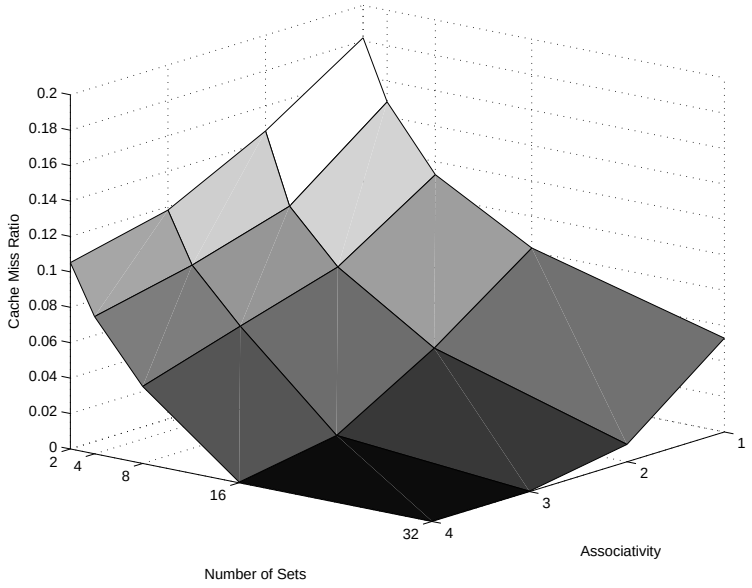


Figure 7.1: Cache Miss Ratios for Processing Element Instruction Stream - This figure depicts miss ratios for the instruction reference stream of a single speech processing element. Based on this data, and recognizing that all processing elements essentially share the same instruction code during any segment of execution, we select a 1K, 2–way (16 sets) cache shared by each set of two processing elements in further evaluations. Note that, while this configuration appears to have a miss rate of around 4%, most of these misses occur at well defined program transition points. Thus, preloading during serial execution brings the actual miss rate to near zero during parallel execution.

amount to a minor modification of a standard vector processing technique. Furthermore, as expected from knowledge of program characteristics, stream references have high locality between processing elements during an execution phase as well. This suggests that the only constraint on multiple processing element sharing the same instruction store is bandwidth into that instruction store.

Taking these factors into account, we extend our design to include a set of 1K instruction caches. Each cache is shared by two processing elements, and each processing element buffers a single cache line (32 byte lines equates to eight instructions) on reads, to reduce demand on the cache itself. The caches are all preloaded with instructions for the next parallel phase during serial execution via a special command sequence executed on one of the processing elements. This command sequence issues loads to fetch the necessary instruction blocks, and all instruction caches snoop the data off of the shared memory bus. As a result of this configuration, the parallel run–time miss rate is effectively zero.

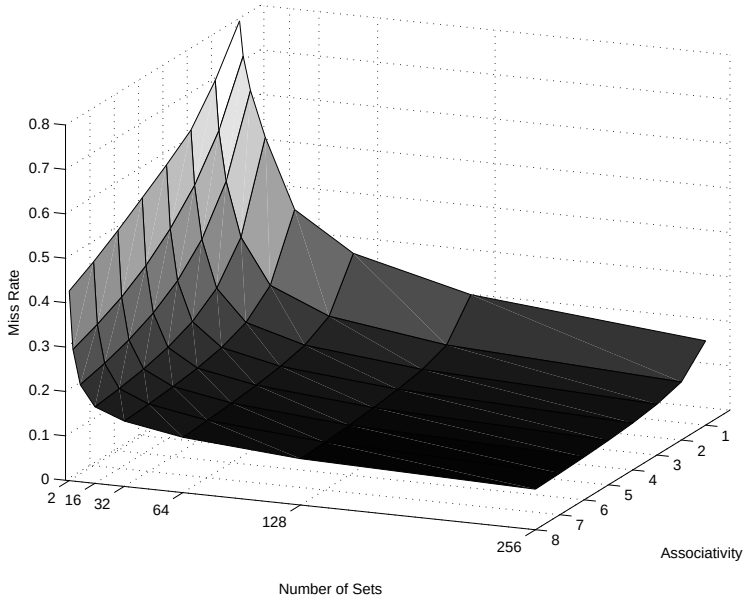


Figure 7.2: Cache Miss Ratios for Single Speech Processing Element - This figure depicts miss rations for the data reference stream from a single speech processing element in a multi–processing element system (assuming 32–byte line size). Cache sizes range from 64 bytes (upper–left) to 64K (lower–right). Based on this data, we employ a 2K, 4–way cache in our base analysis.

7.2 L1 Data Cache Configuration

We begin our memory system analysis by re–visiting the small L1 cache associated with each processing element. Earlier analysis assumed that this was a 2K, 4–way cache configuration. In this section, we will work through the analysis behind that architectural decision.

As discussed in earlier chapters, the overall memory reference patterns of this application domain show many stream–style characteristics. Data for a senone is brought into the processor during Gaussian Scoring, is processed, and in then left untouched until the next input frame. Similarly, each of the currently active linguistic model nodes is evaluated sequentially over the course of multiple passes within a single iteration (once for evaluation, and once for tree pruning). Despite these general stream characteristics, some locality is inevitable due to multiple accesses to the same data node during a pass. Furthermore, each processing element must deal with artifacts of parallelization, such as local lists of active model nodes, which show considerable locality when accessed.

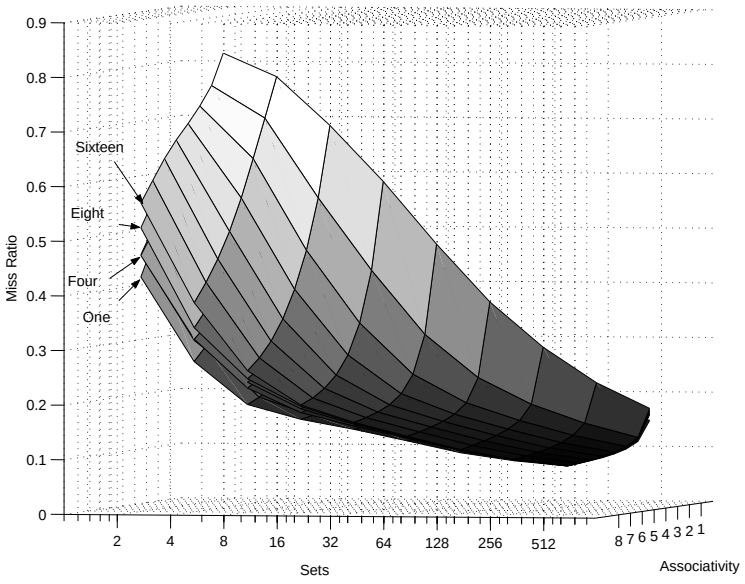


Figure 7.3: L1 Cache Miss Ratios by Number of Contexts - This figure depicts miss ratios for a single processor, exploring the effects of varying the number of thread contexts. Each surface represents a "number of contexts" configuration, as defined by each label. Note the logarithmic scale along the forward axis.

We evaluate the potential of capturing this locality by considering miss rates on a range of cache configurations for the data request stream of a single processing element in an array. The results of this analysis are depicted in Figure 7.2, and show that small local caches can provide a substantial benefit in capturing available spatial locality. At approximately 16–32 sets, and 2–4 associativity, increased cache resources provide diminishing returns on miss rate. It is from this analysis that we adopt a 2k, 4way L1 cache in our evaluations. Such a configuration has the added benefit of making this architecture more amenable to other problems in the general stochastic search domain.

Given our multi–threaded implementation, an obvious question in relation to this local cache is whether it should be unified across all thread contexts, or whether a model in which each context has exclusive control over some portion of the cache is better. This really becomes a question of utilization efficiency, whether more of the cache is utilized by common data such as local active lists, or context exclusive data such as linguistic model nodes. If exclusive data predominates, then inter–context conflicts could result in a less efficient utilization of the total cache resource. For example, context two may begin working

on a model node, and in the process kick out model data for a node currently being processed by context one. If, however, shared data predominates, one context may effectively prefetch data for another context. As a natural result of our scheduling methodology, one could even imagine an environment where one context is able to complete work on a data element while the data for another context is being fetched, then issuing loads for its next data element before switching off.

We explore whether a partitioned (by context) cache structure is useful or not by considering two approaches. First, we evaluate the cache performance of a single processor across an array of context configurations. The overlapping surfaces of cache miss ratios for such configurations are shown in Figure 7.3, and suggest that while added contexts clearly leads to increased misses on smaller caches, the miss ratios mostly equal out by the 2k configuration previously identified. Note the use of a logarithmic scale along the "number of sets" axis to better visualize the different surfaces. This strongly suggests that the L1 caches are mostly capturing metadata needed by the application, as opposed to any substantial quantity of model data for speech recognition itself. We further verify this by by explicitly allocating a subset of cache lines to each thread context. Loads of stream oriented data (Gaussian Score information, Linguistic Model nodes, etc) are directed to these exclusively allocated cache lines, while all other references are directed to a remaining set of cache lines shared by all contexts. We attempt to match the size of the context exclusive cache segments to the size of stream data nodes in the application, thus ensuring that knowledge base data needed by one context is never kicked out of the cache by another context. We find that this structural modification is ineffective, and in fact leads to an average 20% slowdown relative to a fully shared cache model. Once again, this supports the notion that the L1 data stream is primarily concerned with application metadata. Bypassing knowledge base data around the L1 entirely, however, also leads to a substantial slowdown, suggesting that the L1 is effective in maintaining a few knowledge base elements. In sum, these results point back to our hypothetical utilization picture for the L1, maintaining mostly program metadata, but also servicing requests for an actively running context while simultaneously

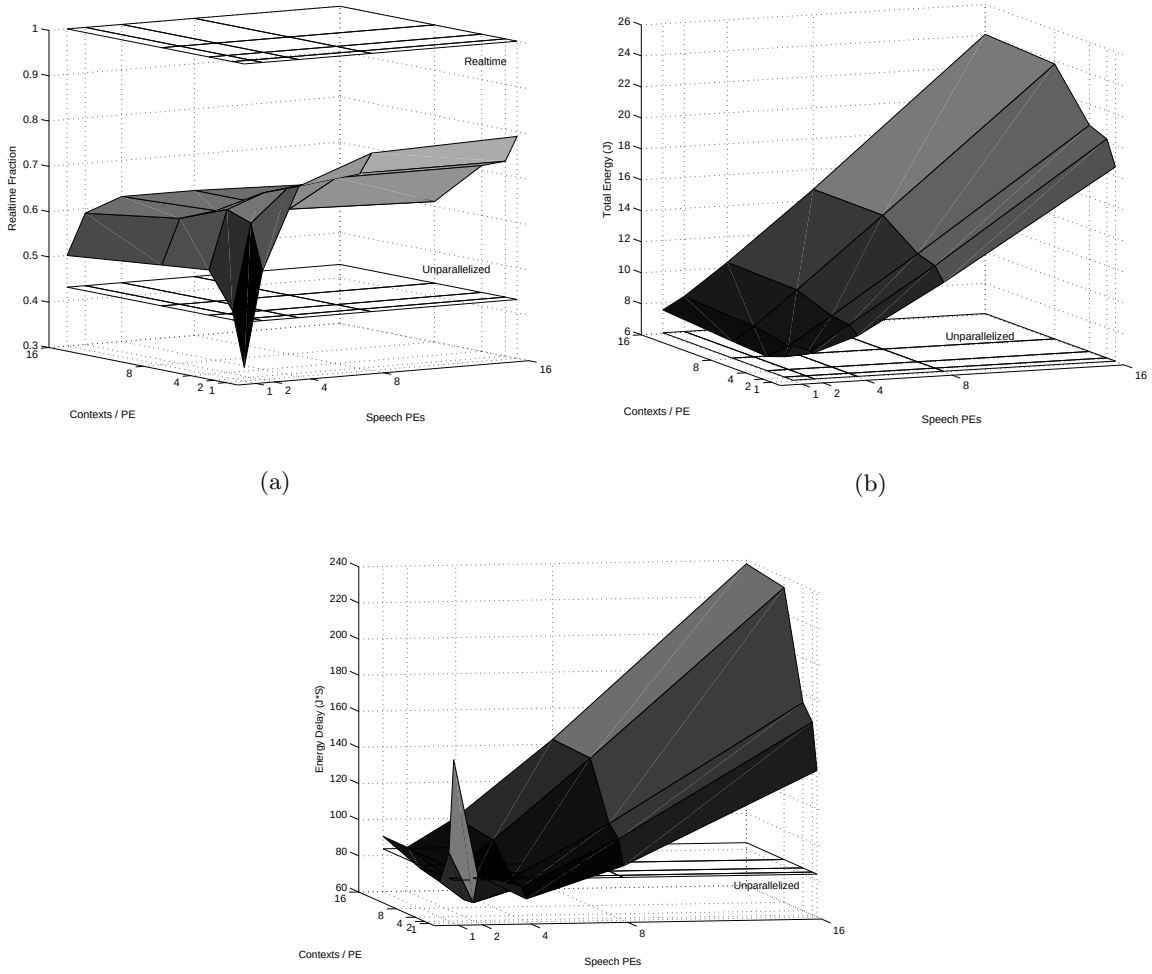
pulling in data in preparation for execution of another context.

While the analysis thus far demonstrates the usefulness of a small local cache structure on each processing element, Figure 7.4 shows that with a 100MHz SDRAM system, performance is still substantially constrained. Indeed, the total memory bandwidth of this configuration is simply insufficient to provide anything close to realtime performance. Consequentially, the energy delay product (shown in Figure 7.4c) also fluctuates around the area of the original unparallelized code, and while some benefit can be achieved for a small number of processors, it remains fairly unsubstantial. An interesting observation in these graphs is the relative loss in performance for added contexts. As the addition of thread contexts should help tolerate longer memory latencies, this is initially rather surprising. We discover, however, that the source of this performance decrease is a rapid growth in the average memory latency of a request due to a greater pile–up at the memory system interface. This growth in average latency far exceeds the potential added computation provided by these extra contexts, leading to an overall performance decrease. We will consider solutions to this problem in later chapters. We first must arrive at a memory system configuration more capable of providing for real–time performance. Before we consider more advanced memory systems, however, we will explore a few other potential adjustments to maximize the performance we are able to achieve.

7.2.1 Streaming and Prefetching

We have discussed at numerous instances the stream–characteristics of this application domain. This immediately raises the question of utilizing stream handing techniques to improve performance. For example, Mathew et al [65] modify the Gaussian Scoring component of Sphinx3 to improve performance by streaming knowledge base data through the processor.

Attempts to make similar optimizations in our infrastructure fail for a number of reasons. During normal operation, the Gaussian Scoring phase of both Sphinx2 and Sphinx3 utilize feedback data on currently active linguistic model nodes to process only the currently active



(a) (b)

(c)

Figure 7.4: Performance Breakdown of base SDRAM System - This figure depicts the performance of our base architecture (with the 2K cache) with a full 100MHz SDRAM memory system implementation (a), workload energy dissipation estimates (b), and EDP (c) for each such configuration. Compared to our ideal memory system evaluations, this clearly demonstrates the bottleneck imposed by memory bandwidth. Note the performance loss for added contexts. This is due to bandwidth constraints, and is discussed in later chapters.

senone set. In order to truly exploit stream characteristics, the Mathew work breaks this interdependency and simply computes scores for all senones. This allows them to treat the acoustic knowledge base as one large contiguous memory space that can be streamed through the processor in total. By contrast, leaving the feedback path in place causes the Gaussian Scoring system to jump around in the acoustic knowledge base, picking out only active senones and disrupting this characteristic. The tradeoff is in the time spent computing scores for senones that are currently inactive and will not be utilized. In our evaluations, this tradeoff leads to significant reductions in overall system performance due to the significantly larger number of scores that need to be generated. Indeed, the Mathew work alleviates this problem by packing 10 frames worth of data together and processing them in a single pass, and by stripping some of the accuracy of the Gaussian Scoring. As Sphinx3 performs a complete scoring at run–time, such optimizations are feasible without significant loss of accuracy. In our Sphinx2 environment, vector quantization has already reduced the acoustic model accuracy, and the scoring phase operates more as a series of table lookups than as a straightforward mathematical computation.

Similar problems arise in the linguistic model evaluation. While the nature of data evaluation is stream–esque in the "use once and discard" sense, the actual memory reference pattern does not show stream characteristics, as the evaluation model only accesses currently active nodes. This leads to the question of prefetching necessary data into the processor. While earlier analysis demonstrated that simple hardware prefetching based on reference stream patterns is not viable, intelligent software based prefetching remains a valid option. Indeed, at the software level, Sphinx memory access patterns would seem imminently predictable. The Gaussian Scoring phase operates off of a list of currently active senones which could be sequentially prefetched ahead of processing, and the linguistic model evaluation operates off of a similar list of currently active model nodes.

In general, we assume that the bandwidth bottleneck imposed by our use of concurrency would eliminate any potential benefit of prefetching efforts. We consider the possibility, however, that limited prefetching when memory bandwidth is not over–utilized may have

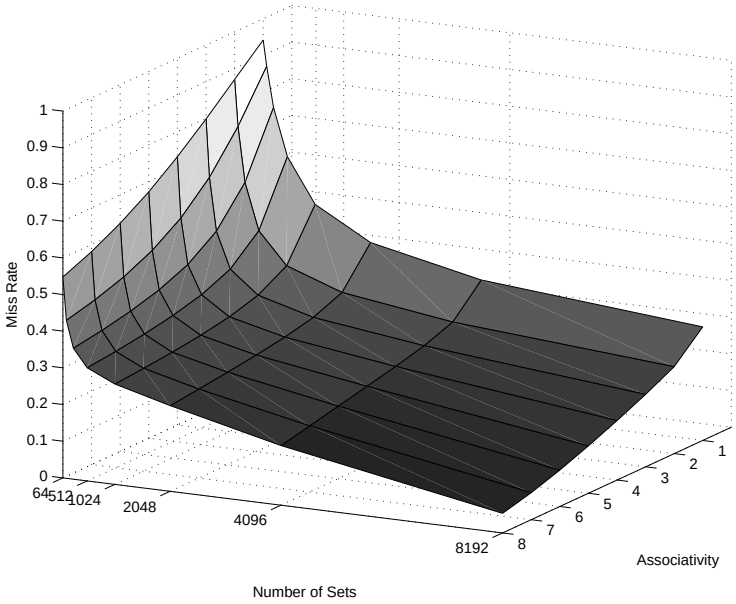


Figure 7.5: DL2 Cache Miss Analysis - This figure depicts the cache miss ratio for the data memory reference streams from speech processors with 2k, 4way per–processor L1 cache and main processor with standard cache configuration. Cache sizes range from 2K (upper–left) to 2MB (lower–right). The knee in the curve occurs at a cache size of around 128K for a 4–way cache.

a beneficial impact on performance. To this end, we consider two alternative prefetch implementations: a separate prefetch thread run on a standard hardware context of a given processing element, and a separate, independently programmed prefetch engine.

In both cases, we find no consistent performance benefit over simply adding an extra thread context to tolerate the latency that a prefetcher attempts to reduce. In essence, the need for a programmable prefetch engine implies that the system must duplicate computation (once along the critical path for the prefetcher, and once for the actual data processing). Thus, it is either equally efficient to simply run an independent workload while awaiting the memory request, or the added prefetch request stream only exacerbates existing memory bandwidth constraints, producing no benefit.

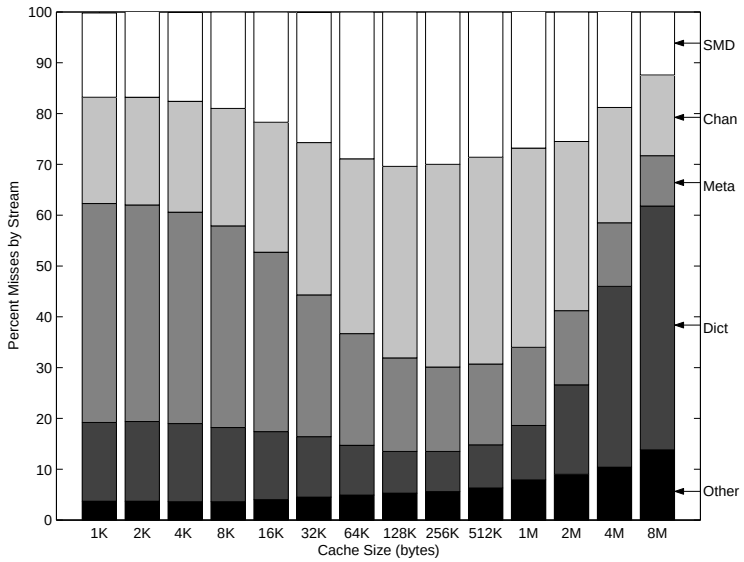


Figure 7.6: DL2 Cache Miss Stream Analysis - This figure depicts a fractional breakdown of L2 cache misses for various L2 sizes (4–way) by the data element the missing request references. It demonstrates that the initial falloff in misses in the region of 128K cache sizes corresponds to a falloff in metadata misses, and knowledge base data becomes a more significant source of non–locality beyond this point.

7.3 L2 Data Cache Configuration

As we continue to analyze the memory characteristics of this application space, we make the observation that due to pruning and selective evaluation efforts, the actual working set of the application, while large, is in fact far smaller than the total size of the knowledge base. Furthermore, as the same node or acoustic data may be evaluated across multiple iterations of search, there is significant potential for large scale spatial locality. In this section, we will consider the potential use of large L2 data cache models to capture and exploit some of this locality. It is important to note that the degree of such locality may vary substantially with the specific knowledge base and search parameters used. The analysis presented here, however, should remain fundamentally valid (though the specific tuning points may change).

We begin this evaluation by once again considering cache miss ratios of memory reference streams. In this case, we inspect the overall stream of memory requests seen on the back side of processing elements with 2K, 4–way L1 caches. The results of this evaluation are presented in Figure 7.5, and demonstrate that a large L2 cache could be quite effective in

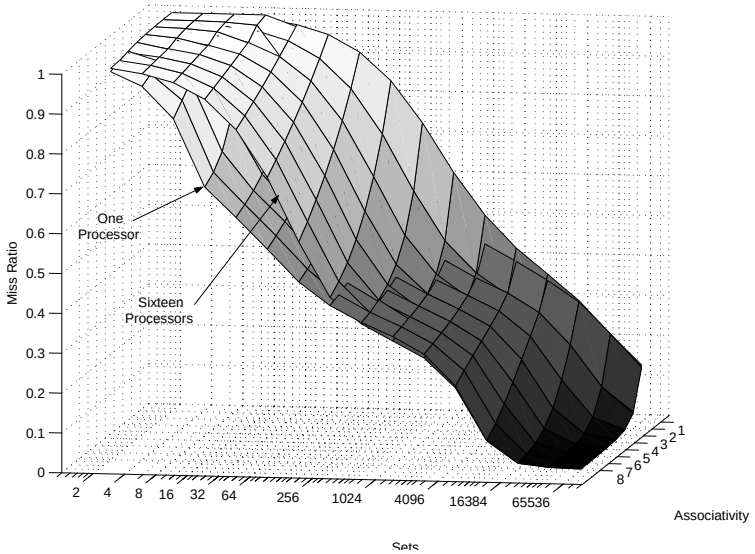


Figure 7.7: DL2 Cache Miss Analysis by Processor - This figure depicts L2 cache miss ratios (note the logarithmic scale) for one and sixteen processors. Note the slower initial falloff in cache misses for 16 processors, which corresponds to greater non-locality in reference patterns to program metadata. Once this data is captured in the cache, both streams consolidate on knowledge base misses.

reducing the traffic out to main memory. We note from this graphic that, while the point of diminishing returns appears to occur at around 512–1024 sets and 2–4 way associativity (corresponding to a 64K–128K size range), the miss rates of these configurations remains in the 30–40% region. Further increases in cache size (particularly through an increased number of sets) continues to improve miss ratios considerably. This is a very different characteristic than that seen in L1 cache analysis in Figure 7.2, which very much levels off after the initial reduction in miss ratio.

It appears somewhat surprising, however, that an application with such a large knowledge base and known stream characteristics should demonstrate a knee in the cache miss curve for such relatively small L2 caches. Further exploration reveals that we are actually observing *not* the working set of the program with respect to knowledge base data, but once again at program metadata (in this case data being kicked out of the L1 cache, then requested shortly thereafter). The slow falloff in cache miss ratio seen after this initial fall off is related to the actual inter–iteration knowledge base working set. This effect is best depicted in Figure 7.7 which shows the percentage of overall L2 misses attributable to each

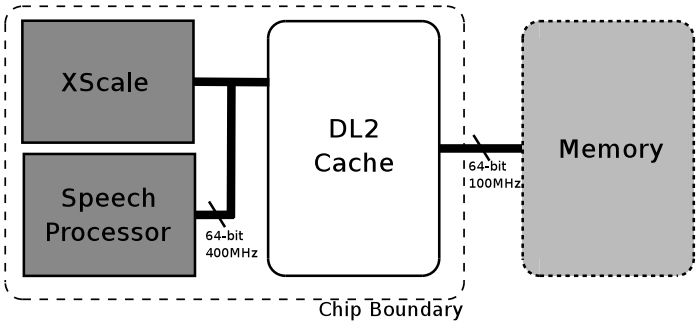
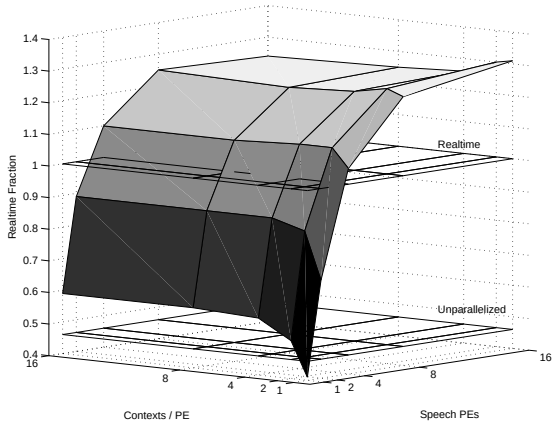


Figure 7.8: L2 Cache System Organization

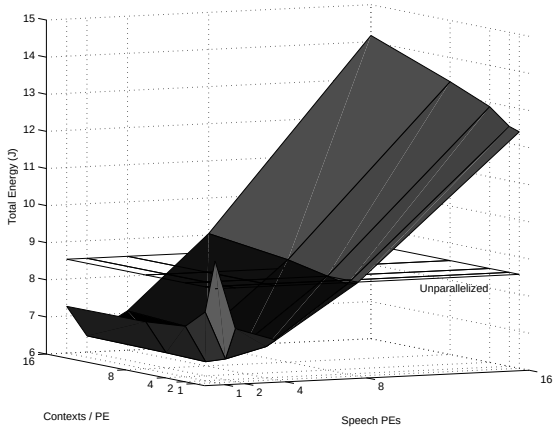
of a number of the data streams in the application for a 4 processor, 4 context system. This figure clearly shows the shift in cache misses from program metadata to knowledge base data in the region of the first level off in cache misses. A second shift from channel and static model data to language model, dictionary, and Gaussian Scoring data, which show the most stream–esque characteristics even across search iterations, is observed as we approach very large cache sizes.

The effects of varying the number of processors (and thus the number of partitions and the overall patterns of access to metadata) is observable in Figure 7.6, which shows cache miss ratios (once again on a logarithmic scale) for the L2 cache given a single processor and given 16 processors. As sixteen processors result in more metadata and a greater number of metadata related L2 requests, the miss rate is substantially higher for such data at smaller cache sizes. Once this metadata is captured in the L2, however, we see the direct effects of knowledge base related data. This data does not fall off until much of the actual knowledge base itself can be captured in the L2, corresponding with the stream percentage shifts seen in the previous figure.

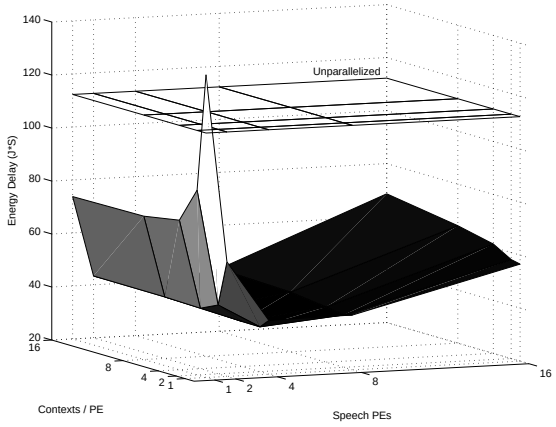
Based on this information, we wish to consider the impact of including a number of L2 cache models to explore the tradeoff between performance through lower miss rates and energy through larger cache area and dissipation. We focus on capturing program metadata, as increases in knowledge base size of more advanced speech recognition systems is likely to always outpace the amount of cache memory that can be placed on–chip. It is, however, important to note that advances in Embedded DRAM technology allows for much higher



(a) Realtime Fraction



(b) Energy



(c) EDP

Figure 7.9: Performance of base system with 128K DL2 - This figure depicts the performance relative to realtime (a) energy (b) and EDP (c) of a base system (2k,4W DL1, 100MHz SDRAM) with a 128K, 4–way DL2 cache.

density on–chip memories. Attempts to capture the inter–iteration knowledge base working set may be feasible using such technologies, but the performance effects of this datapoint are fairly straightforward and are not considered here.

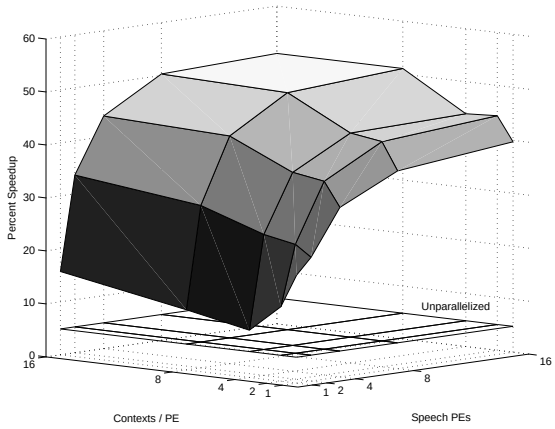
We begin with a 128K L2 cache (1024 sets x 4–way associativity x 32–byte lines), targeting the edge of the initial miss ratio falloff. We model this cache by assuming all misses from L1 caches lead to communication over a common 64–bit bus operating at processor frequencies. Misses from the L2 go out over the main memory bus, which is assumed to operate at the frequency of the DRAM system (100MHz). This organization is depicted in Figure 7.8. Based on access latency information from Cacti, plus a number of protocol related cycles, we assume a 2 processor cycle access latency to the L2. This number is intended to be conservative and provide for protocol / bus synchronization overheads within the L2, and thus does not exactly match the access time metrics provided by Cacti. Rather, we assume that for every half–cycle of time suggested by Cacti, the L2 cache is given 2 full cycles in which to operate. In this case, a single cycle at 400MHz is 2.5ns, so a 128k L2 with an access latency of 1.13ns (less than half a cycle) is assumed to have an internal latency of 2 cycles. This is the latency to service a request once it *reaches* the L2. The actual observed latency seen by a processing element also includes latency to acquire the bus and await any other requests being serviced by the L2. We assumed for the moment that L2 accesses occur on a single port, and are handled internally by a two stage pipeline, allowing the cache to service two requests simultaneously from the viewpoint of the system (sequentially, but pipelined from the viewpoint of the L2).

The performance and energy breakdown for a system with this initial configuration is depicted in Figure 7.9. The performance improvement over the base SDRAM system is fairly substantial. So too is the reduction in overall workload energy consumption, as the added power contribution of the L2 cache occurs for far less time due to the performance improvement. While the performance improvement of this system exceeds realtime for larger configurations, we still see a number of artifacts of memory bandwidth related constraints. First, we see performance essentially level off beyond 8 processing elements (as opposed to

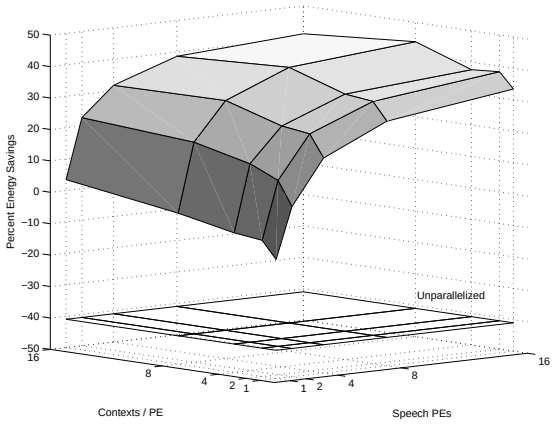
the continued performance improvements seen in the ideal model). Secondly, we once again see performance fall–off after addition of a sufficient number of thread contexts. In this case, the falloff occurs when the number of misses in the L2 create a memory system bottleneck, leading the performance reduction to be tempered considerably relative to the non–L2 case. If we consider the EDP to be the metric by which to differentiate configurations, we find that the lowest EDP in this set of experiments occurs with 8 processing elements, and 2 contexts each. The lowest energy occurs at 4 processors, 2 contexts.

A more direct comparison of the performance and energy improvements of adding a 128K L2 cache can be seen in Figure 7.10, which depicts the relative performance and energy of the L2 configuration over the base system with no L2 cache. This most clearly depicts the benefits of a larger L2 design. An interesting artifact in these graphs is the small performance trench occurring at 4 contexts for all processor counts. In understanding the source of this trench, it is important to realize that these are relative metrics, and that the base data from which these results are derived show dramatic performance improvements until around four contexts are added, at which point performance begins to fall off due to bandwidth constraints. The artifact in these results, then, is simply reflecting the fact that the performance peak for the base system (relative to other base system configurations) is slightly higher than for the L2 based system, leading to a slightly lower relative performance improvement at that transition point. Thus, while the relative performance numbers for a smaller number of contexts depicts the improved performance of the L2 system, the relative performance improvements seen beyond 4 contexts is really more reflective of the rapid *decrease* in performance of the base system. This is best depicted in Figure 7.11, which shows the realtime performance fraction of both the base and L2 systems, looking at only the 2 speech processor case and varying the number of contexts.

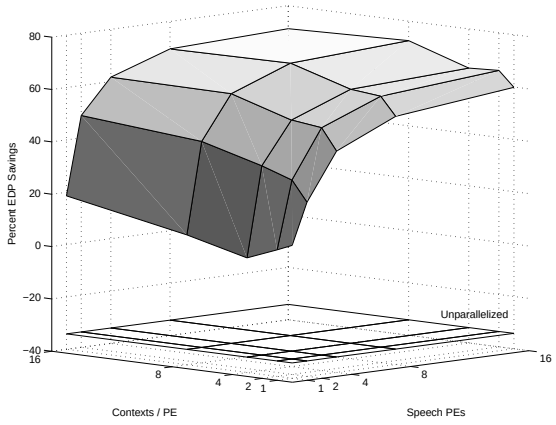
Another important point to note in these depictions in the trivial impact of a L2 cache on the performance of the unparallelized system. This tracks well with the fact that the unparallelized code is limited more by the computational performance of the processor than the bandwidth of the memory system. It is only after we exploit concurrency to



(a) Relative Performance



(b) Relative Energy Savings



(c) Relative EDP Reduction

Figure 7.10: Relative Performance Breakdown of base system with and without 128K DL2 - This figure depicts the relative performance (a), energy (b) and EDP (c) of a system with a 128k L2 cache over one with no such cache.

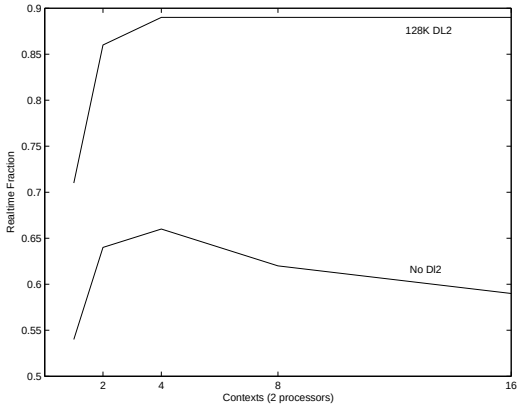
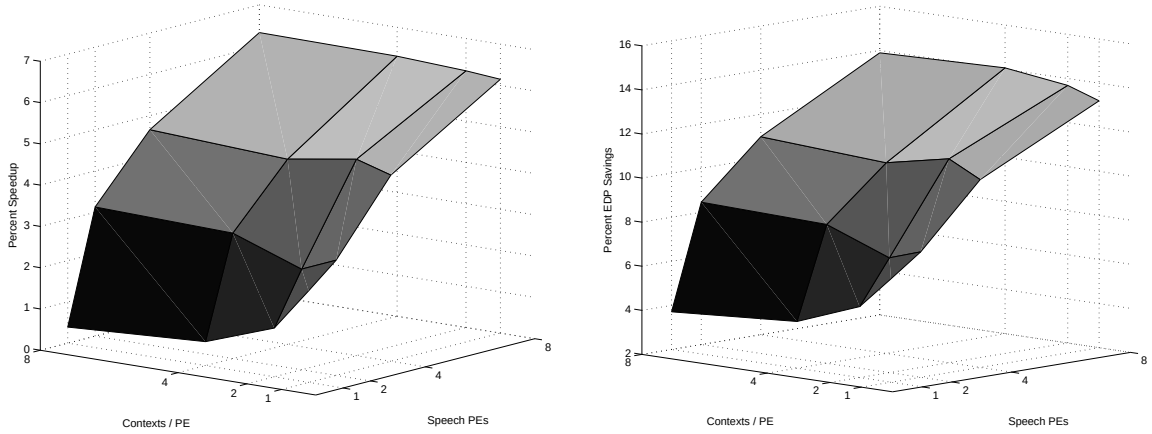


Figure 7.11: Cross Section of 128k DL2 and No DL2 with 2 Speech PEs - This figure depicts a slice of the no–DL2 and 128k–DL2 experiments, looking at a range of contexts for two processing elements.



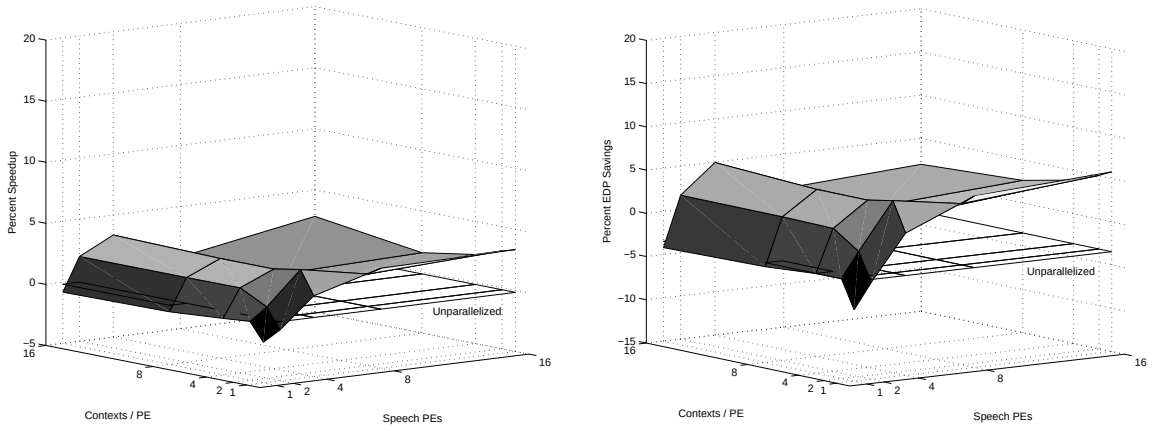
(a) Relative Performance

(b) Relative EDP Reduction

Figure 7.12: Relative Performance and EDP of 128K DL2 over 64K DL2

improve processor–side performance that the memory system effects become visible. Consequentially, the unparallelized system incurs the energy penalty of the L2 without the corresponding performance improvement. By contrast, even the single processing element, single context configuration sees some performance improvement, as even in this configuration there is some concurrency between "background" processing on the main processor and execution of the single speech processing element.

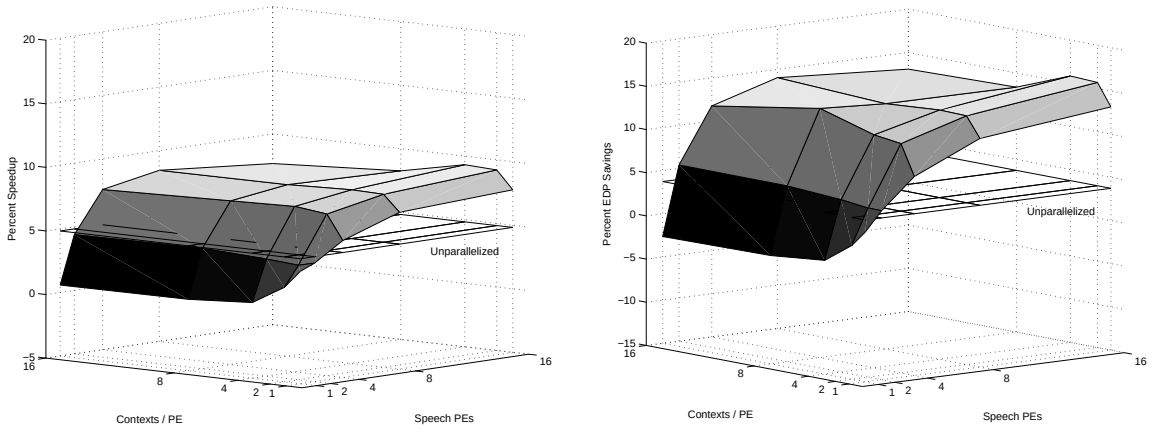
As our L2 cache miss analysis indicates, the 128K cache we have just discussed represents a knee in potential cache impact, but added cache size can still lead to considerable reduction



(a) Relative Performance

(b) Relative EDP Reduction

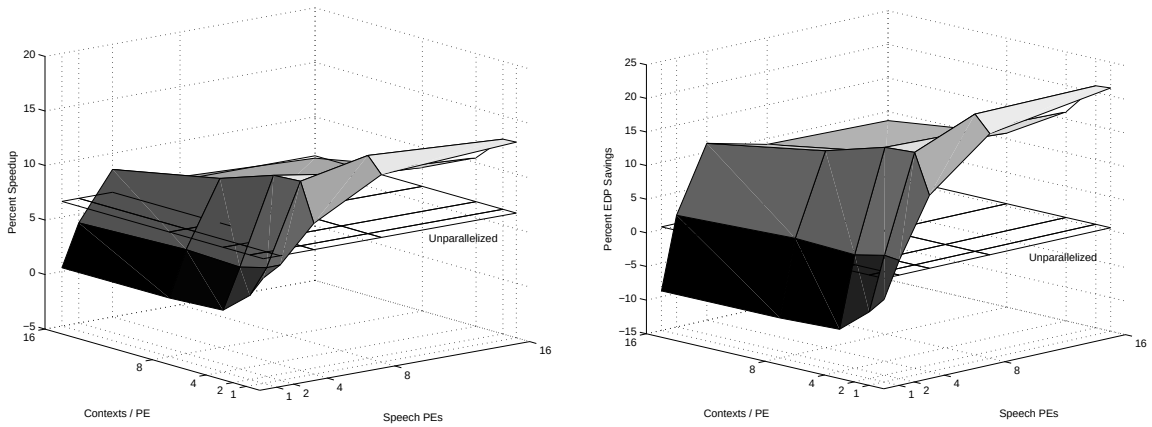
Figure 7.13: Relative Performance and EDP of 256K DL2 over 128K DL2



(a) Relative Performance

(b) Relative EDP Reduction

Figure 7.14: Relative Performance and EDP of 512k DL2 over 256K DL2



(a) Relative Performance

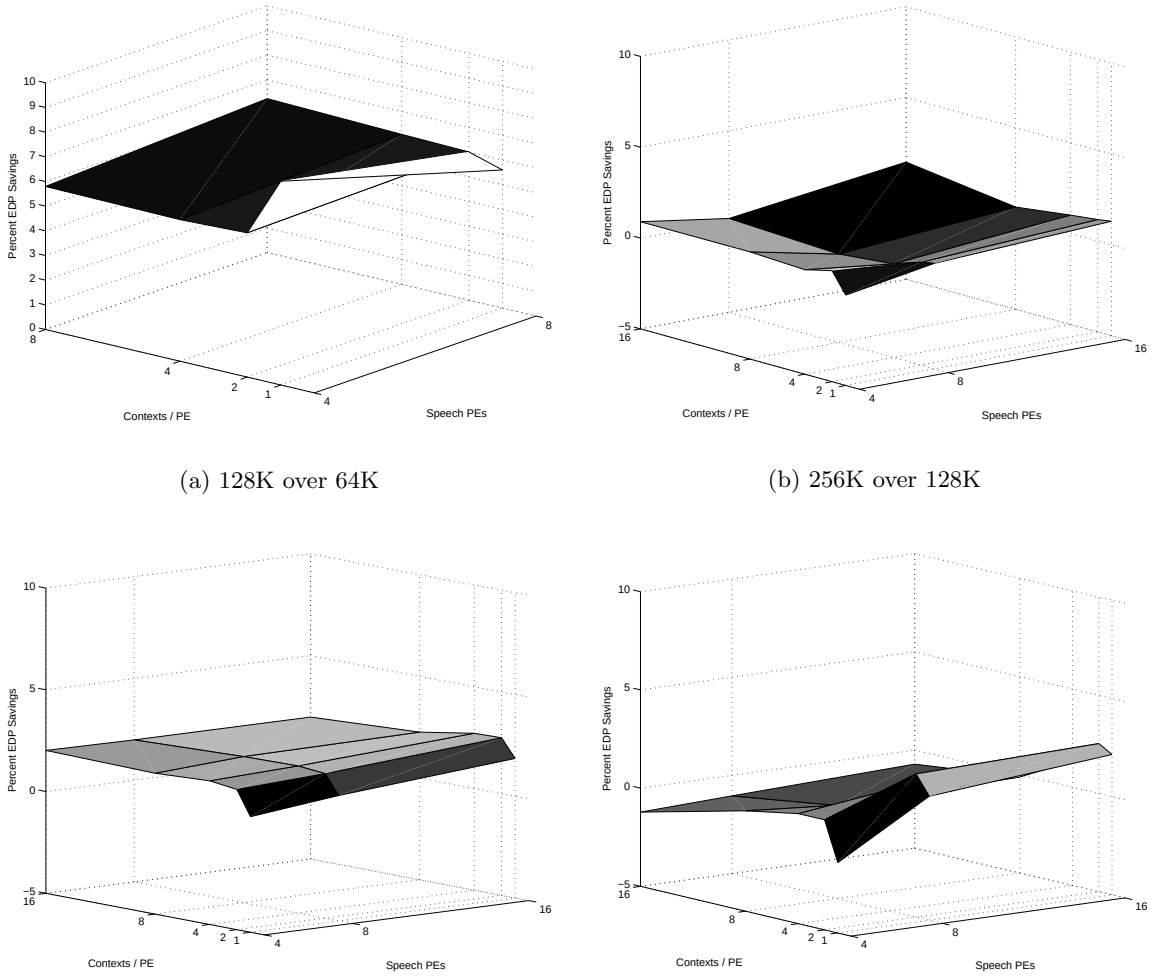
(b) Relative EDP Reduction

Figure 7.15: Relative Performance and EDP of 1MB DL2 over 512K DL2

of cache miss rates. This reduction in cache miss rates would have the effect of improving performance, but also of reducing the number of high–energy off–chip memory accesses. Countering these energy decreasing effects is the increased energy dissipation effects of the larger cache structure itself. In order to explore these effects is greater detail, we consider the relative performance and EDP of processor configurations with cache sizes ranging out to 1Meg. We cap our analysis at this size due to area considerations, as a 1Meg cache is itself rather large given the size of the initial XScale processor system. The relative performance and EDP for various cache sizes stepping up from 64k (slightly below the knee in the cache miss curve) to 1Meg (maintaining 4–way set associativity) are depicted in Figures 7.12 to 7.15. We see from these figures a continued performance and EDP improvement as we increase the L2 size within our selected range. This suggests that the larger cache energy dissipation is a good tradeoff for the reduced computation time and off–chip memory accesses.

Applied Realtime Constraints

While the continued performance improvements and EDP reductions for larger caches would seem on face to suggest the processor should include the largest possible L2 cache, there are a number of important considerations that are excluded in the analysis presented



(a) 128K over 64K

(b) 256K over 128K

(c) 512K over 256K

(d) 1M over 512K

Figure 7.16: Relative EDP of DL2 Configurations with Realtime Matching - These figures depict the relative EDP of stepping through L2 cache sizes when they system is constrained by input rate to perform at realtime. The large improvements seen in previous evaluations are not found here, as the extra idle time of faster configurations trades off with the smaller active power dissipation of slower ones.

thus far. First off, we make no considerations at this point (beyond limiting our analysis to a 1MB cache), to overall area or processing cost, which will grow (possibly substantially) with increase cache sizes. Furthermore, with a 128K L2 cache, this architecture meets the realtime constraint for most configurations with greater than 4 processors. Our analysis, however, essentially assumes processing is done off–line, rather than at–realtime. If processing were to occur at the time the speech input is generated, then performance cannot exceed realtime. If the speech processor completes evaluation of a given input frame before the next frame is available, it may potentially have to wait until the speaker and front–end DSP processing provide the next input frame before continuing. In practice, one would likely buffer some quantity of input data and perform processing in batches, but in the baseline case, it is certainly viable to simply proceed one frame at a time, halting processor execution until the next frame is available.

In order to consider the performance implications of applying an actual realtime constraint to processing, we revisit our analysis assuming that the processor is placed into a low–power standby mode from the point at which it completes processing of a frame until the data for the subsequent frame is ready. The details of standby power consumption are detailed in Appendix B with the rest of our power analysis. We assume that it requires 50 cycles to bring the processor into a power–down state, and as we utilize architectural techniques such as drowsy caches and V_{DD} gating, we assume a further 50 cycles to come out of standby. Fortunately, the nature of this domain presents the opportunity to perform this powerdown without any performance loss, as the "time" at which the next frame of input data will become available is known well in advance. Thus, power up from standby can begin early enough to ensure that the processor is in a ready state before the next round of processing begins. In our power analysis, we assume that the system dissipates idle power while entering and exiting standby.

The relative EDP for each size stepped cache analysis assuming this fixed realtime constraint is presented in Figure 7.16. Note that these figures exclude smaller processor configurations that did not achieve realtime performance, and that the "relative perfor-

mance" metric is irrelevant under these realtime matching conditions. These figures depict a somewhat different picture than the off–line input assumption. While we still see some improvement in EDP when moving from a 64k cache to a 128k cache (note that the 64k cache is on the leading edge of the cache miss ratio "knee", while the 128k cache is on the trailing edge), the EDP tradeoff essentially brakes even for most of the other relative configurations. The tradeoff occurring here is not the energy reduction seen from reduced execution time in previous discussions. Rather, the added energy of larger caches, which dissipate more power even in standby mode than smaller caches, trades off with more time spent in standby on a per–frame basis. What this data suggests is that the increased time in a low–power state is a roughly even trade for the added resources within the range of our evaluation. Note that as we move from a 512K cache to a 1MB cache, we see the first signs of a tradeoff loss in the EDP.

Before moving on to extended analysis, it is important to make one final note. The L1 cache analysis is likely to remain fairly valid across speech recognition parameters (though it may be affected by dramatic shifts in the underlying knowledge base structure). This is because the L1 cache is primarily dealing with data the processing elements are actively utilizing. The performance improvements produced by the L2 cache, however, are related to both the application metadata size, and to the active knowledge base size. While the metadata aspect is likely to remain similar across similar programs, the program working set aspect is likely to vary substantially with knowledge base and speech recognizer configuration. For example, configuring the speech recognizer to operate a wider search will undoubtedly increase the size of the active working set of the program. By contrast, increasing the size of the knowledge base can either have the effect of increasing the working set due to added nodes, or decreasing the working set because the added knowledge base information allows for better pruning. As speech recognition is an evolving field, and as a thorough exploration of speech recognizer configurations is beyond the scope of this work, we conclude here only that use of an appropriately sized L2 can be very beneficial to speech recognition performance, but can not make a claim to specific cache configurations outside

of the speech recognizer parameters utilized in our experiments.

For much of our remaining analysis, we will adopt a 512K cache (unless there is specific reason to do otherwise). This represents a reasonable tradeoff between area, miss rates, the potential improved performance and energy dissipation demonstrated in our off–line analysis, and that seen in our realtime constrained analysis. While this is considerably larger than the cache miss knee (which would place us in the 128k range), we adopt the larger cache in recognition of the fact that increased accuracy demands are likely to lead not only to larger knowledge bases and a larger number of active elements, but also to a larger amount of metadata.

7.3.1 Stream L2 Bypassing

Thus far, we have considered the impact of treating all data identically with respect to the memory system. As the L2 cache attempts to capture the relevant working set of the program, it re–introduces the potential of exploiting stream characteristics of the program to improve performance. If portions of the program reference stream show extensive non–locality, then it may be beneficial to route these data reference around the L2 cache entirely, reducing potential pollution effects, and leaving more space for data with higher reference locality.

We explore the potential for such an optimization by selecting several data streams and bypassing their requests around the L2. We perform this analysis using configurations with 64k and 128k L2 caches, focusing on these smaller sizes because large L2s that capture more than the working set of the program may hide potential benefits from reduced pollution. Indeed, evaluations of larger caches show uniform relative performance losses, as the added requests to the memory system exacerbate the existing bottleneck.

We begin this stream analysis by considering the effects of bypassing run–time mutable data (predominantly linguistic channel data) around the L2. The relative performance of this modification on a system with a 128k L2 cache is shown in Figure 7.17. This figure depicts a number of interesting effects. First, note that a small performance improvement

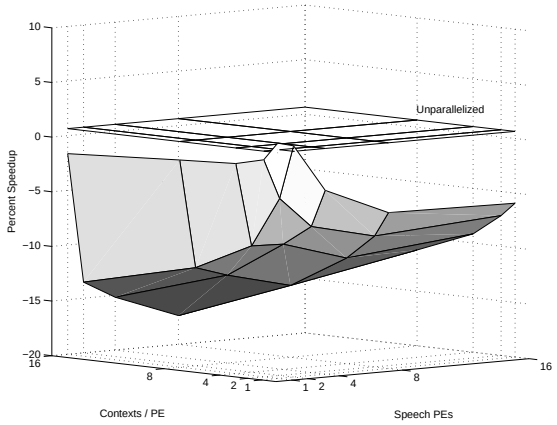


Figure 7.17: Relative Performance of 128K DL2 with channel (mutable) data bypassed

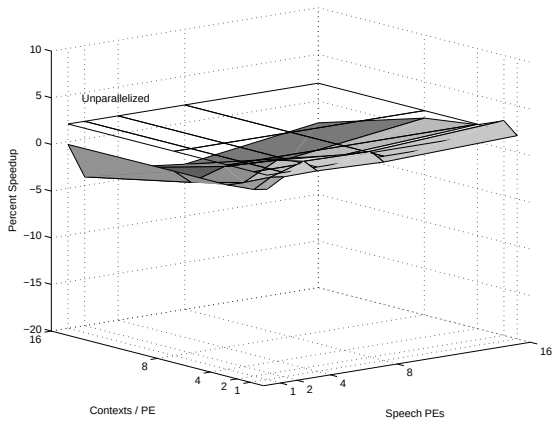


Figure 7.18: Relative Performance of 128K DL2 with SMD data bypassed

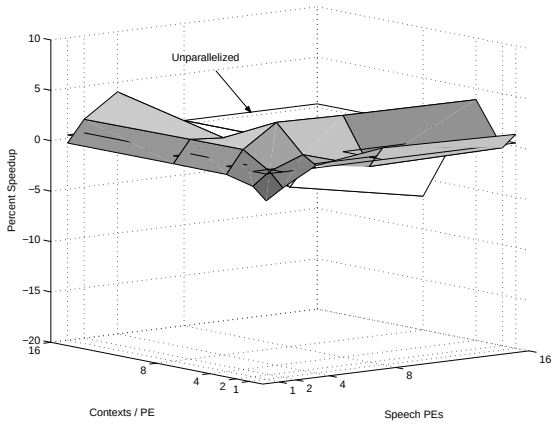


Figure 7.19: Relative Performance of 128K DL2 with PDF data bypassed

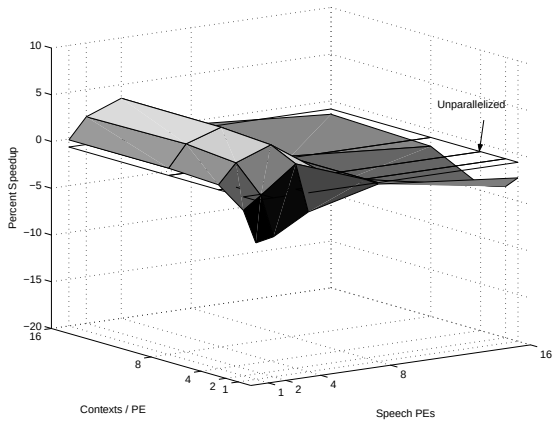


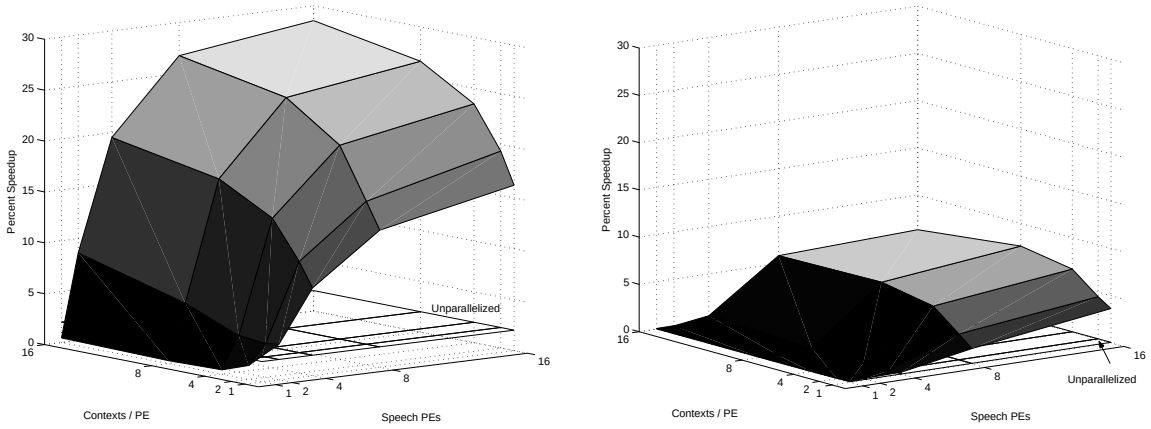
Figure 7.20: Relative Performance of 128K DL2 with LM and DICT data bypassed

is seen for the unparallelized case and for small parallelized configurations. This is reflective of the stream characteristics of the application data. Under these conditions, the number of cache misses in the base case is sufficiently high, and the memory bandwidth bottleneck sufficiently low, that the added latency of going through the L2 cache incurs a slight penalty. Furthermore, performance is compute limited in many of these configurations, and the added latency of a greater number of main memory accesses is not significant. Recognizing the margin of error of our simulation infrastructure, one could essentially consider this a break–even. As we increase the size of parallel configurations, we see a greater negative impact on performance. This effect is seen most clearly as we increase the number of contexts. Essentially, the incorporation of multiple contexts means a given processor is operating on several search nodes simultaneously, switching between them. This directly increases the amount of perceived locality, and takes better advantage of maintaining larger quantities of data closer to the processor. Put another way, a single context may operate on data in a stream oriented fashion, completing one node and moving to the next. The L1 cache is particularly effective at dealing with multiple references to this single data node, and the L2 has a minimal effect due to the reference characteristics. Given a sufficient number of contexts, the L1 cache is no longer sufficient to hold all of the actively processed node data, and from the viewpoint of the memory system it appears that the processor is repeatedly making references to the same data node with periods of accesses to other nodes between (as the processor switches between ready contexts). This creates locality effects among all simultaneously evaluated data elements. By contrast, we see a somewhat lesser performance loss as we increase the number of processors. More detailed inspection suggests that this is directly related to increased demands on the memory system made by parallel processing elements, and is the result of the memory bandwidth bottleneck.

Figures 7.18, 7.19, and 7.20 consider immutable data segments of the knowledge base, looking at static model data, Gaussian scoring data, and language model / dictionary data streaming respectively. In general, all of these experiments show minimal relative performance variations, and it is difficult to claim specific causes, as each modification

alters the overall dynamic schedule of the parallel processing system which itself introduces performance variations. As we increase the size of the system, we generally see improved performance due to the characteristics of the application until the bandwidth limits of the memory system are reached, at which point performance begins to fall off. The apparent ridges at four contexts or four processors in the Gaussian score stream and language model stream results are, as discussed previously, related to minor variations in the point at which memory bottleneck begins capping performance or reducing overall performance. Note also that the PDF and DICT streams constitute a fairly small fraction of the L2 misses for this cache size, contributing to their small effect. The fraction of SMD misses, however, is comparable to that of CHAN data. The minimal performance loss seen here suggests a potential candidate for bypassing.

The most important aspect of these results is in the lack of a significant performance impact. While there is a shift in memory requests from the L2 to the memory system, the added latency of these requests trades off more or less evenly with the improved hit rate for those requests that do go to the L2. At both larger *and* smaller cache sizes, we see greater performance losses for stream bypassing than depicted here. In the case of smaller caches such as a 64k configuration, the memory bandwidth bottleneck is reached for much sooner, and at larger cache sizes the working set effects of the L2 lead to better performance. In fact, the most relevant result of this analysis is that incorporating a larger cache leads to better performance, and in general a better energy delay product, than utilizing a smaller cache and attempting to exploit data stream bypassing to optimize cache utilization. As such, we do not consider stream bypassing in future evaluations. The fact that stream bypassing of certain segments of data leads to potential (if small) performance benefits, however, suggests that this effect should remain under consideration as speech recognizer knowledge base properties change.



(a) 1 port, pipelined over 1 port, unpipelined

(b) 2 port, pipelined over 1 port, pipelined

Figure 7.21: Relative Performance of Varied L2 Bandwidth - These figures consider the relative performance impact of constraining our base single port, 2–stage internal pipeline L2 cache to a single stage internal pipeline (a), and to a dual–ported cache structure (b). The reduction in bandwidth in (a) leads to a notable loss of performance. By contrast, the added bandwidth of a second port does not see a substantial performance increase. At this point, performance is constrained by request rate and the L2 bus more than the L2 latency itself.

7.3.2 L2 Bandwidth Constraints

As a final consideration of L2 characteristics, we wish to consider bandwidth effects between the L2 and the speech processing elements. Thus far, we have assumed that the L2 internal bandwidth is limited by a single access port capable of pipelining two requests. Figure 7.21 depicts the relative performance improvement of this base configuration over a non–pipelined configuration (a) and of a 2 port, pipelined configuration over the base configuration (b). As this figure depicts, constraining the bandwidth by eliminating pipelining would lead to a fairly considerable constraint on performance. By contrast, adding a full second port only has a notable effect with a larger number of processors. The bandwidth provided by the single port (with a small amount of internal pipelining to allow for more efficient processing) is sufficient for most configurations of this architecture. Note that moving to a full second port would lead to potentially substantial increases in energy consumption, which are not discussed here.

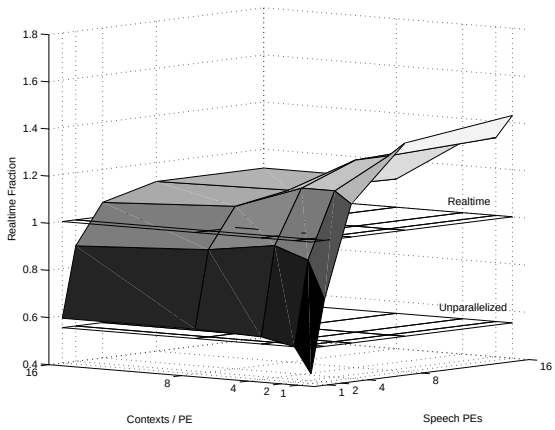
7.4 DDR Memory Systems

Thus far, we have focused our analysis of the on–chip components of the memory system, investigating performance improvement through cache arrangement and data manipulation. In keeping with the introduction to this chapter, we have been looking at ways of reducing the demand on memory bandwidth. We now begin considering approaches that alter the amount of available bandwidth. The first step in this analysis is to consider the impact of simply increasing the memory system frequency by moving to a DDR system and a 200MHz memory bus.

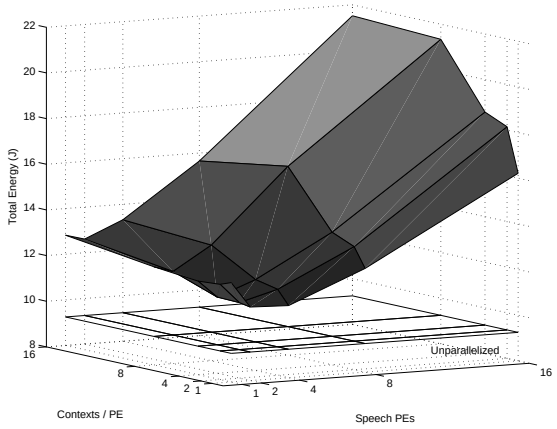
The performance / energy breakdown for a system with a DDR based memory is presented in Figure 7.22. As these values should be compared to the base SDRAM figures (with no L2 cache) depicted in Figure 7.4, the performance improvements are quite obvious. They are also fairly unsurprising, as doubling the memory system frequency essentially doubles the potential throughput.

It is important to note that there is a secondary effect here as well. Switching to a DDR model also increases the number of banks in the memory system, potentially improving throughput by increasing the number of open pages or the number of bank accesses that can be overlapped. We explore the effect of this modification by evaluating a normal SDRAM system with an increased number of banks, and by performing data placement optimizations to place all the data used exclusively by a single processing element in the same bank. Neither of these optimizations show a major effect, and generally result in a 4–5% performance improvement. This reinforces the understanding that the bottleneck in this application is the width of the pipe into the memory system, rather than conflicts within the memory system itself.

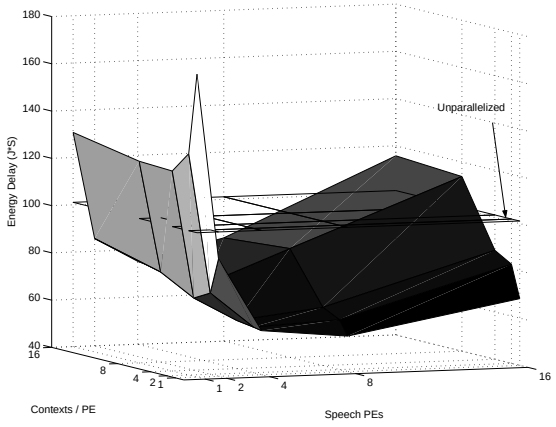
As stated, the fact that doubling the memory system data rate improves performance is unsurprising given the known bottlenecks in this application. A more interesting comparison is presented in Figure 7.23, which considers the relative performance improvement of a system with a 100MHz SDRAM and a 512k L2 cache over a system with a DDR based memory system and no L2 cache. The relative performance analysis of these configurations



(a) Realtime Fraction

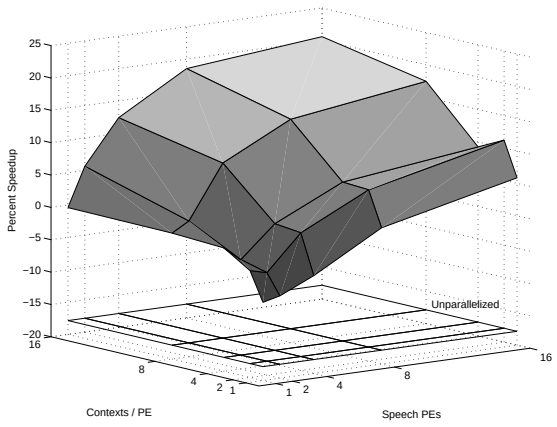


(b) Energy

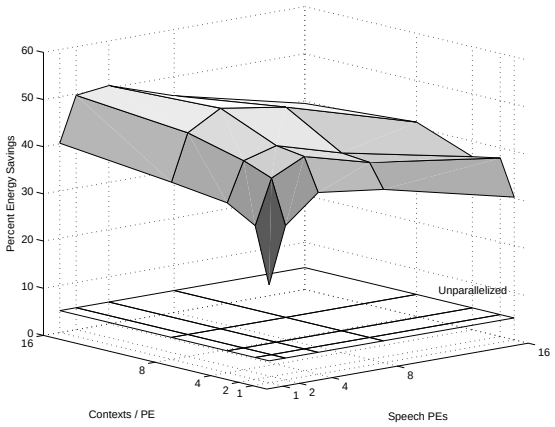


(c) EDP

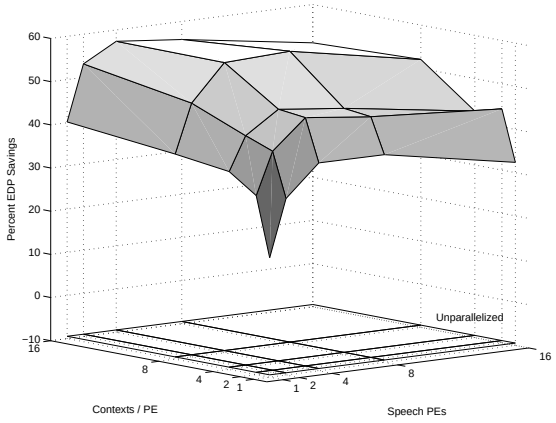
Figure 7.22: Performance Breakdown of base DDR (200MHz) system - This figure depicts the performance, energy, and EDP of a system with a 2.4V Micron Technologies 200MHz DDR SDRAM (or comparable configuration).



(a) Relative Performance



(b) Relative Energy Savings



(c) Relative EDP Reduction

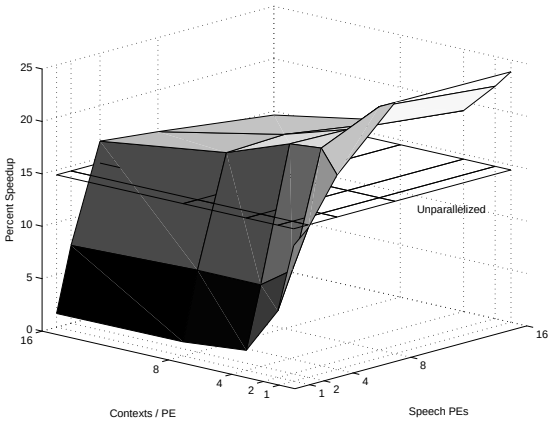
Figure 7.23: Relative Performance Breakdown of DDR vs. DL2 - These figures consider the relative performance, energy, and EDP of a 512K L2, SDRAM based system over a no-L2 DDR based system. It is clear that the large L2 is a far better approach to gaining performance than the higher speed, less energy efficient RAM.

shows that the two configurations essentially break even for small parallel configurations. A reasonable performance improvement is seen for very large configurations (owing substantially to the "L1 Backing" effect previously discussed), while the unparallelized configuration (which does not see efficient utilization of the L2, but does take advantage of the overall lower latency to memory in the DDR case) sees a relative performance loss. The important evaluations here, however, are the relative energy and EDP breakdowns showing a marked energy reduction for the L2 configuration. Overall, these results indicate that a large L2 cache can achieve comparable performance to a DDR memory with much higher power efficiency. Much of this comes from the fact that DRAM systems are inherently leaky devices, and the DDR model used in our evaluations runs at a considerably higher supply voltage level than the low–power SDRAM system model used (note that we selected parameters for DRAM systems with the lowest possible energy dissipation in both cases). The added energy dissipation of off chip communication is also a factor. Note that, as our cache energy estimation framework is likely to over–estimate the actual energy dissipation of large cache structures, the energy efficiency of the L2 configuration is likely even higher than we depict in this analysis.

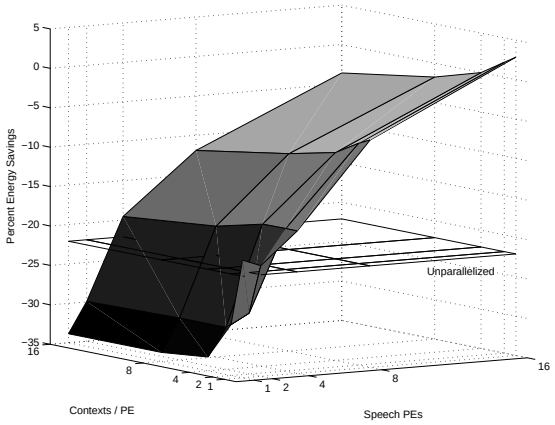
This data strongly suggests that there is little benefit to moving to faster but higher energy consuming memory systems. We attempt to verify this conclusion by considering the relative performance and energy of a SDRAM system with a 512K L2 cache with that of a DDR based system with a 512k L2 cache in Figure 7.24. While a performance improvement is once again apparent in these results, there is a substantial increase in energy dissipation, and the resulting relative energy delay product comparison breaks even at best, and shows considerably lower efficiency at worst. Based on the data presented here, we will focus on configurations with SDRAM based memory systems for the remainder of our evaluation.

7.5 Data Stream Partitioning

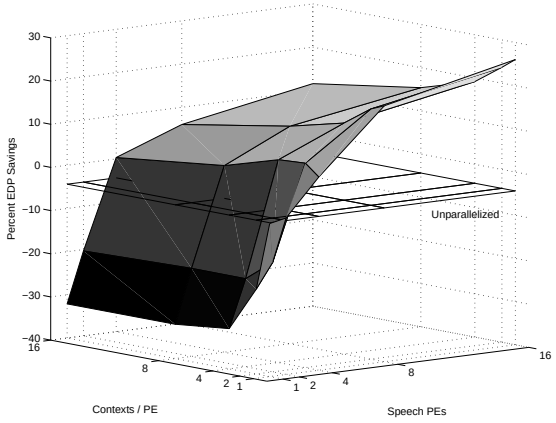
Analysis of cache structures to reduce demand on memory bandwidth considered the potential to take advantage of properties of the application data stream to improve bandwidth



(a) Relative Performance



(b) Relative Energy Savings



(c) Relative EDP Reduction

Figure 7.24: Relative Performance Breakdown of SDRAM+DL2 vs. DDR+DL2 - These figured depict the relative performance, energy, and EDP of a DDR based DL2 system over a SDRAM based DL2 system. While there is a performance improvement, the cost is seen in the energy dissipation of the DDR dram, leading to a generally negative EDP.

utilization. We now consider taking advantage of properties of the application data stream to reduce system energy consumption and provide more system bandwidth. More precisely, we note that substantial portions of the knowledge base utilized is speech recognition are treated as immutable during runtime execution. Acoustic and linguistic data may require occasional updates as more accurate models or larger vocabularies become available, but during recognition itself, only the current search tree status need be writable. While the precise size of this writable data varies with respect to the vocabulary and knowledge base size, it is generally only a small fraction of the total application knowledge base. This opens the possibility of placing immutable knowledge base data in memory structures such as flash or ROM chips, which are far more power efficient than standard DRAM technologies. We evaluate the potential use of each of these components, both as part of the standard memory system, and as chips placed within the same package (allowing for potentially greater bandwidth) as the processor core itself. Based on the data presented thus far, we assume a base configuration of a 100MHz SDRAM memory system (for mutable data) and a 512K L2 on–chip cache. Note that our previous results have strongly suggested that 16 processor / 16 contexts configurations are well past the positive tradeoff point for most systems with realistic memory systems. As such, we narrow our analysis to a maximum of 8 processors and 8 contexts for much of the remainder of our evaluation.

7.5.1 Flash

Flash memory is much more power efficient, but also notably slower than corresponding DRAM technology. In order to explore the possibility of utilizing flash based memory, it is necessary to develop a reasonable timing model for flash data accesses. We develop such timing estimates using information from Micron Technologies page mode flash datasheets [4]. Based on this information, we establish a virtual flash memory component consisting of four page mode flash chips connected to a common bus and interfaced to the rest of the system through a controller (which could be the normal memory controller). The controller is responsible for managing and coordinating flash accesses, and for interfacing the 16–bit

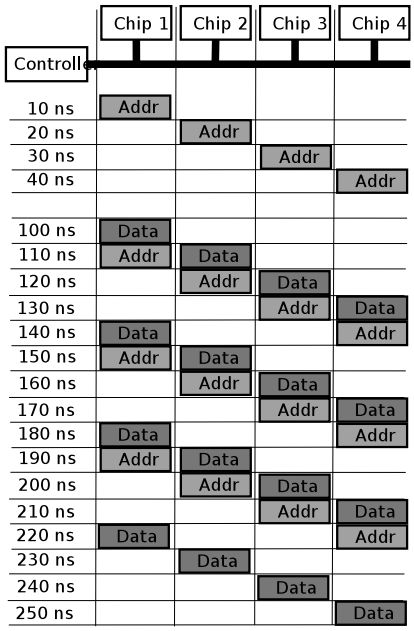
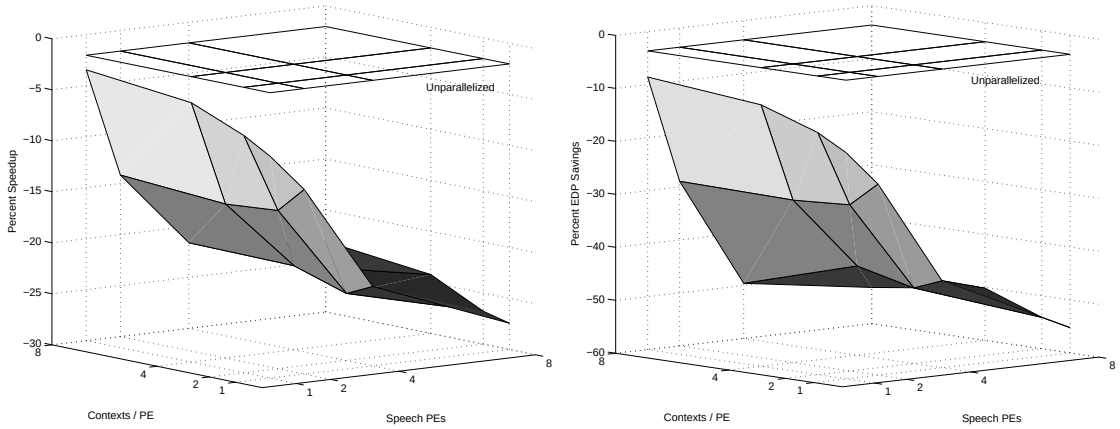


Figure 7.25: Flash Model Timing Example - This figure depicts a hypothetical cache line read from our base cache model. Note that, based on flash chip pin data, we assume separate address and data buses, and further assume that each data transfer constitutes 16 bits of data.

data output from each flash chip with the 64–bit memory system interface. Each chip operates at 1.7V, and requires 90ns between receiving an address to a new page and returning the first data from a page. Subsequent page mode accesses return data at 30ns intervals. Note that this represents a low–power (longer latency) operation mode for the flash chip. As each data transfer from the flash chip to the controller transfers two bytes of data, a full page read transfers 8 bytes. Based on our four chip assumption, transfer of a full 32–byte cache line would require a full page read from each chip. We assume that the transfers involved may be overlapped, time–multiplexing bus utilization and using delayed address propagation to synchronize address and data return times from each chip. An overview of a hypothetical 32–byte cache line read (assuming a 100MHz bus) into the controller is depicted in Figure 7.25. Note that, following the pin configurations of an individual flash chip, we assume separate address and data buses. One could also envision a single chip model, in which data is transferred through multiple reads to the same flash cell. We adopt a four chip model because a single chip configuration (requiring four sequential page mode reads



(a) Relative Performance

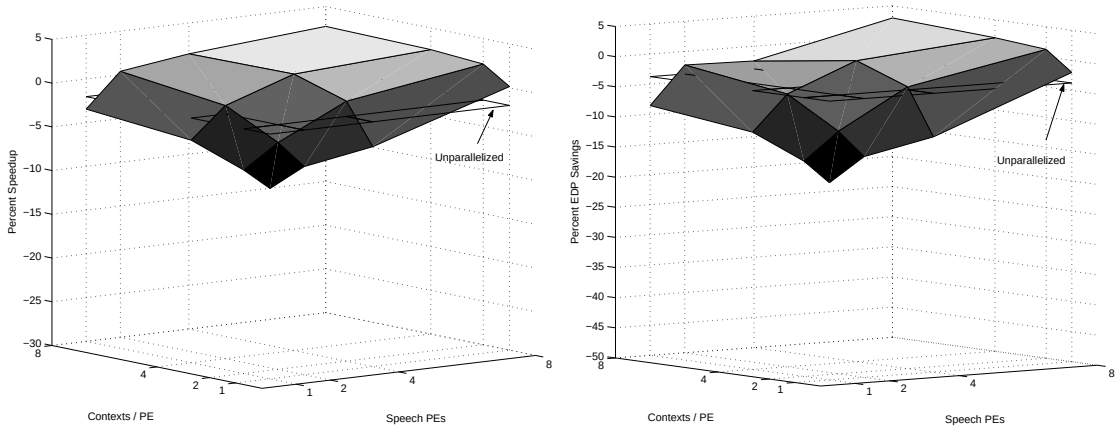
(b) Relative EDP

Figure 7.26: Relative Performance and EDP of Base System vs. Flash - This figure depicts the relative performance and EDP of a single flash module, unbanked system over the base system with no flash. The longer access latency to the single flash bank and resource bottlenecks lead to considerable performance loss and, due to also to added background energy dissipation, equally comparable EDP increases.

from a single chip) or a comparable double chip solution are prohibitive in latency. Note, however, that we are utilizing this model to extract timing estimates. In a real system, one could envision all such data maintained on a single, multi–banked flash chip, or some other similar configuration.

Based on this figure, we determine that the critical word in the memory request is available approximately 140ns after the request begins processing. This corresponds to approximately 56 processor cycles (at 400MHz). We make the assumption that the critical word needed to service a request is returned first (that is, the hypothetical sequence of accesses shown in the figure could begin at any chip), but that the flash component can only accept another request after the full cache line has been read (assuming overlap with the next request, this occurs at around 230ns). Note that these latencies refer to delays in the flash component of the memory system itself, and that the actual latency seen by the processor for a given request also includes latencies incurred in allocating the memory bus to both transmit the request and receive the results.

The relative performance and EDP improvements of a base 4–chip Flash system with the



(a) Relative Performance

(b) Relative EDP

Figure 7.27: Relative Performance and EDP of Base System vs. Banked Flash - These depictions are identical to the previous figure except that we consider a banked flash component, allowing for added bandwidth. This shows that the primary source of performance loss in the single flash bank example was resource bottleneck, not the added latency to complete a given request. This suggests the need for a multi-banked static memory storage system that can overlap accesses.

flash modules located alongside memory modules (that is, out in the memory space) over a base system with straight SDRAM memory is depicted in Figure 7.26. The key observation here is the overall relative loss of performance. This is directly tied to the increased latency of flash accesses relative to the original DRAM accesses, and the increased contention for the single access point for this data (following the latency model between accesses as previously described). The tradeoff here is the reduction in access and idle energy of a flash chip relative to a DRAM chip countered by the added overall energy dissipation due to reduced performance. Based on our energy estimation model, this does not appear to be a tradeoff win.

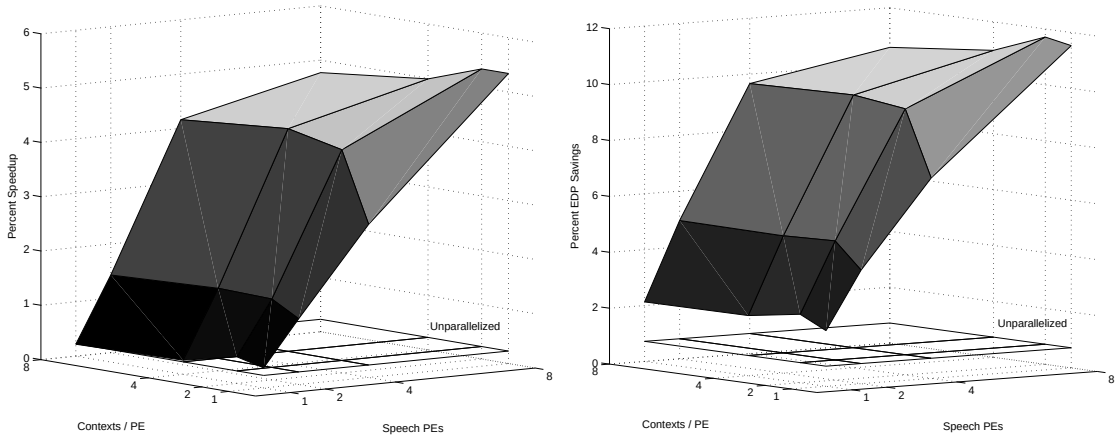
Multi–Banked Design

The previous evaluation considered what amounts to a single–banked flash system. This leads not only to a latency constraint, but a bandwidth constraint as well, since only one request to static data can be in progress at a given time. A straightforward solution to this would be to consider a multi-banked flash system in which lookups can be performed

simultaneously on either bank, though data passed back to the processor is serialized by the communication network. Such a configuration is considered in Figure 7.27, which evaluates relative performance of a flash model with two banks, selected by a low order address bit such that contiguous cache lines alternate in bank location. Here we see not only a performance improvement, but also see an energy improvement due at least in part to the reduced energy dissipation of the flash device. The results shown here strongly suggest that the poor tradeoff seen in the single bank flash results are directly related to latency and bandwidth bottleneck issues within the flash, and not due to overall memory system bandwidth constraints or lack of energy savings from the flash device. Note, however, that these benefits are only really seen once sufficient concurrency resources are added to the processor to begin placing constraints on the memory system. Until that point, the effect of extra latency on individual requests predominates.

On–Package

One further interesting option for data arrangement when partitioning out the immutable knowledge base data is the possibility of utilizing a large on–package multi–banked flash. While one could also conceive of a on–die memory space, this seems impractical for the static knowledge base data due to sheer size. An on–package solution, however, does not run into the same technology and size issues, and may be more feasible. At the resolutional accuracy of our simulation infrastructure and power modeling environment, the effect of moving the flash on–package is seen in two ways. First, from a performance standpoint, we assume that as a result of its location, it does not operate on the same memory bus as the main memory system. This results in some amount of added overall bandwidth into the processor, as access to immutable data may occur in parallel with memory system accesses. Flash access still requires the same amount of time, however, so we continue to assume the equivalent of a 100MHz communication rate across the wirebonds between chips. Secondly, as off-chip communication is no longer necessary, the energy cost of accessing the flash drops. The methodology by which we account for this effect is detailed in Appendix B.



(a) Relative Performance

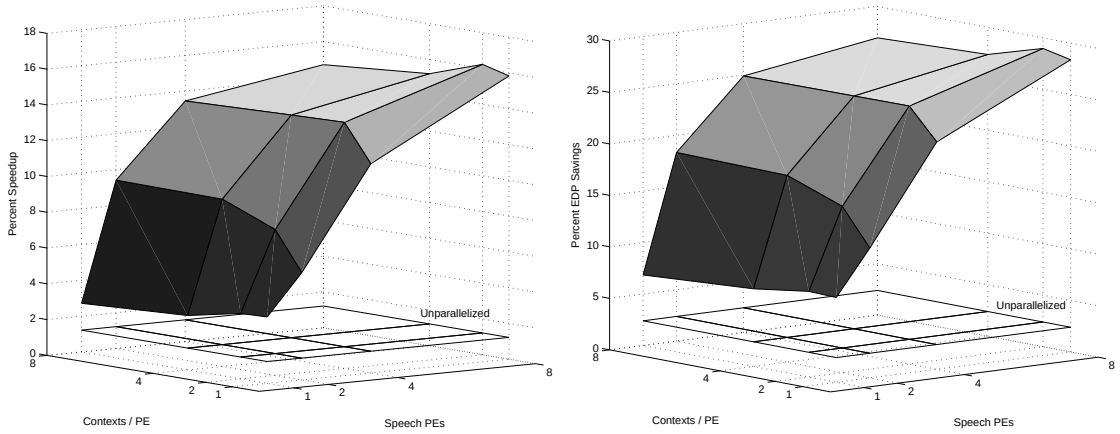
(b) Relative EDP

Figure 7.28: Relative Performance and EDP On-Package Flash Over Off–Package - These figures consider the relative performance and EDP of an on–package flash component over an off–package (in the memory system) flash component. The major performance factor here is the added bandwidth to static knowledge base data, while the major energy component is the reduced capacitative load on accesses.

The relative performance of an on–package, dual–banked flash system over an off–package system is shown in Figure 7.28. These figures clearly demonstrate the potential benefit of higher bandwidth access to static knowledge base data, however they do not say anything with respect to the cost of providing such functionality.

7.5.2 ROM

One final memory system configuration that is viable in this application domain is the use of a ROM chip to maintain static knowledge base data. ROM chips have the marked advantage of fairly low energy dissipation, and extremely high density. The obvious downside is the inability to perform field–updates to the knowledge base data. As such, a ROM knowledge bases are most likely to be found as plug–in modules that may be replaced as updated data is available. This is, however, in itself a very reasonable approach. It is important to note, however, that many applications of handheld speech recognition may desire speaker dependent training. Depending on the techniques used to perform such training, this may make ROM based static knowledge bases infeasible.



(a) Relative Performance

(b) Relative EDP

Figure 7.29: Relative Performance and EDP of Off–Package ROM over Flash - These figures consider the relative performance and EDP of a system with an off–chip ROM based static memory system with SDRAM timings over the base flash system. We consider energy impacts without timing effects in the next two figures.

As ROM chips are particularly domain and use specific in nature, accurate timing information for a ROM chip needed in this environment is unavailable. In order to use reasonable values, we note that the ROM system design we consider is conceptually very similar to a DRAM array. As such, we utilize DRAM delay information in our ROM analysis. This results is an approximate 50 processor cycle latency for a single uncontested ROM request. The primary consideration here, however, is potential energy savings.

We begin by considering the effects of replacing the flash component in our previous analysis with a ROM component. The relative performance and EDP of such a replacement is depicted in Figure 7.29 and shows a fairly substantial benefit to this configuration variant. These figures are somewhat misleading, however, because we are once again varying both the energy model and the timing model. In order to better explore the underlying variables in this modification, we consider two more specialized situations. Figure 7.30 depicts the relative EDP of the a single bank, off–chip ROM system over the base SDRAM system. As we are utilizing latency values similar to the SDRAM system for the ROM, this figure essentially depicts the overall EDP benefit for a similar performance characteristic (in practice, we find that a single serial request servicing ROM using SDRAM latency actually performs

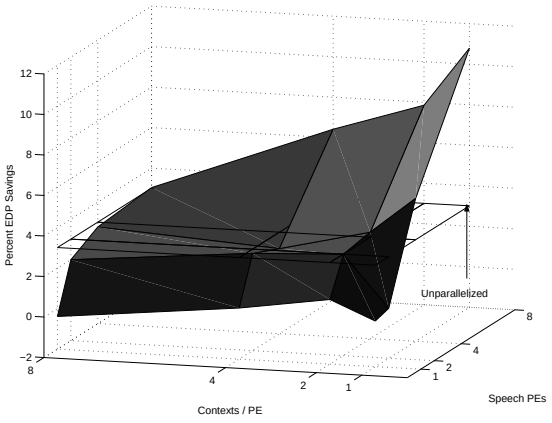


Figure 7.30: Relative EDP of ROM over Base System - This figure considers the EDP of a ROM based static memory component with SDRAM timings (thus the performance is the same). Note that we observe a small EDP improvement. In a real system, however, the use of a ROM may allow a reduction in the total amount of DRAM included, leading to a substantially larger EDP improvement. That degree of resolution, however, is beyond the capabilities of our energy estimation infrastructure.

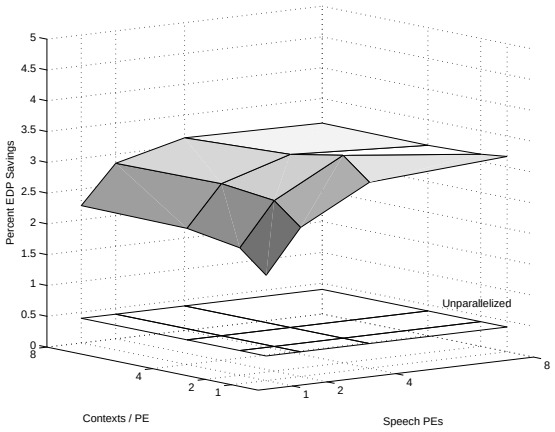


Figure 7.31: Relative EDP of ROM over Flash with Identical Latency - This figure depicts the relative EDP improvement of a ROM based static memory system over a flash based system when the ROM is constrained by the slower flash timing model. We see a slight improvement due to the better energy efficiency of the ROM device.

slightly worse than the base SDRAM system). Note that, as seen in other evaluations, the relative benefits become more substantial only as the memory system is stressed. In this case, the benefits are most obvious when many processors are competing for memory system resources with no added contexts to reclaim some of the lost bottleneck performance. By contrast, Figure 7.31 considers the EDP of a off-chip ROM system with flash latency levels. This more directly depicts the energy benefit of utilizing the ROM system over a flash based system. In all of these cases, we see a slight EDP improvement from switching to the ROM system, as one would expect. Note, however, that in our evaluation methodology this value is fairly small even compared to a SDRAM system. This is due to a number of factors. Firstly, the energy savings in our evaluation framework really comes from explicit accesses to the ROM versus the SDRAM system, which really only affects reads to static knowledge base data. In a real system, one would expect this relative energy improvement to be some-

what greater (even in the previous flash analysis), because as knowledge base sizes increase, use of ROM and flash technology correspondingly reduce the *amount* of DRAM needed by the system. The resolving ability of our energy estimation framework does not account significantly for potential reductions in DRAM quantity (as this will also be determined by the needs of other software running on the system). Secondly, recognize again that the SDRAM model utilized in our energy computations is itself fairly low power. It is quite possible that a comparable ROM system could be designed to operate at considerably lower power than our energy estimation framework accounts for.

On–Package

As with the flash model, we once again consider the implications of placing ROM data on package with the processor core. This leads to basically the same variations in simulation infrastructure as previously discussed for the on–package flash model, and results in fairly similar performance variations. Thus, it tracks what one would expect for a similar organizational change. In practice, however, an on–package solution is much more feasible for a flash, which allows the possibility of updates. A ROM based system would most likely be found as an off–package plug in module which could be replaced as needed to update knowledge base information. Thus, while we do see an overall performance and EDP improvement, we do not consider this a truly viable memory organization.

7.6 Embedded DRAM Configurations

As a final memory system configuration, we consider the potential of utilizing system–on–chip capabilities (specifically Embedded DRAM technology) to place fairly substantial quantities of DRAM on–chip with the processor core. Embedded DRAM technologies have the advantage of much higher density as compared to SRAM based on-die memory technologies, but also generally demonstrate a higher access latency [44, 5]. This added latency, however, is far lower than access to off chip memory, and large quantities of such memory can be quite useful in many application domains. One obvious potential use of embedded

DRAM technology would be to replace the previously discussed L2 cache with an embedded DRAM array. Density analysis suggests that such a change could allow 5–8x the amount of memory to be fit into the same area of the chip [44]. Given our extensive L2 cache analysis, however, it is clear that such an optimization will have its uses, and may be particularly vital to performance as knowledge base sizes increase. Rather than reiterate an already explored area, we consider in this section a more drastic variation in the memory hierarchy.

In the previous section, we considered the implications of isolating immutable knowledge base references, utilizing the read–only nature of this data to perform energy saving optimizations. Here, we consider the other side of the coin, and evaluate the potential of moving partitioned mutable data into a set of on–die embedded DRAMs. In this configuration, each processing element has direct and exclusive access to a single embedded DRAM module. Mutable knowledge base data (that is, search tree channel and scoring data), is loaded into this embedded DRAM space on a per–partition basis. Data that must be shared across processing elements is left in the existing SDRAM based memory system. For the purposes of this evaluation, we will begin by considering a system in which such partitioned mutable data is stored locally, but all other data is fed directly to the memory system with no intervening L2 cache. Maintaining our analysis from the previous section, we assume that static knowledge base data is stored in an off–chip, two bank ROM module.

In order to model this embedded DRAM space, we recognize that from the viewpoint of a single processing element, this space appears to be a small, direct–mapped cache (representing the current set of open pages), with a fixed cache miss latency (representing the time to close the current page, and open a new page). Thus, for our performance evaluations, we utilize internal configuration information for an example 64 MB SDRAM chip from Micron Technologies [4] and represent the open page cache as a four sets (representing banks), direct mapped, with 256 byte lines (representing column width). We assume a 10ns (4 processor cycles @ 400 MHz) latency for a page hit, accounting for address decoding and page matching time. We further assume that the embedded DRAM can handle one outstanding "page miss" at a time, and servicing such a miss requires a fixed latency of

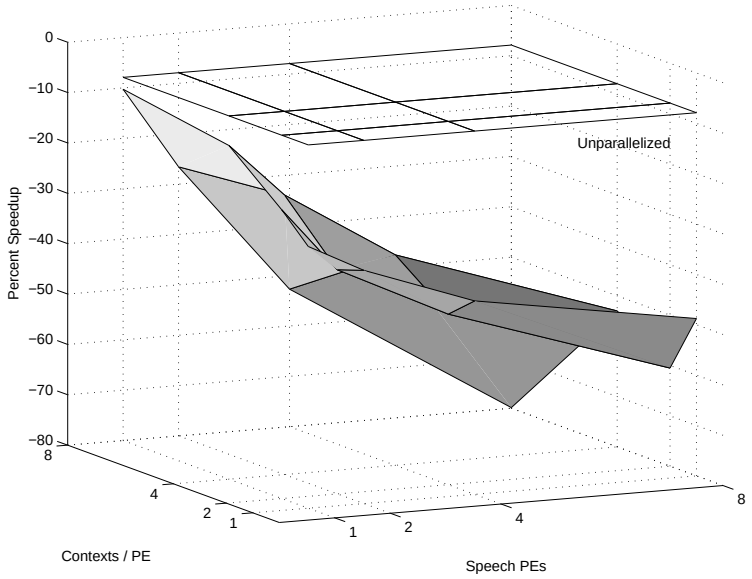


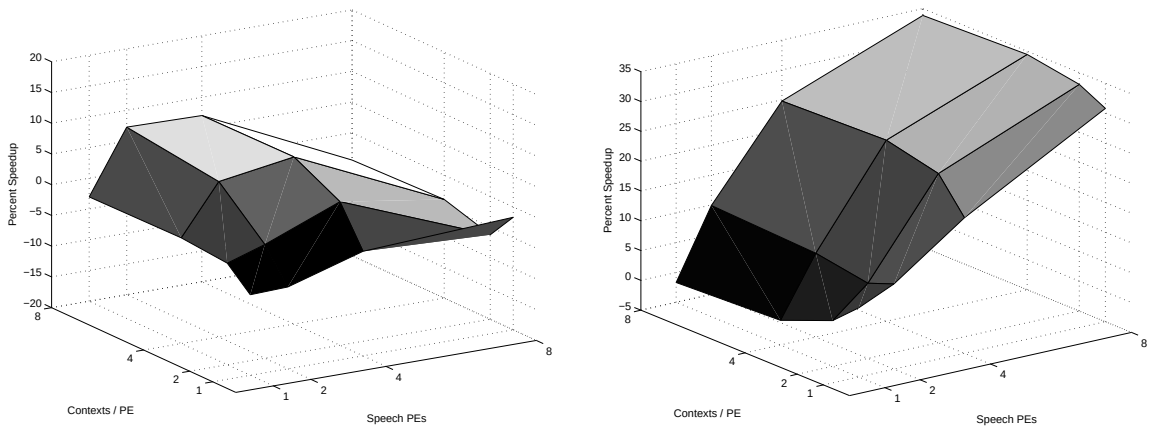
Figure 7.32: Relative Performance of Base EDRAM System - This figure depicts the relative performance of a system with approximately 2MB of embedded DRAM on chip for mutable knowledge base data relative to the base system with a 128K L2 cache. Much of the performance loss is due to bypassing of the L1 cache for data found in the EDRAM, and lack of fast access to shared metadata and some static knowledge base data.

40ns (accounting for page close, array precharge, and new page opening operations). We conservatively estimate the energy dissipation of each embedded DRAM as equivalent to a cache attached to a small standard SDRAM chip. That is, in reference to the power models discussed in Appendix B, each embedded DRAM dissipates idle energy of a appropriately sized cache and a small SDRAM cell. An open page hit dissipates the added energy of a cache access, and a page miss dissipates the energy of a cache access and a active DRAM cycle. It should be noted that embedded DRAM tends to dissipate less power than comparable standard DRAM systems. Furthermore, our model cannot fully account for the energy dissipation equivalent of a set of small DRAMS, as we are using energy consumption data from somewhat larger DRAM devices (equivalently detailed energy characteristics for embedded DRAMS was not available). Thus, our energy estimates are likely to once again be conservatively high. Given this growing incongruity with information available for our energy estimation infrastructure, we will focus primarily on performance issues here.

Figure 7.32 considers the relative performance of a system utilizing approximately 2MB

of embedded DRAM to capture the partitioned mutable knowledge base data used in speech recognition over a standard configuration with a 128k L2 cache. To place this evaluation in perspective, relative density analysis based on data in cited work [44] suggests that 2MB of embedded DRAM would be roughly comparable in size to a 300–350K SRAM cache. Note that the embedded DRAM system in this evaluation includes no L2 cache. Despite the presence of the embedded DRAM, this system suffers from a substantial performance loss relative to the raw L2 system. This is really a result of a number of factors. First, our model directs data references to the embedded DRAM directly to the DRAM component, bypassing even the L1 cache. This added latency at the first level of the memory hierarchy contributes significantly to the performance loss. Secondly, certain segments of program metadata are shared across processing elements (for example, the result array of Gaussian Scoring). In the L2 configuration, this is among the metadata captured by the L2 cache. In the embedded DRAM configuration, this shared data must be accessed from the DRAM. Secondly, note from earlier miss analysis that along with program metadata, the L2 data stream consists of considerable quantities of acoustic and static linguistic data that go along with the mutable search tree data. This information too must be accessed from main memory. These factors together also have the added effect of exacerbating the bottleneck at the memory system interface, as seen in many previous evaluations.

The true effects of the embedded DRAM is better seen when we include a L2 cache to deal with the extra metadata. Figure 7.33a shows that including a fairly small (16K) L2 cache between the L1 and the memory system recovers the performance loss seen in the previous figure. In fact, as Figure 7.33b shows, incorporating the same 128K L2 cache as the base system we are comparing to demonstrates a substantial performance improvement. It should be noted that, except for the effect of reduced execution time due to improved performance, all of our embedded DRAM evaluations were less energy efficient than standard L2 systems (an example shown in Figure 7.34 for a 16K cache comparison). We do not explore this in further detail as we believe a precise accounting of the energy consumption of an embedded DRAM system is beyond the accuracy of our energy estimation framework. One



(a) 16K L2 Cache

(b) 128K L2 Cache

Figure 7.33: Relative Performance of EDRAM with L2 - This figure depicts the relative performance of embedded DRAM systems with a 16K (a) and 128k (b) L2 cache to the standard memory system over a standard system with a 128K DL2. Note that a much smaller L2 cache is sufficient to recover the performance loss seen in the previous figure, and that a larger cache leads to significant performance improvements

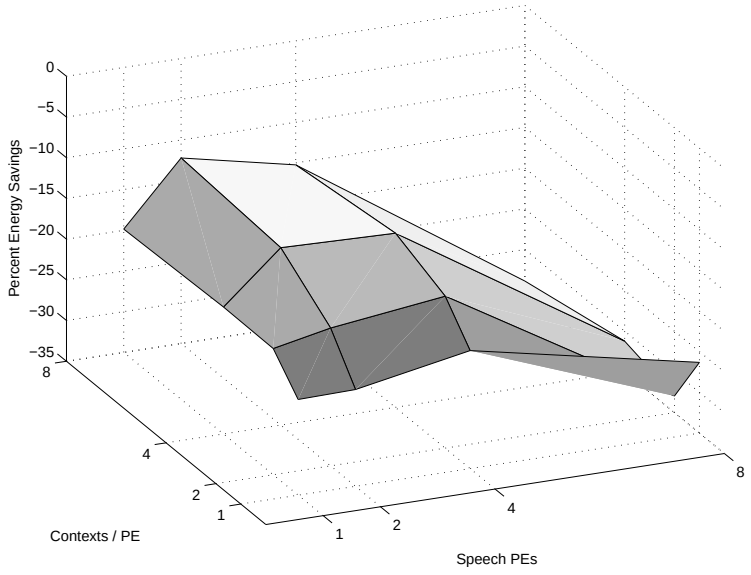


Figure 7.34: Relative Energy of Base EDRAM System - This figure depicts the relative energy of a 16K L2 embedded DRAM system versus a standard system with a 128K L2 cache. The significant loss of energy efficiency is (in our energy estimation model) due to a greater number of accesses to the DRAM component, and the energy consumption of that component.

could likely conclude, however, that due to the inherent leakiness of DRAM components, the energy efficiency is likely less than a carefully engineered SRAM cache.

The performance improvements depicted here suggest that embedded DRAM may be a viable approach to improve speech recognition or stochastic search performance on this style of architecture. It suffers from a number of constraints, however, which lead us to focus on more standard systems for the remainder of this work. First, placing considerable amounts of embedded DRAM on chip can substantially increase chip size and correspondingly fabrication and production costs. As existing systems such as the processor on the Nintendo Gamecube incorporate more embedded DRAM (8MB) than we are considering here, this tradeoff may or may not be overly costly, and must be evaluated in more of a production environment than the scope of this work allows. Secondly, we assume that all of the mutable knowledge base data of the application is loaded into this embedded DRAM space for low latency access. The size of this data, however, may vary considerably with speech recognizer and knowledge base characteristics. In general, much of the growth in knowledge base size in advanced speech recognizers is found in the static knowledge base (better initial training and modeling information), but a more thorough accounting would need to be performed before a preferred embedded DRAM size can be identified.

7.7 Conclusions

In this chapter, we present a detailed memory system design space evaluation on our low–power parallel architecture. We determine that, due to program flow characteristics, ensuring fast access to relevant portions of the instruction stream is a fairly trivial problem. Based on our multi–processor cache miss analysis, we find that a pre-loaded 1K cache for every 2 processing elements, and a 8 instruction buffer (one cache line) on each processor is sufficient to virtually eliminate stalls due to instruction fetches.

Exploration of application data characteristics on our architecture reveal that a small L1 cache is very useful for providing fast access to data currently under evaluation by a processing element, and some immediately needed metadata. We arrive at the use of

a 2K L1 cache per processor, and find that it predominantly maintains metadata, and operates most efficiently when shared across all thread contexts. We further find that a shared L2 cache targeted in size to capture program metadata and knowledge base data currently under evaluation can have a significant positive effect on performance. This result is particularly important because program metadata size is likely to grow at a *far* slower rate than program knowledge base data, and knowledge base data currently under evaluation is tied to the computation resources of the processor, not the knowledge base itself.

As we begin evaluating off–chip components of the memory system, we determine that DDR based RAM systems can improve performance, but not as much, or with as high of a power efficiency as a large L2 cache. We also explore the potential use of flash and ROM based memories for storing the static knowledge base, determining that some potential for energy reduction through use of such components exists.

The result of these memory system constraints affect the performance of concurrent execution as well. We find that performance improves dramatically to 4–8 processors, and then essentially levels off. A similar characteristic is seen in terms of the number of available contexts. Essentially, once the available memory bandwidth has been filled, added parallel resources only create added bottleneck and expose underlying overheads.

Considering all of the results presented here, we propose a configuration of 4–8 processors with 2–8 contexts per processor. We incorporate a 1K instruction cache per two processors, 2K data cache per processor, and generally utilize a 128K to 512K L2 cache in future evaluations (recognizing that wider or more extensive searches may lead to added metadata). We settle on a low–power 100MHz SDRAM memory system, and believe static knowledge base data will be placed in an external, replaceable, multi–banked ROM component. We will use versions of this basic architecture model with slight variations as appropriate in extended evaluations in the next chapter.

Finally, it is important once again to caveat this analysis on the fact that speech recognition is an evolving domain. The primary intuition presented in this chapter should not be thought of as a memory architecture that will support speech recognition in all future

incarnations, but rather as directions in which to focus future efforts as needed. For example, we determine that memory bandwidth is the primary bottleneck to performance on this architectural model. The demand for bandwidth will grow with larger knowledge bases and higher accuracy recognition systems. Techniques such as partitioned knowledge base data with a separate high bandwidth interconnect to the processor will become more important as speech recognition evolves along these lines. This does not eliminate the benefits of caching program metadata or invalidate the general reference patterns seen on this analysis. By recognizing the source of future difficulties and the applicability of current solutions, efforts can be focused on addressing problems more directly.

CHAPTER 8

Extended Optimizations

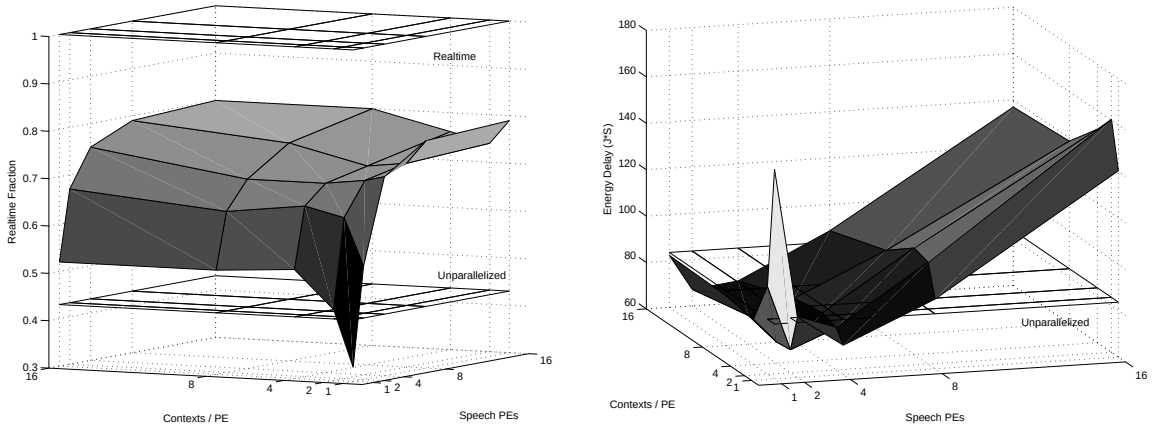
In previous chapters, we have presented a low–power parallel architecture optimized to operate efficiently on stochastic search style operations, with the goal of achieving low–power real–time continuous speech recognition. We have evaluated concurrency and operational performance of the architecture, and explored an array of potential memory system configurations, arriving finally at an image of a system with a large on–chip L2 cache designed to capture program metadata, and static knowledge base data of the application stored in an off chip ROM component (presumably a plug–in module). This chapter will build upon the analysis and conclusions presented thus far. We first consider potential solutions to the memory induced performance loss seen for systems with large numbers of contexts on constrained memory systems. Specifically, we evaluate the potential of exploiting intrinsic characteristics of our programming model to dynamically match exposed concurrency to system bottlenecks. Second, we evaluate the potential of improving L2 cache utilization through data compression. Finally, we consider issues of runtime power management. While we have discussed energy dissipation throughout our analysis, we here consider potential runtime enhancements designed to trade off excess performance for energy savings.

8.1 Concurrency Throttling

Recall the curious result seen in many earlier memory system evaluations of large numbers of contexts resulting in decreased performance (relative to smaller numbers of contexts). Examples are seen in the performance of the base SDRAM system (Figure 7.4 on page 143) as well as the base SDRAM system with an L2 cache (Figure 7.9 on page 149). As discussed in those performance evaluations, these performance losses are somewhat curious in that added contexts in ideal memory system simulations served to tolerate added latency. They can be explained, however, by recognizing the bandwidth limitations of real memory systems, and noting that as memory bandwidth becomes a bottleneck, the perceived latency of a given cache miss increases. In essence, what these performance losses show are the overhead of managing a larger number of active (but stalled or repeatedly retrying memory requests) processor contexts. Under more relaxed conditions, this overhead is more than made up for by improved processor utilization.

Clearly, exposing concurrency to the point where management overhead is exposed due to other system bottlenecks is not ideal. Even in configurations that exceed this bandwidth limit, however, there may be opportunities to improve performance if extra contexts are available. Thus, we need to evaluate ways to maintain the peak performance point regardless of the number of available hardware resources.

Fortunately, a system for manipulating exposed concurrency based on system level constraint decisions is an inherent aspect of our architectural model (though we have not explored its uses until now). Recall that, in keeping with the philosophy of the programmer exposing all available concurrency and allowing the architecture to match utilized concurrency to available resources, our programming model utilizes standard function call semantics for thread spawn requests. Thus far, we have assumed that the only limit on thread creation is the availability of physical storage resources (hardware thread contexts). We now exploit some of the power of this programming model by placing a bottleneck driven constraint on the creation of new threads. Specifically, we convert all thread spawn instructions to function calls if the memory queue is already full. This ties the number actively



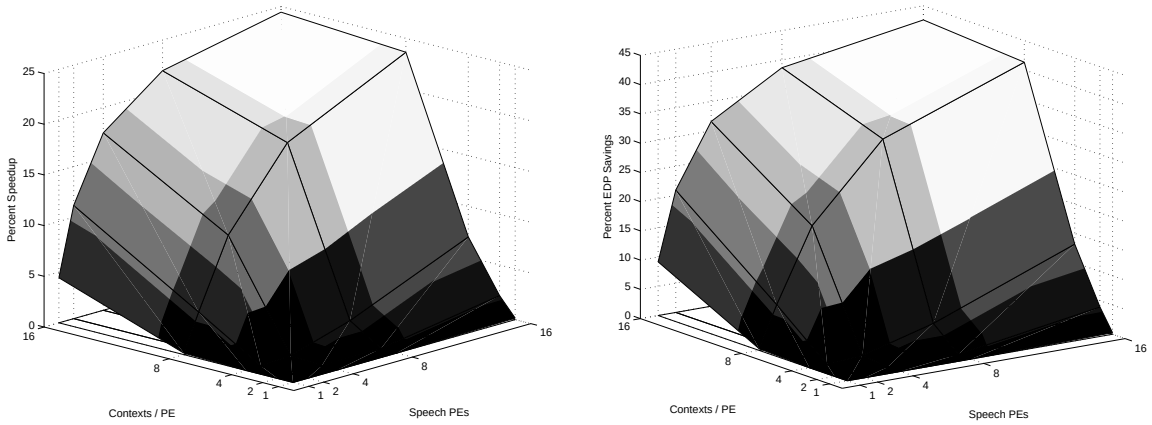
(a) Realtime Fraction

(b) EDP

Figure 8.1: Realtime Fraction and EDP of Base SDRAM system with Concurrency Throttling - This figure depicts the performance of the base SDRAM system (100MHz SDRAM, 2k L1 caches) with a concurrency throttling threshold set at 20 outstanding requests (a), and its EDP profile (b). We clearly see here the beneficial effect of throttling on the performance of added thread contexts in (a). As shown in (b), despite the improved performance profile, the EDP still increases at these larger configurations (performance never exceeds smaller configurations), raising the question of whether it is better to have excessive contexts and throttling, or just maintain a smaller number of contexts (less hardware / energy consumption).

running parallel contexts to currently available memory system resources, and results in a dramatic reduction in use of concurrency (and, consequently, number of issued memory requests) at times when they would be ineffective. The existence of the added physical contexts, however, allows the system to run with greater concurrency if running threads are not placing demands on the memory system.

The effects of this modification on realtime relative performance and on EDP is shown in Figure 8.1 with concurrency throttling occurring at 20 outstanding memory requests. Note that this is not meant to imply the presence of a single centralized memory queue, as the actual data for these requests can be maintained in a distributed fashion across processing elements. As figure (a) shows, this simple throttling policy alleviates much of the performance loss seen in systems with higher contexts. In fact, the figure depicts an interesting transition point, as performance drops for a few added contexts, and then



(a) Relative Performance

(b) Relative EDP

Figure 8.2: Relative Performance and EDP of Base SDRAM system with and without Concurrency Throttling - This figure depicts the relative performance (a) and relative EDP (b) of the throttled model versus the base model. We see that both performance and EDP are substantially improved through throttling, which is expected given the improved performance. We also see that, in general, the throttling is either effective or irrelevant. We do not see a case where throttling actually decreases performance.

recovers with a greater number of contexts. What we observe here is the effect of limiting request latency through throttling. The dip in performance corresponds to regions where the 20 outstanding request limit remains too high, but the number of contexts is still too low to tolerate the latency. As we add an even greater number of contexts, we see better toleration of the now relatively well limited memory latency. Furthermore, the extra physical contexts can still be exploited when memory is not bottlenecked. As a result, unlike the original memory system analysis, including a larger number of physical contexts slightly out-performs the previous peak (at around 4 contexts).

Figure 8.2 presents this new data in a slightly different light, looking at the relative difference in performance and EDP between the system without throttling, and with throttling. That is, this figure shows the relative benefit of throttling as compared to the base system. It is clear from this figure that throttling leads to substantial relative improvements in performance and EDP when bandwidth is constrained. More importantly, we see that for a throttling threshold that is sufficiently high (as 20 is), we do not see circumstances where

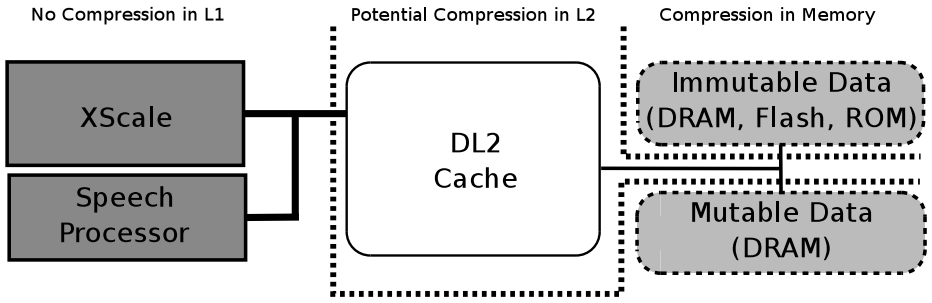


Figure 8.3: Overview of Compression Domains - This figure depicts the two compression domains considered in this work. Primarily, our evaluation focuses on compression at the level of the L2 cache. In–memory compression has minimal impact on performance, and is therefore clearly beneficial

performance is actually hindered. Thus, in the worst case, throttling incurs only an energy / complexity overhead, not a performance overhead.It should be noted that if the throttling threshold is set too low, the full memory bandwidth will not be utilized and performance may, in fact decrease.

While these results demonstrate the power of our programming model and the effectiveness of a simple optimization (conceptually, concurrency throttling could be tied to any measurable system bottleneck), it is important to note that performance never substantially exceeds that achieved before the bandwidth peak. As overall performance is at this point almost completely constrained by memory throughput, this is a fully expected result. The effects of this are best seen in figure 8.1b, which shows that despite the improved performance, the lowest EDP is still found in the range of lower numbers of processors and contexts. Thus, when energy dissipation is taken into account, it appears better to simply construct a smaller system tuned to match memory bandwidth in the general case than to include extra, often unused resources and allow for a dynamic optimization.

8.2 Data Compression

Data compression is an obvious potential optimization for this application. Effective use of low-overhead data compression could substantially reduce the memory needed to store knowledge base data. Indeed, a number of works have shown that compressed main

memory systems can dramatically improve performance in situations where program sizes exceed available system memory [7, 48, 80, 98]. While virtual memory system and disk latency is not the direct motivation for our interest in compression, the principle remains true at other levels of the cache hierarchy as well. For example, compressed data maintained on the processor (say, the L2 cache) could improve performance of smaller caches and help to alleviate memory bottlenecks by reducing the amount of data that must be transferred. On the other hand, systems employing data compression must deal with potential added latency due to compression and decompression operations. Moreover, data compression invalidates program generated addresses, requiring either compile time knowledge of the compressed address space, or runtime address translation.

The resource benefit of maintaining compressed data in the memory system is fairly obvious for this domain. As our architectural analysis thus far suggests that small latency additions for decompression at the memory level would have negligible effects on performance, this is not really an interesting permutation to explore. Rather, what we explore here is the potential benefit of maintaining compressed data on–chip. We note that the L1 cache is very intolerant of increased latency due to high utilization and demand by processing elements. We therefore focus our evaluation on maintaining compressed data at the level of the L2 cache. This has the potential benefit of increasing the effective size of the cache, and relieving some of the bottleneck to the memory system proper. Recognizing the intrinsic characteristics of the speech recognition application domain, we explore compression of only static (immutable) knowledge base data. This constitutes the bulk of the memory footprint of the application, and eliminates the need for runtime compression (as uncompressed data need never be written back), producing the greatest potential benefit at the lowest cost. An overview of these compression domains is depicted in Figure 8.3.

8.2.1 Compression Overview

Data compression in general involves replacing sequences of original data with short, representative code sequences. The efficiency of compression is related to the size of these

code sequences relative to the size of the data they replace. For example, a commonly utilized set of large block serial compression algorithms are built around techniques proposed by Ziv and Lempel [104], which take advantage of repeated sequences found in written text and replace repeat sequences with pointers to previous occurrences.

Unfortunately, many highly efficient off-line data compression techniques can not be directly applied to on-line runtime data compression. The LZ family of algorithms, for example, require sequential traversal of data in the same order during compression and decompression, making random access to data elements impossible. While techniques such as block based compression can be utilized to mitigate some of these problem (trading off compression efficiency for pseudo-random decompressibility), an understanding of compression methodology and application structure opens up the potential for compression techniques that resolve access problems while maintaining high compression ratios. We therefore begin our analysis by considering the compression problem, and techniques used for applications such an code (instruction) compression, which has similar constraints to those seen in our target data. While we will not explore all of the potential optimizations described in this subsection, this overview should provide a general understanding of current compression techniques, and potential applications with respect to increasing effective memory size and reducing power dissipation.

LZ Compression

As discussed previously, LZ style serial compression schemes suffer from a number of problems when applied to runtime data, the most serious being difficulties in performing random accesses. A brief exploration of block based techniques suggests that if blocks are broken up on the natural boundaries of knowledge base elements, the resulting block sizes are far too small to achieve any reasonable compression efficiency. It is important to note that, while this work does not explore such options, variations to program access patterns such as those performed by Mathew et al [65] which result in fixed, sequential access to large portions of the knowledge base may make selective sequential compression methods

viable.

The premise of LZ compression is taking advantage of sequence repetition found in serial traversals of many data sets, particularly written text. For example, in the LZ77 algorithm, codewords represent indexes into a dynamically constructed array of character sequences. The first 256 entries are statically defined to represent single ASCII characters. Entry 257 represents the first two character sequence seen in the application. Each time the algorithm observes a character sequence, it matches to the longest known (previously observed) sequence, exporting the numeric code for the longest match. It then adds a new sequence representing the full known sequence plus the next (currently unmatched) character, thus making this longer string available for future sequence matches. As the dictionary of known sequences grows, the potential for compression improves. It is important to note that the resulting dictionary is comparable in size to the original data. The advantage to such compression methods is that the dictionary may be discarded once compressed output is generated. The decompression algorithm can rebuild the entire dictionary on demand by traversing the compressed data sequentially. A small alteration of this approach utilized codewords that represent pointers to earlier portions of the original text sequence. Conceptually, this is identical to the described algorithm except in the specific encoding utilized for each codeword. It allows the data to be utilized as the dictionary, but it practice still constrains the system to serial decoding.

The difficulties in applying this style of compression to random access data should be obvious. It is impractical to store the entire compression dictionary, as much of the size benefit of compression is wasted (note that this assumes a time efficient dictionary storage method, and that other techniques may be used to reduce dictionary size at the expense of increased decompression time). Without a static copy of the dictionary, decompression must occur at the beginning of the compressed data sequence, as the codewords observed in the compressed data stream would be meaningless otherwise. One potential approach to bridging these problems is block based compression, which compresses data in fixed size blocks, allowing decompression of any data needed in a given block to begin at the start

of the block rather than the start of the entire data component. Due to the nature of the algorithm, however, blocks must be of considerable size (in the kilobytes) before benefits are seen. Clearly, if the block ends before a sufficient dictionary of in–block sequences is built up, compression will be ineffective.

Code Compression

A subset of the general research area of data compression that better matches the algorithmic constraints of this application space is code compression. This involves compressing program code in the memory system, and uncompressing it as it is loaded into the processor for execution. Work in this domain must deal with essentially random access patterns, and relatively small block sizes (generally corresponding to basic block sizes).

The issue of code compression has been addressed in a number of works [60, 59, 99]. While the specific algorithms and techniques utilized in these works are not directly relevant to our current discussion, a number of general themes are important. Most important is the very common use of a Line Address Table (LAT, first described by Wolfe et al [99]) or similar structure to remap program generated addresses to compressed address block addresses. This capability is crucial to performing random data retrieval, as it eliminates the need to decompress all previous data in order to identify the start location of the target data. In general, LAT entries correspond to blocks that are of interest to the domain of coding, such as basic block sizes [60, 59] or procedure size [47].

We conclude from these works that a number of compression approaches may be more or less viable (depending on application domain) for a given random access compression task. The problem of address translation, however, introduces a substantial concern. Specifically, the LAT approach, while potentially viable for program code, is likely to be problematically large and unwieldy when applied to large data sets that must still be resolved at fairly small block levels. Solutions to this general problem exist in the memory system (such as the translation tables utilized it IBM's MXT technology [66]), but the space constraints involved will be an issue as we explore bringing compressed data onto the processor to

improve cache performance.

Compressed Cache

An obvious step in the progression of compressed data usage is the idea of maintaining compressed data on cache. This increases the virtual size of the cache structure, leading to potential increases in cache hit rates. If the decompression overhead seen by the processor is less than the overhead that would be incurred by cache misses, the result can be substantial improvements in program performance. Once again, a number of works have investigated the issue of cache level data compression, and explore a number of interesting program characteristics that could be beneficial in this domain.

The Frequent Value Cache presented by Zhang, Yang, and Gupta [103, 102] makes note of the high frequency of occurrence of a small subset of values in many data accesses. They show that these common values can be exploited to increase effective cache size or reduce overall cache power dissipation by either storing or accessing fewer bits when frequent values are detected. Related to this type of optimization is the observation of the prevalence of zero values often stored in caches and utilized by programs. Villa et al [93] optimize the case of reading and writing zero values by storing and reading only a single bit, thus saving considerable energy.

Similar optimizations to expose potential energy savings are performed by Kim et al [46], who observe that much of the data stored in cache lines consists of sign or zero extended data. In this work, they compress such data into a single bit, reducing access power consumption and allowing the potential storage of a second data word in the same physical space, doubling the effective size of the cache for the cost of some extra tag matching hardware.

The main observation to take from this discussion is that derivative techniques related to data compression may be utilized for other applications such as energy reduction. In general, this is achieved by taking advantage of general characteristics of the data stream, such as the prevalence of common values or well known values.

8.2.2 Analysis Methodology

In order to explore the potential benefits of data compression to the on–chip cache level, we consider two configurations. First, we consider a somewhat optimal design, which could be thought of as "LZ for free". Specifically, we consider compressing the entire static knowledge base using a single window LZ scheme. That is, we treat the entire static knowledge base as a single block operating off of a single sequence dictionary, thus maximizing the potential benefits of this compression style. We then assume essentially free access to this data, remapping addresses to compressed address locations, loading from memory, and decompressing with no added delay. While this represents only one of a wide array of potential compression techniques, a full accounting of all permutations is beyond the scope of this work, and we believe this idealized configuration is sufficient to illuminate the potential benefits of cache level compression overall. On our baseline knowledge base, this technique achieves approximately 3x compression of static data.

Our second compression technique focuses on observed characteristics of the baseline knowledge base to achieve reasonable compression while meeting basic feasibility constraints (such as random–access and easy on–cache decompression). The goal here is not to suggest a general scheme for compressing speech recognition knowledge base data, but to produce a timing model by which to evaluate the tradeoff between lower cache miss rates and increased latency on individual requests.

Analysis of the sum of static knowledge base data in Sphinx 2 reveals that at the word level, the majority of data elements are only a few different values. For example, in our sample knowledge bases over 50% of the individual word values in the data range are zeros. We recognize that this is a property of techniques already applied to the knowledge base (such as vector quantization on the Gaussian scoring data), and may not remain true in other knowledge base models. We will, however, take advantage of this fact for our tradeoff evaluations.

Utilizing this static data characteristic we consider a system in which the set of most frequently observed word values is identified at compression time, and is used to produce

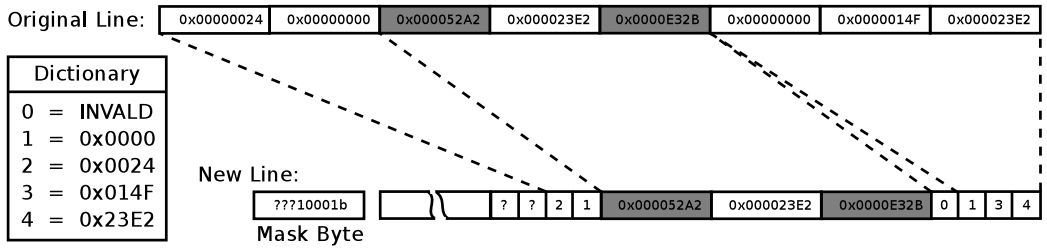


Figure 8.4: Example of Custom Compression Algorithm - We utilize a simple compression algorithm that takes advantage of the presence of high frequency data values produce estimated timing information for tradeoff analysis. This approach remaps frequent word values identified by profiling to byte values. Note that data is selectively compressed, such that the 5th word in the original cache line (surrounded by uncompressible segments) is left expanded. Our baseline model uses a 256 entry dictionary of word values. This method achieves over 2x compression on our sample knowledge base, and provides a number of useful characteristics (fixed knowledge base, random access) for this domain.

a small, fixed–size mapping dictionary. We then employ selective encoding of symbol sequences made up of dictionary entries, achieving compression by packing codes for a number of word–sized symbols in a single data word. Each codeword is of fixed length, simplifying decoding. Since the dictionary is statically defined, decompression can begin anywhere.

At the word level, an example of this approach is depicted in Figure 8.2.2. In this case, we are assuming a 256 entry dictionary, where each byte code represents a word of original data. Conceptually, this approach is the equivalent of each entry representing a common four byte sequence, and could be extended to any convenient or optimal sequence length. We assume that only contiguous symbols may be compressed together. Thus, as the first word of the example shows, only three out of the four possible byte codeword slots in a single word are filled when an uncompressible symbol is encountered and must be placed at the next word boundary to maintain alignment. In order to minimize book–keeping, we reserve the codeword zero to represent an invalid entry. Thus, unfilled byte codeword slots in a given word are set to zero, and techniques such as Dynamic Zero Compression or sign compression may be applied.

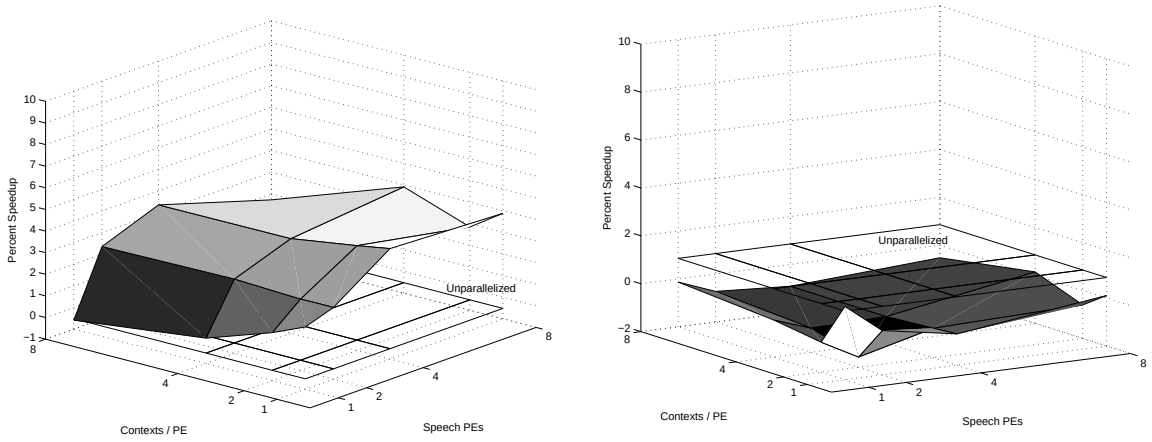
As we are performing selective compression, some method is required to identify which data is encoded and which is not. To minimize the overhead of such book–keeping information, we track it at cache line size blocks. Where the original system is assumed to have

a 32–byte cache line, the compressed memory system maintains 33–byte cache lines. The extra byte is not considered part of the actual address space, as it is never directly referenced by software. Rather, it is a bitfield recording which words in the cache line contain encoded data.

When compressed data is maintained in the L2 cache, loads from the memory system pass this 33rd bit back, but also pass information regarding the actual address range included in the loaded cache line. As we assume a 64–bit data bus to the memory system, this extra address data can be incorporated into the extra cycle needed needed for the compression mask, essentially adding an extra bus communication cycle to memory requests. Inherent is this organization is the assumption that address mapping and translation (from the "uncompressed", program generated address to the actual location of the requested data in the compressed data range) is done at the level of the memory system itself. This organization offloads the space requirements of address translation to a portion of the memory hierarchy where space is not as constrained, and is already essentially supported by existing systems such as IBM's Memory Expansion Technology [66].

Incorporating the added information for compression obviously requires modifications to the L2 cache structure as well. Specifically, given our compression methodology, three modifications are necessary. First, each cache line requires space for the extra non–address byte holding compression mask information. Second, the tag matching infrastructure must be modified to note the actual data range returned by the memory system on a load, and match addresses within that range to the existing cache line. Finally, on cache accesses, data must be uncompressed before it is passed up to the L1 stage.

For the sake of this evaluation, we assume the added cache byte per line and tag matching resources are free in terms of latency and energy. We account for the added cost of compression by adding an extra memory bus cycle to all requests accessing portions of the static knowledge base, and by modulating the latency for a L2 cache hit by a factor representing decompression time. Specifically, we assume conservatively that each compressed word in a cache line requires a single cycle to decompress, and that full decompression of



(a) LZ With Zero Cost

(b) Custom Algorithm Supporting Random Access

Figure 8.5: Relative Performance of L2 Static Data Compression - This figure depicts the relative performance benefit of a system using latency free, random access (idealized) LZ compression in a 128K L2 over a system without data compression (a), and the relative performance of our custom algorithm for random access data compression (with all associated delays and resource allocations) over a base system (b).

the cache line is necessary on each hit. Thus, a cache line containing 4 compressed words will require the base cache hit latency plus 4 cycles. As before, this is the internal delay of the L2, and does not account for bus communication and request / response transmission time, which are accounted for in our communication network model.

Note that neither of the compression techniques considered here involve lossiness, or reductions in data quality. Given the domain under evaluation, slight modifications that trade off accuracy for size are certainly feasible. We avoid such compression techniques in this work because techniques such as vector quantization (a lossy compression scheme) have already been applied to the Sphinx2 knowledge base. Furthermore, such optimizations could require significant algorithmic changes, and introduce overall speech recognition tradeoffs which are beyond the scope of this work.

8.2.3 Performance Analysis

We evaluate the potential performance / storage benefits of compression by dynamically compressing marked static knowledge base data at runtime through address translation at the level of our simulator memory model. That is, compression evaluation is done in the simulator, with no modification to the original program, ensuring consistency in program flow. Translation from the raw memory address to the compressed address is performed between the L1 and L2 cache (in the idealized simulation model), ensuring that the L2 and all subsequent components of the memory system account for cache / page hits corresponding to the compressed address space.

The relative performance of an LZ based compression scheme over the base system using this methodology and assuming no added latency or resource consumption on a system with a 128k L2 cache is shown in Figure 8.5a. As this relative performance graph clearly depicts, the performance benefit of compressed data on–cache is minimal, despite achieving over 3x compression in an idealized environment. Indeed, if we incorporate a more realistic timing model with our custom compression scheme, overall performance essentially breaks even as shown in Figure 8.5b.

The reason for this complete lack of performance improvement becomes clear if we explore the access characteristics of the program. First, as previously discussed, static knowledge base data demonstrates stream–style characteristics in reference patterns. Thus, while data compression increases the effective size of the L2 cache, this increased effective size at 2x, 3x, or even substantially greater compression only captures a little more of the working set of the program.

Even given this factor, however, we would expect at least to see performance increases on par with the effects of doubling or tripling L2 size in the previous chapter. The reason for lack of such improvements comes from the way the program accesses individual static model data elements, and the fact that these data elements generally span multiple cache lines. If, as is occurring here, the program accesses a few bytes out of a mult–cache–line data element, the number of cache lines accessed in a compressed environment may not

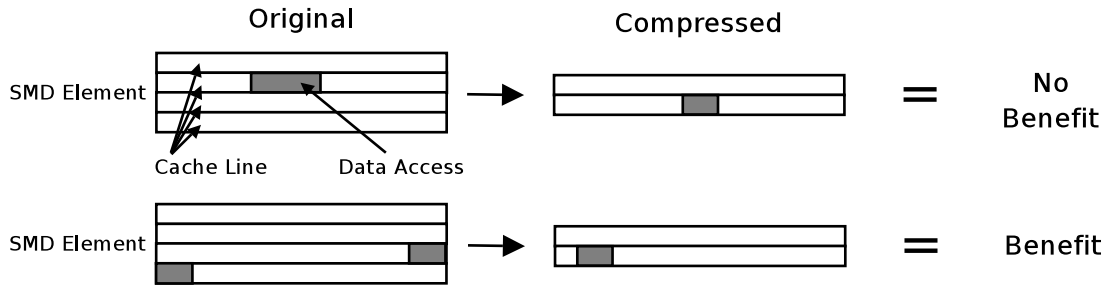


Figure 8.6: Static Model Reference Example - This example considers simplified access patterns to multi–cacheline static model data elements, suggesting why compression of such data may not lead to as significant a reduction in cache miss rates as might be otherwise expected.

actually decrease. In the degenerate case, if the original system only accesses data from a single cache line in a multi–cache line data element, then even in the compressed case a single cache line would be accessed (assuming compression is not sufficient to pack multiple data elements into a single line) and no reduction in cache miss rates would occur at all. In reality, the small performance improvement comes from program references which, in the original case, span multiple cache lines of a single data element. The frequency of such accesses, however, is low relative to the total number of static knowledge base references. An example of these reference patterns is depicted in Figure 8.6.

It is important to recognize that while compression at the level of the L2 cache does not appear to lead to a significant performance improvement or reduction in L2 misses, it does reveal important information about the potential uses of compression in this domain. If we modify our modeling environment to perform decompression in the memory system and maintain standard uncompressed data in the L2, we find minimal performance loss even for significant decompression latencies. This suggests that the best approach to reducing data size in this domain is to maintain highly compressed data in the lowest level of the memory system, along with an appropriate decompression system. Such a configuration has a number of advantages, allowing for the use of highly specialized knowledge base specific compression approaches without consideration for generalizability (so long as random access and reasonable decompression latencies are maintained). One could even

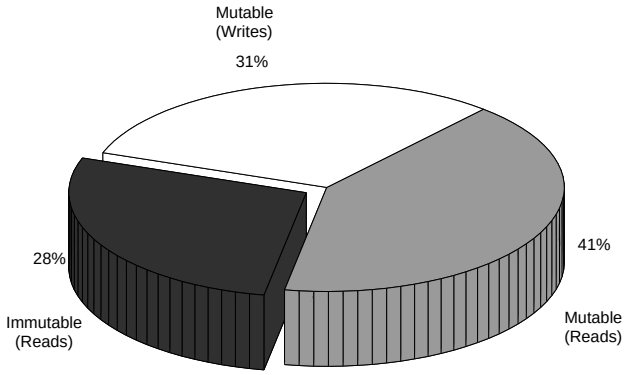


Figure 8.7: Memory Bus Transfer Breakdown - This figure depicts utilization of the memory bus for immutable and mutable data transfers. As clearly seen, while immutable data may constitute the vast majority of the memory footprint of the application, it does not represent the majority of bandwidth consuming data transfers along this resource.

envision a plugin ROM module containing the compressed knowledge base and a on–chip decompresser, making the compressed data entirely transparent to the rest of the system. Given the substantial size of static knowledge base data, such a methodology could have a significant impact on physical resources needed to support speech recognition.

8.2.4 Bandwidth Analysis

While the discussion thus far has demonstrated that maintaining cache line size and compressing data into the L2 is ineffective due to a lack of advantageous spatial locality in the static knowledge base data, it ignores potential bandwidth benefits of such an optimization. Reducing data transfer size to match compression rate will certainly have the effect of reducing memory bandwidth demand. We consider such an optimization by envisioning a system in which compressed data is decompressed at the chip boundary before being placed into the L2 cache structure. Our analysis finds a fairly small 5–7% performance improvement in the general case.

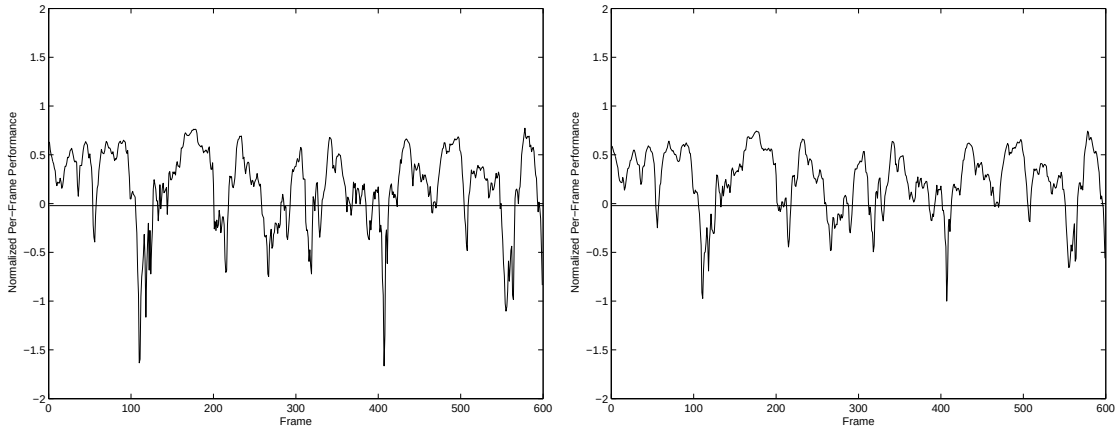
Further exploration of the data streams shows why performance improvements are fairly small in this configuration. While the static knowledge base data makes up the vast majority of total memory footprint, it makes up only a fraction of the data transferred across the memory bus over the course of program execution. This is intuitively understandable

considering that program characteristics generally combine loads of mutable and immutable data, and that mutable data must subsequently be written back to the memory system. It is best depicted, however, in Figure 8.7, which shows percentage of transfers (on an uncompressed system) along the memory bus, broken into mutable data reads and writes, and immutable data reads. Clearly, while reducing the payload size of immutable data reads will have a benefit on bandwidth utilization, it will not have as substantial an effect on overall system performance as might initially be surmised. Requiring on–chip decompression also resurrects constraints on compression methodology not necessarily present in off chip decompression. Thus, the potential costs and benefits of such an implementation are likely to vary with individual system and speech recognizer configurations, and must be under those contexts. Note that compression and decompression schemes for mutable data are far more likely to provide a bandwidth benefit in this domain. Given the relative size of mutable knowledge base data, the overall reduction in memory footprint will not be as significant.

8.3 Runtime Power Management

Over the course of evaluations performed thus far, we have arrived at a processor and memory system architecture that is able to perform continuous speech recognition at realtime and with energy consumption on order of the base low–power system (for a given workload). Much of our analysis thus far has treated the speech input as though it were fully buffered, allowing the architecture to proceed ahead of realtime when computationally possible. We briefly note in the previous chapter that this may not be a realistic mode of system operation. Minimally, it is far more likely that the system would buffer some fixed quantity of input speech and pass that to the recognition infrastructure as a batch. In the degenerate case, the system buffers a single input frame, and the processor architecture must halt if evaluation of that frame completes before the next is available.

As discussed earlier, this "extra performance" presents the opportunity to perform more rigorous searches (requiring added computation) or reduce overall energy consump-



(a) 4 Processor

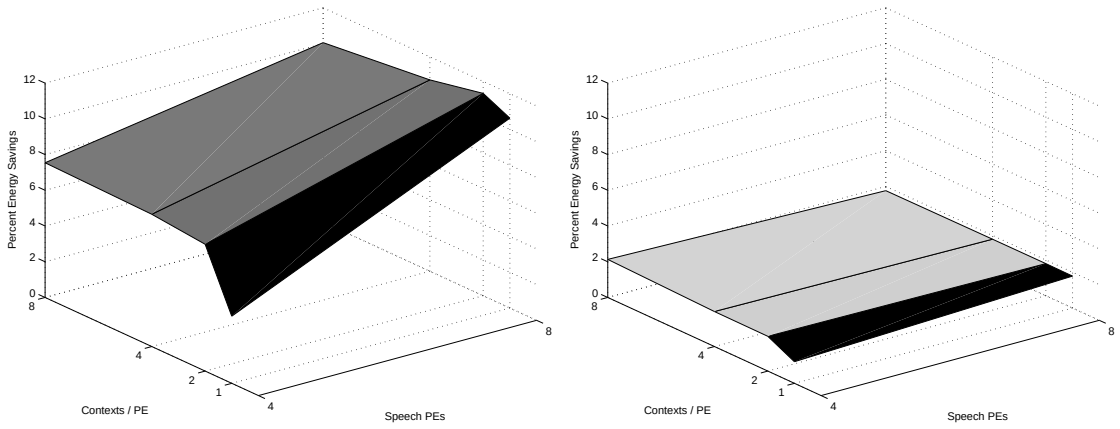
(b) 8 Processor

Figure 8.8: Normalized Per–Frame Realtime Comparison - These figures depict the per frame performance of a 4 and 8 processor system, depicting the number of cycles by which the system beats realtime performance for each input frame (normalized to the total number of cycles required to match realtime performance per–frame).

tion through runtime power management techniques. As we are investigating low–power domains, this section will consider the second of these two options, exploring the use of runtime power management to trade off excess performance for reduced energy consumption. We primarily evaluate the potential for voltage and frequency scaling, as well as frame buffering and halting, and conclude with an analysis of how processing technologies and background energy dissipation alter the dynamics between these options. The motivation for this effort can be seen in Figure 8.8, which considers the number of cycles by which a 4 and 8 processor (4 context) sample configuration beat realtime on a per–frame basis (normalized to per–frame realtime). While there are instances in which the system falls behind, a significant fraction of input frames are completed before their realtime deadlines expire.

8.3.1 Standby Halting

We begin our evaluation of power management by considering the effect of placing the system into a low–power standby state while awaiting new input data. As discussed during memory system evaluation, the fixed rate of input means the cycle at which future input



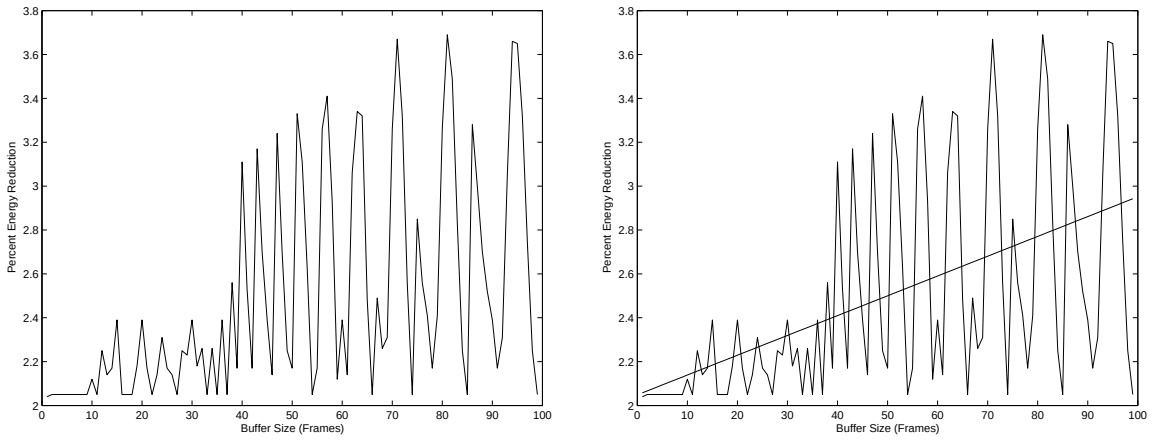
(a) Full Input Buffer

(b) No Input Buffer

Figure 8.9: Energy Benefit of Standby Mode - This figure considers the energy benefit of entering a low power standby mode over remaining at idle power dissipation during times when the system is awaiting input data. Figure (a) considers a situation in which the entire input is buffered, and processing proceeds uninterrupted until its conclusion. Figure (b) by contrast considers the effect of no buffering, placing the processor into standby mode on a per–iteration bases if processing is completed before the next frame of input is available.

data will become available can be predicted with high accuracy. Thus, it is possible to enter and exit standby mode without affecting performance. In this section, we evaluate the energy effects of such an approach, as well as the potential uses of buffering several input blocks for contiguous processing. Note that the evaluations performed up to this point essentially model an infinite buffering capability, processing the entire speech input without pause. Live speech recognition most likely excludes the possibility of recording the entire input before processing, but buffering a portion of the input may effectively amortize the cost of entering and exiting standby mode, and maximize efficiency.

We first begin by considering the workload energy benefit of buffering the entire input, processing it, and then entering a standby state (accounting for standby power until "realtime") versus simply maintaining an idle active state. This comparison, shown in Figure 8.9a, establishes a baseline with respect to the benefits of going into a standby mode. Note that since total processing time is considered to match realtime in both configurations, the energy differences are the metric of interest. Contrast this energy reduction with that shown in Figure 8.9b, which considers the same comparison with no buffering of input. In



(a) Raw Energy Savings

(b) Trend Line Overlay

Figure 8.10: Energy Savings by Buffer Size - This figure considers the percent energy savings over the idle waiting configuration for a 4 processor, 4 context system. The artifacts seen in (a) are determined to be due to end effects. Thus, the trendline presented in (b) is a better depiction of the potential benefit of buffering.

this configuration, the processor is placed into standby mode if processing of a given input frame completes before the next one is available, and if there is sufficient time to enter and exit standby mode. These graphs demonstrate that input buffering can be helpful in amortizing the latency of entering and exiting standby mode (100 cycles, as described in Appendix B), allowing the processor to remain in a low–power state for longer.

In order to explore the space between these two extremes, we attempt to evaluate the energy tradeoff for various intermediate buffer sizes, expecting to see a smooth curve from the no buffer (more precisely, single frame buffer) case toward the full buffer case. Instead, we find a curious variation in energy savings, as seen in Figure 8.10a. As it turns out, these oscillations are due to a number of overlapping factors related to the specific characteristics of our evaluation workload, some of which remain elusive. The major contributor, however, appears to be the fact that the last several frames of the evaluation workload we use for this set of experiments require longer than realtime for processing. Thus, if the specific buffer size causes the system to halt just before entering this longer running section, the last portion of execution ends up adding to the total number of execution cycles. By

contrast, if a large buffer size correlates well with the end of the input, then the extra time required to process the last several frames is hidden behind "faster than realtime" performance on earlier frames, allowing the system to finish sooner. Based on this analysis, the same patterns would be observed even if the last frames could be completed faster than realtime, as the system would still hold off execution of those frames until the input became "available". The effect is more pronounced in this case simply because the resulting exposed performance loss in greater.

We determine from this data that should power management be performed by use of a low–power standby state, some degree of input buffering to maximize standby time is likely beneficial. Since the actual length of the input sequence can not be known beforehand, it would not be possible to select a buffer size that maximizes performance effects. Furthermore, for longer running continuous input, the actual effect would be far less visible. Having a buffer will, on the average, save energy by amortizing entry and exit from standby. This is best depicted in Figure 8.10b, which overlays the raw data with a trendline, showing increased benefit. Note, however, that even with half second and full second buffers, the energy savings appears to be fairly small. As we will discuss toward the end of this chapter, varying our assumptions about process technology, clock gating, and leakage current can dramatically affect the potential tradeoffs seen here.

8.3.2 Voltage Scaling

While the previous subsection considers a "get it done and wait" approach to power management, a second approach to mitigating energy consumption is to slow down the processor to match the demands of the application. This subsection will consider voltage and frequency scaling, both in an ideal sense and through runtime frequency modulation in our simulation infrastructure, to explore the potential energy savings of such an approach.

Voltage / Frequency Scaling Computations

Before beginning considering voltage scaled data, we take a moment to detail the scaling equations used to arrive at these estimates in our work. Much of the base computation and information used here is taken from Kim et al [45] and developed to meet our needs. Note that we apply scaling only to on–chip components. Memory system components are assumed to dissipate extra background energy (as described in Appendix B) to account for the extra idle time due to a slower processor.

We begin by considering the overall power consumption of a device. At a high level, this can be described as:

$$P = P_{dynamic} + P_{static} \tag{8.1}$$

where $P_{dynamic}$ represent the dynamic power lost from charging and discharging capacitative loads, and P_{static} represents power dissipated due to leakage current. Note that, as in cited work [45], we ignore power lost in the momentary short circuit that occurs during gate switching. Expanding Equation 8.1, we arrive at:

$$\begin{aligned} P &= ACV^2f + VI_{leak} \\ &where \\ I_{leak} &= I_{sub} + I_{ox} \end{aligned} \tag{8.2}$$

In this form, A represents the activity factor of the device (fraction of gates actively switching) and C represents the total capacitative load. The leakage power component is seen to involve the system voltage and leakage current component, which in turn is made up of subthreshold leakage and gate oxide leakage. In practice, the contribution of leakage current to overall power dissipation at .18μm technologies is negligible (though it becomes a significant factor for smaller technologies). This allows us to simplify Equation 8.2 to:

$$P = ACV^2f$$

from which we can deduce relative power dissipation from voltage scaling to be:

$$P_{rel} = \frac{P_{new}}{P_{old}} = \frac{A_{new}C_{new}V_{new}^2 f_{new}}{A_{old}C_{old}V_{old}^2 f_{old}}$$

or

$$P_{new} = P_{old}(\frac{A_{new}C_{new}V_{new}^2 f_{new}}{A_{old}C_{old}V_{old}^2 f_{old}})$$

Recognizing that the activity factor of the device and the capacitative load do not change with our scaling variations, we arrive at a near final form of:

$$P_{new} = P_{old}(\frac{V_{new}^2 f_{new}}{V_{old}^2 f_{old}}) \tag{8.3}$$

While this equation represents the basis for much of our scaled power and energy estimation, one further step is necessary to make it useful. Specifically, the new voltage and frequency values must be represented in terms of original system values. Determining the new frequency is fairly straightforward. In the idealized scaling case, the new frequency is simply determined to be the number of processor cycles required to complete the workload divided by the number of seconds of the original speech input (realtime). In the case of simulated frequency modulation, our power management system reports the number of cycles of processor operation at a given frequency, and thus the scaled frequency is directly available.

Determining the scaled voltage requires a slightly more sophisticated formulation. Referring once again to Kim et al [45], we work from a normalized voltage and frequency model (operating voltage normalized to maximum system voltage), producing the equation:

$$V_{norm} = \frac{V_{op}}{V_{max}} = \beta_1 + (\beta_2 \cdot f_{norm})$$

where

$$\begin{aligned} \beta_1 &= V_{th}/V_{max} \\ & and \\ \beta_2 &= 1 - \beta_1 \end{aligned}$$

In this case, we can treat V_{op} as the new (scaled) voltage and V_{max} as the original system voltage because all scaling efforts are directed toward reducing system voltage and frequency. f_{norm} is simply the ratio of f_{new}/f_{old}. When $f_{new} = 0$, $V_{norm} = \beta_1 = V_{th}/V_{max}$, which is approximately 0.3 for modern technologies. Thus:

$$V_{norm} = \frac{V_{new}}{V_{old}} = 0.3 + f_{norm}(1 - 0.3) = 0.3 + (0.7)\frac{f_{new}}{f_{old}}$$

which finally concludes as:

$$V_{new} = V_{old}(0.3 + (0.7)\frac{f_{new}}{f_{old}}) \tag{8.4}$$

We may now combine Equation 8.3 and 8.4 to arrive at a means of estimating voltage/frequency scaled power consumption given original power consumption, voltage and frequency, and new scaled frequency, which are all known terms. As we utilize workload energy consumption in much of this work, we extend on Equation 8.3 to arrive at a scaled

energy estimation framework as follows:

$$
\begin{aligned}
Energy &= Power \cdot time \\
&\textit{and} \\
simtime &= \frac{simcycles}{frequency} \\
&\textit{thus} \\
Energy &= V^2 \cdot f \cdot \frac{simcycles}{f} \\
&\textit{or} \\
Energy &= V^2 \cdot simcycles \\
&\textit{thus} \\
E_{new} &= E_{old}(\frac{V_{new}^2 \cdot SC_{new}}{V_{old}^2 \cdot SC_{old}})
\end{aligned}
\tag{8.5}
$$

In Equation 8.5, SC represents the number of processor cycles registered by the simulator. In our idealized frequency scaling model, the number of simulation cycles are the same as in the maximum frequency case (only the time of each cycle changes), and thus the scaled energy is the original workload energy scaled by the relative operating voltages (as determined by Equation 8.4). When frequency modulation is modeled, the simulator provides a distribution count of the number of processor cycles spent at a given frequency and a given operating state, allowing direct computation of energy based on scaled energy at a cycle by cycle level.

Related Work

A number of works have explored the area of dynamic voltage scaling and power management, particularly under deadline constraints such as for realtime applications. Im et al [39] consider input and output buffering as a means of manipulating the occurrence of slack time in multimedia applications, dynamically scheduling voltage to minimize energy. They find significant benefit to such techniques in many real–world, multi–application environments. We explore similar effects with respect to buffering standby halt intervals, and

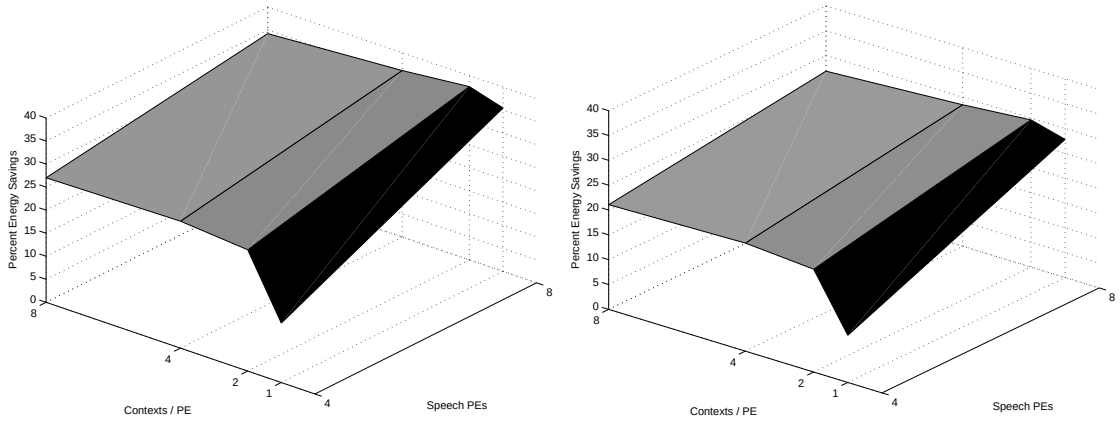
also conclude that some amount of buffering is beneficial in this domain.

Pillai and Shin [72] consider the use of DVS algorithms integrated with realtime OS task scheduling infrastructures to scale operating voltages while meeting all realtime deadlines. They find potential for considerable energy savings through such techniques. The approaches we consider in this work does not utilize direct interfacing with the running software, using instead known characteristics of the speech recognition domain to modulate frequency.

Hughes et al [37] consider both architectural and DVS style adaptation techniques to save power on multimedia workloads on a general purpose processing framework, finding that DVS techniques consistently outperform architectural adaptations such as gating off portions of the system to reduce capacitance. They base their adaptation system on frame boundaries (as we do here) and find that this natural work division has the potential of far better energy savings than previous work using fixed evaluation intervals [32, 57] which have been shown to produce little actual effect [27]. This supports the findings of earlier frame based evaluation that looked at smaller workloads and synthetic benchmarks [71, 72, 73].

A number of algorithms have been proposed to determine how DVS should be performed. These generally utilize a reactive feedback approach. Most interesting among these is a formal control theoretic approach proposed by Skadron et al [63] which attempts to select trigger conditions and reactions to DVS and thermal management tasks based on program behavior and history as opposed to fixed thresholds utilized in earlier works. In this work, we consider far more straightforward control approaches that make simple decisions based on immediate realtime performance information. The use of more sophisticated control feedback systems such as this are, however, certainly viable options in this domain.

An even more aggressive approach is suggested by Srinivasan and Adve [91]. This work takes advantage of known properties of multimedia applications to predictively alter system characteristics. Unlike their previous work [37], the goal here is actually to maximize performance within a thermal budget. The recognition of common properties and characteristics in frame–level processing of multimedia applications, however, can be applied



(a) Versus Idle Wait

(b) Versus Standby Weight

Figure 8.11: Energy Benefit of Ideal Voltage Scaling - This figure considers the relative energy decrease of ideal voltage scaling (exactly matching realtime) as compared to the base system with no power management and idle power dissipation while waiting (a) and against the ideal (fully buffered) standby halting approach (b). Note that, based on ideal scaling estimates, the 4 processor configuration (other than 1 context) would operate at around 370MHz, and the 8 processor configuration at around 300MHz.

to either goal (maximizing performance or minimizing energy consumption). In this work, we consider the simple case of predicting immediate future performance to be identical to current performance, as suggested by the relative performance relationship curves shown in Figure 8.8.

While we do not explore compiler directed techniques in this work, use of such techniques to reduce energy consumption on memory bound applications has also proven to be a fruitful technique [35].

Evaluation

We begin by considering an ideal view of voltage scaling, taking the total runtime of the application without realtime constraints, and directly scaling the energy consumption of processor components based on a frequency ratio determined by the relationship between reported execution time and real time. As discussed previously, only processor and cache energy values are scaled. Off chip components are assumed to dissipate background energy for the added time. In essence, this scaling model approximates the energy consumption

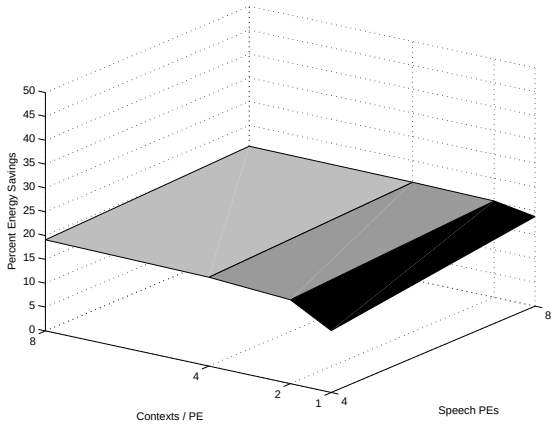


Figure 8.12: Energy Benefit of Base Dynamic Frequency Scaling System - This figure depicts the benefits of per–iteration voltage scaling in 25MHz increments over the idle wait implementation. Note that the energy benefit is not as high as ideal scaling due to the discrete nature of the adjustments.

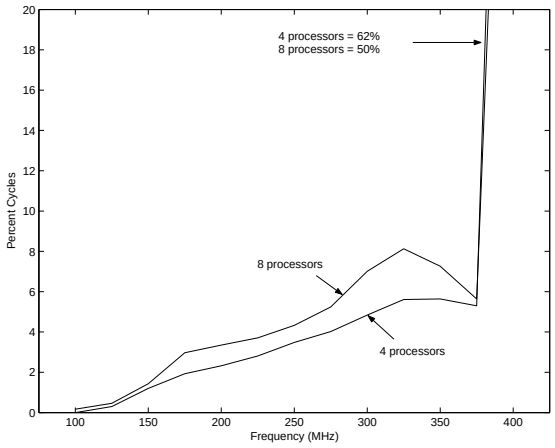


Figure 8.13: Operating Frequency Distribution of Base Scaling System - This figure depicts the distribution cycles spent at a given operating frequency for 4 and 8 processors (4 contexts each) using per iteration frequency scaling at 25MHz increments.

when the device can be set to a voltage and frequency that will match realtime beforehand.

The relative energy savings of this ideal voltage scaled model over a system that dissipates full idle power while waiting and one that fully buffers the input, processes it, then enters a standby state are shown in Figure 8.11. Our scaled frequency estimates suggest that the 4 processor configurations with more than one context would operate at around 370MHz, while the 8 processor configurations would operate at closer to 300MHz (recall that the baseline operating frequency is 400MHz). It is clear from these figures that voltage scaling presents a significant opportunity to reduce overall workload energy consumption.

Unfortunately, the scaling technique presented thus far is not implementable, as it requires foreknowledge of the computational complexity of the speech recognition task. A more realistic implementation would require recognizing specific points during execution and evaluating the current performance of the system relative to realtime goals. The most obvious point at which to perform this evaluation is at frame boundaries (similar to standby evaluation in the previous section). We therefore consider a runtime dynamic scaling model in which the current progress through the speech recognition task is evaluated at each 10ms

frame boundary, and operating frequency (and correspondingly operating voltage) are increased or reduced appropriately. Note that this corresponds to 100 "progress evaluations" per second, which is a fairly high resolution for such scaling efforts, particularly considering the high invocation overhead of DVS approaches. For this initial evaluation, we consider a system which, upon exceeding realtime constraints on a single frame, decrements the operating frequency by 25MHz, and on failing to meet realtime increases frequency by 25MHz. The frequency is bound between 100MHz and 400MHz. Note that for simplicity we assume more or less instantaneous voltage and frequency transitions when they occur. Various implementations of actual voltage and frequency scaling incur various (often high) invocation overheads which we do not consider here.

The energy benefit of this configuration over the base idle waiting configuration is shown in Figure 8.12 and the operating frequency distribution for 4 and 8 processors (4 contexts each) is depicted in Figure 8.13. We see that, while a reasonable energy savings is produced, it is only slightly more than half the energy benefit seen in the ideal case. This reduction in energy efficiency is due to the response time of the system, and resulting mismatches from the best frequency. For example, a series of short iterations will cause this system to progressively step down in voltage, expending excess energy until the clock rate has been slowed enough to match processing demands. The ideal configuration assumes the processor runs at the ideal frequency the entire time. Similarly, falling behind realtime will cause the frequency to be stepped up progressively, potentially causing the system to fall considerably behind before a high enough frequency is reached to catch up. This leads to unnecessary time spent in a higher frequency, higher energy state.

We consider a number of variations of our basic frequency scaling algorithm. Among them, we consider raising the lower bound of selectable frequencies and incorporating a hysteresis in frequency reduction (that is, the system must beat realtime for x iterations before the frequency is dropped). The intent is to explore the possibility that spending slightly more time at a higher frequency (rather than dropping down to 1xx MHz), may lead to a reduction in time spent at higher frequencies and improve the overall energy

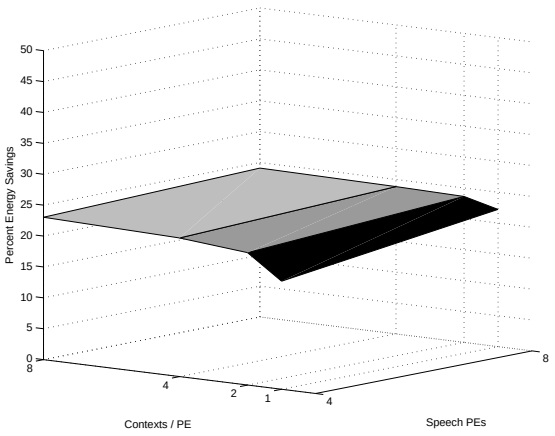


Figure 8.14: Energy Benefit of Performance Match Scaling over Idle Wait - This figure depicts the relative energy benefit of frequency scaling based on the latency of the most recent input frame relative to the base system with idle waiting. This approach achieves a slight (5%) energy benefit over the stepped approach previously considered.

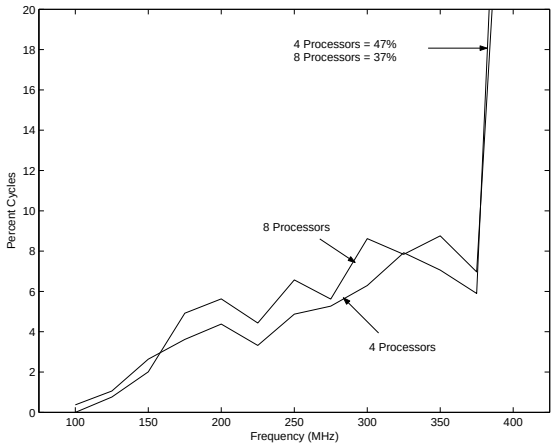


Figure 8.15: Operating Frequency Distribution of Match Scaling System - This figure depicts the distribution cycles spent at a given operating frequency for 4 and 8 processors (4 contexts each) using per iteration frequency scaling at 25MHz increments to match the optimal frequency of the most recent frame.

characteristics. We find that neither of these modifications have a significant effect on overall workload energy, and what effect they have is generally negative. We further explore the possibility of altering the scaling increment from 25MHz to 50 and 100 MHz, which lead to a nearly insignificant impact on overall energy consumption. Finally, we consider faster ramp–up when the system falls behind realtime. In the extreme case, regardless of current frequency, the system returns to 400MHz if it falls behind realtime goals. This too has a negative impact on overall energy consumption.

We determine from the data thus far that a wider scaling range and narrower resolution within that range are more beneficial to energy savings. We further explore this space by considering alterations to the basic scaling algorithm. Rather than stepping up and down in frequencies as done thus far, we note from Figure 8.8 that in the general case the current performance relative to realtime is a fairly good predictor of near future performance relative to realtime. Thus, we consider a system in which the scaling engine selects the closest frequency (at 25MHz intervals) that is greater than would have been necessary to

beat realtime in the last evaluated input frame (again, bound by 100MHz and 400MHz). Note that such an evaluation would require significantly more sophisticated logic and likely more time than our earlier scaling algorithm, and that these added costs are not considered in our framework. The relative energy of this algorithm over the idle waiting model is depicted in Figure 8.14, and the operating frequency distribution is depicted in Figure 8.15. We see that this approach achieves a slight energy benefit (around 3–5%) over the previous stepping model, corresponding to more time spent at lower frequencies. While this does not achieve the idealized scaling benefit, it does very nearly cut the difference in half. More aggressive techniques utilizing program knowledge of impending workload may help further approach ideal benefits.

We conclude from this data that performing frequency evaluation and scaling at the greatest possible resolution (the per–iteration boundary), and providing for the widest possible range of frequencies is beneficial. The specific modulation increment, within reason, does not greatly affect overall energy consumption. An equal ramp down – ramp up model appears quite effective, as does an approach that uses the most recent relative performance data to predict frequencies for near future iterations.

8.3.3 Processing Technology Effects

While the previous discussions have considered the effects of power management though standby waiting and voltage scaling, a true comparison of the two methods must take one other factor into account: idle power dissipation. The effect of such idle power dissipation increase as we move to smaller processing technologies, and can affect the tradeoff between running at lower voltages and simply shutting down.

In order to explore this tradeoff, we perform a high level analysis by evaluating the effect of setting idle power dissipation to various fractions of total power dissipation. The relative energy benefit of scaling over halting for 4 and 8 processors (4 contexts each) as we perform this variation is depicted in Figure 8.16. The figure also lists various modern processing technologies that correspond to general ranges of idle power dissipation. Note

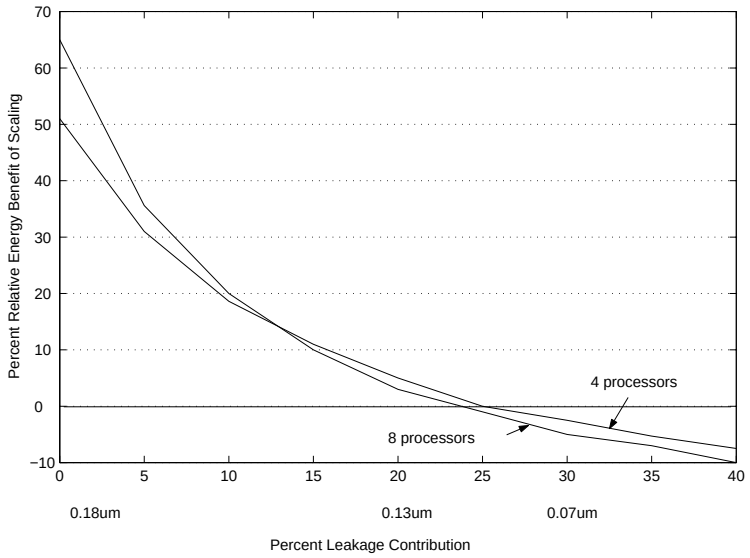


Figure 8.16: Technology Energy Distribution - This figure considers the relative energy improvement of a per–iteration, 25MHz increment frequency scaling system over a half second buffered standby wait configuration for 4 and 8 processors (4 contexts each) across a range of leakage energy contribution factors. This, as a first order metric, estimates the effect of various processing technologies (also listed) on the tradeoff between scaling and halting. We see that, when leakage is a small fraction of overall power dissipation, voltage scaling can produce a significant reduction in overall energy consumption. As the effects of leakage grow, the benefits wane, until eventually it is better to simply finish processing quickly and enter a low power standby state that minimizes leakage current.

that in order to perform this evaluation, we have departed somewhat from the detailed energy estimation framework utilized in the remainder of this work. Our base 25% idle power estimate utilized thus far is intended to account not only for leakage power (which is very low at our assumed .18μm technology), but also for non–aggressive clock gating, clock tree distribution, and other such factors. As such, our actual idle energy estimates thus far have actually been based on a summation of smaller estimates for some of these individual components, and varies from our 25% approximaiton on an component by component basis. In this evaluation, we assume near ideal clock gating, and further essentially assume that the clock tree itself is gated for portions of the design that are not in use. Thus, this tradeoff analysis attempts to focus on the effect of leakage only, and may not correspond precisely with data previously presented.

The key point in this figure is fairly obvious. At larger processing technologies where

leakage current and other contributors to idle (non–switching) energy dissipation are minimal, scaling the system to match realtime performance can provide a significantly larger (1.5–2x) reduction in overall workload energy. As we move to increased leakage current this benefit fades until (in this estimate) in the range of .13μm it is better to simply process the data as fast as possible and then halt into a low power, V_{DD} gated standby state. The reason for this tradeoff is that the realtime goal remains the same. The primary benefit of smaller processing technologies (improved performance) is not relevant if a slower system is able to match the desired performance model. Thus, the precise tradeoff point in this analysis will certainly vary with speech recognizer configuration and usage. Furthermore, the lower V_{DD} max at smaller technologies may also have a contribution to actual energy dissipation. That is to say this analysis only looks at the tradeoff of voltage scaling versus standby halting, not the potential energy tradeoff of moving to a smaller technology. The intuition to be gained here, however, is that the choice between a straightforward "hurry up and wait" approach as compared to a "slow down to match the goal" approach is very dependent on the overall system level effects of these individual designs

8.4 Conclusions

This chapter has considered a number of extended optimizations beyond the base architecture presented and explored earlier. We begin by considering use of the function call semantics of our thread management infrastructure to throttle concurrency exploitation based on system bottlenecks, particularly memory bandwidth bottlenecks. We find that such a system is quite effective at mitigating performance loss due to such bandwidth constraints, but in general conclude that it is better to design a tuned system than to provide extra resources that are not regularly used.

We continue our evaluation by considering the benefits of data compression. We focus on compression of the static knowledge base data found in this application domain. Compression in the memory system proper has obvious benefits, so we instead consider maintaining compressed data at the L2 on–chip cache level. We find that, due to the data distribution

and access patterns of the application, maintaining compressed data in the L2 shows no performance or energy benefits. Decompression latency in the off–chip memory system, however, has negligible effect on overall performance, suggesting that the best solution is to compress data in memory utilizing compression techniques optimized to each individual knowledge base, and including a decompression engine in the data module (for example, a plug–in ROM chip) itself.

Finally, we consider potential power management techniques to trade off excess performance for reduced energy consumption. We find that both low–power standby during idle cycles and dynamic voltage and frequency scaling are quite effective at reducing overall workload energy, with voltage scaling being considerably more effective at larger processing technologies. Leakage dissipation analysis suggests, however, that at newer processing technologies, this tradeoff may shift to better energy savings through leakage reducing standby states.

CHAPTER 9

Summary and Conclusions

This work considers the potential of continuous, real–time speech recognition on low power and handheld devices. Modern large vocabulary speech recognition programs utilize advanced stochastic search techniques to probabilistically explore potential recognition results in relation to models of common linguistic and acoustic patterns. The resulting computational complexity and memory throughput demands can stress even high end systems. Thus, achieving realtime speech recognition within the energy constraints of a low–power device will require exploitation of specific characteristics of the domain to improve efficiency.

This exploration begins by considering the properties of modern speech recognition software that may lead to poor performance or inefficient execution. A copy of the CMU Sphinx speech recognition engine, version 2, is utilized to perform this evaluation. The key constraint to performance is found to be latencies incurred accessing the large quantities of knowledge base and training data inherent in this domain. These access patterns have very stream–oriented characteristics, and violate many of the assumptions of spatial and temporal locality used to maximize performance on modern processors. Further analysis demonstrates the potential for large amounts of thread level concurrency, a common characteristic in applications that employ stochastic search techniques. We contend that efficient use of this concurrency can improve execution efficiency of speech recognition applications and achieve high performance goals on low power environments.

The potential for making use of this thread level concurrency is considered through a

progressive series of architectural models. The final implementation presented here follows a basic design philosophy in which the programmer exposes all available concurrency to the architecture through a series of basic primitive instructions, and the architecture manages this concurrency, making use of it as needed to match existing system bottlenecks. This programming model is achieved by a "thread spawn" interface modeled after a function call. Use of this interface signals to the architecture that the "called" code may be executed concurrently with other code on the system. A special ID flag may be passed with this call to allow fine grain mutual exclusion of code segments operating on the same data, transparently eliminating race conditions. The function call model for exposing concurrency allows the architecture to decide at execution time whether a new thread should be spawned to handle the workload, or whether the request should be converted to a standard system call and executed without exposing concurrency. The inherent characteristics of the speech recognition domain make this programming model particularly effective.

A hybrid multi-threaded chip multiprocessor architecture is proposed to support this design model. This is configured to operate as a coprocessor attached to a standard general purpose embedded processor (in these evaluations, a 400MHz Intel XScale). This approach utilizes multiple concurrently executing processor pipelines to provide the raw resources to take advantage of domain concurrency, and utilizes multiple hardware thread contexts on each processing element to maximize pipeline utilization in light of potentially long memory latencies (due to program characteristics). Each pipeline itself consists of minimal logic and equally minimal support structures, utilizing far less energy than the main general purpose processor to complete a given set of operations.

Preliminary evaluations of this architectural model are performed in an idealized environment modeling significant memory latencies, but with unlimited memory bandwidth. A copy of Sphinx2 was hand parallelized to expose concurrency utilizing our base primitives. Initial evaluations show that in order to minimize communication and coordination overhead, and to maximize processor utilization, it is necessary to utilize an initial static workload partitioning (done offline) and a dynamic load balancing infrastructure. In this

environment, the described architectural model demonstrates near ideal performance improvements as pipelines are added. Multiple hardware thread contexts are also shown to be quite effective at hiding fixed system latencies, maintaining high processor utilization even as memory request latency is varied across a wide range. Furthermore, due to dramatic reductions in execution time, overall workload energy dissipation does not increase, and often decreases, despite the added computation resources.

Having established the potential of this architectural approach, memory system effects are considered. Directly attaching this architecture to a standard 100MHz SDRAM memory system eliminates all of the performance gains seen in the idealized evaluation. While the system is able to tolerate relatively long latencies for memory requests, the bandwidth bottleneck imposed by a realistic memory implementation severely constrains performance. This necessitates an exploration of memory system configurations to provide the necessary bandwidth to keep processor utilization high. It is found that a multi–stage caching approach designed to capture the metadata component of the program working set is quite effective at reducing demands on the memory system. Program knowledge base data, however, is far too stream oriented in nature to be cached effectively, particularly as knowledge base sizes grow. Alternate, lower power storage systems making use of flash and ROM technology are also considered, and resulting data suggests that added request latency of such systems is irrelevant if techniques are employed to provide necessary bandwidth. From this memory system evaluation, a number of configurations that achieve realtime performance for this speech recognizer configuration are determined.

We utilize this data to explore a few more potential optimization avenues. First, the potential for throttling concurrency exploitation based on the currently available memory bandwidth is considered. While this optimization has no effect on configurations that do not max out available bandwidth, it is found to have a substantial positive effect on configurations that do. This performance benefit comes from minimizing the pile–up of requests at the memory interface, allowing the entire system to operate more smoothly.

This work further consider the potential of utilizing data compression techniques to

reduce the overall memory footprint of these applications. Static knowledge base data is made the focus of these efforts, as it represents the largest component of overall knowledge base memory footprint, and eliminates the need for runtime re–compression. Results show that that storing compressed data in the L2 cache has minimal positive effects on program performance, as the application working set remains large and data access characteristics make little use of the added effective space provided by such compression. Significant decompression delays in the memory system proper, however, are well tolerated by this architectural model, suggesting substantial memory space benefits from data compression transparent to the processor core. Thus, while the performance benefits may be minimal, costs associated with storage of potentially hundreds of megabytes of data can be greatly mitigated. To complete this analysis, bandwidth effects of compressed data are also considered. While mutable data compression may have a significant effect on overall bandwidth, immutable data actually represents only a fraction of actual bus allocation time, and thus such compression has only a minimal effect.

As a final evaluation, potential modes of power management, taking advantage of beyond–realtime performance to offset overall energy consumption, are considered. This considers two primary models: processing quickly and then entering a low–power standby state until the next realtime interval is reached (and corresponding input elements become available), and dynamic frequency and voltage scaling to slow down the processor to match realtime performance goals. Both methods may be quite effective at reducing overall workload energy consumption, however frequency scaling appears to provide a considerably greater benefit for the processing technology characteristics considered here. A final first level leakage current evaluation suggests that, as systems move to smaller technology implementations in which leakage power is a greater percentage of overall power consumption, this tradeoff shifts toward finishing the required workload quickly and entering a low power, low–leakage state.

A few summarizing views of the data discussed in this work are presented in Figures 9.1 to 9.4. All of these figures consider 2–4 processor and 2–4 context configurations, beginning

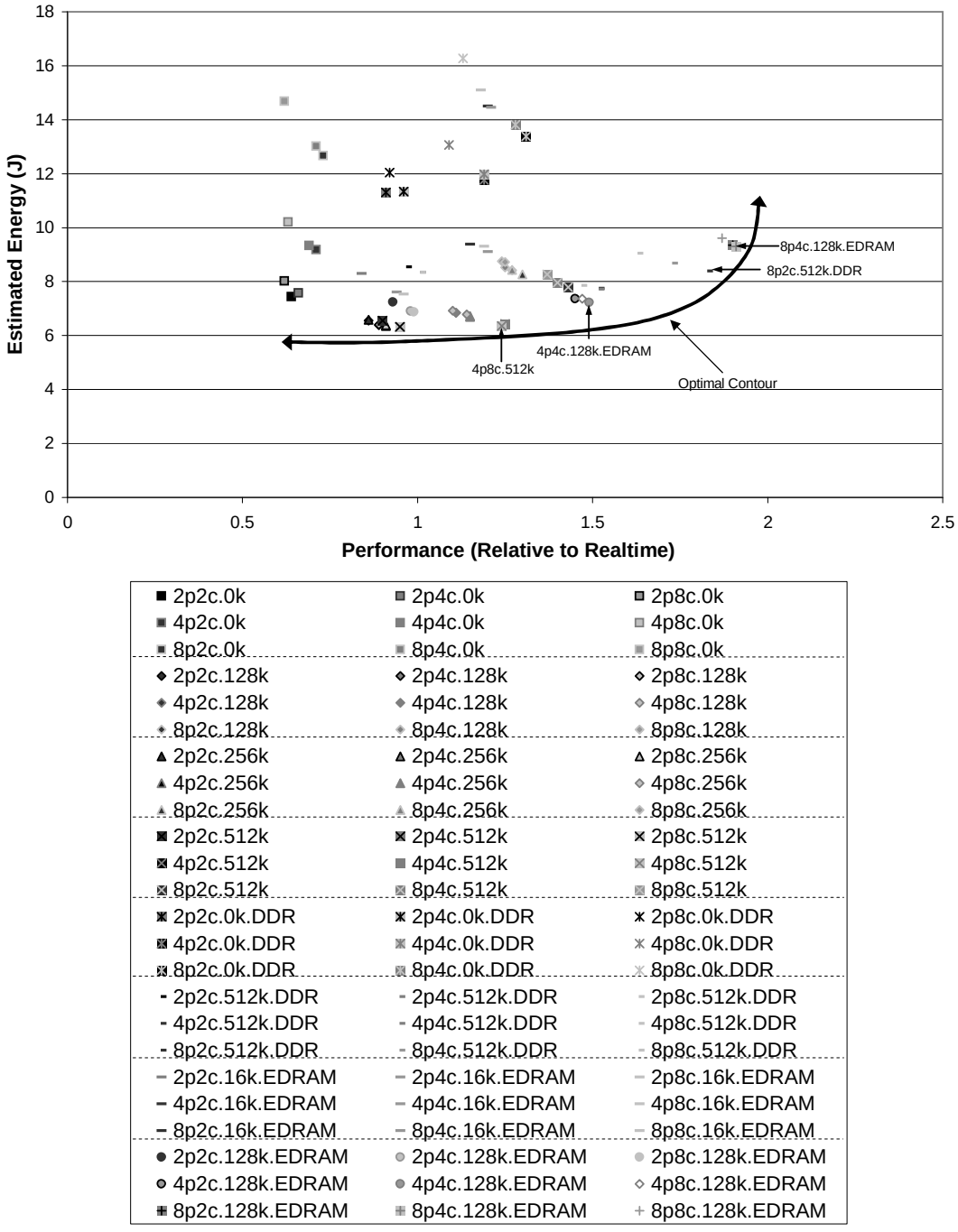


Figure 9.1: Summary Energy vs. Performance - This figure depicts workload energy consumption relative to achieved performance for an array of processor, context, and memory system configurations considered in our evaluation. As "realtime" is a variable metric in this domain, we include configurations that did not achieve realtime in our study in order to cover the scope of possible systems. Key values represent (processor config).(L2 size).(DDR or EDRAM). The optimal contour (lower–right) by this metric generally favors EDRAM and large L2 based system.

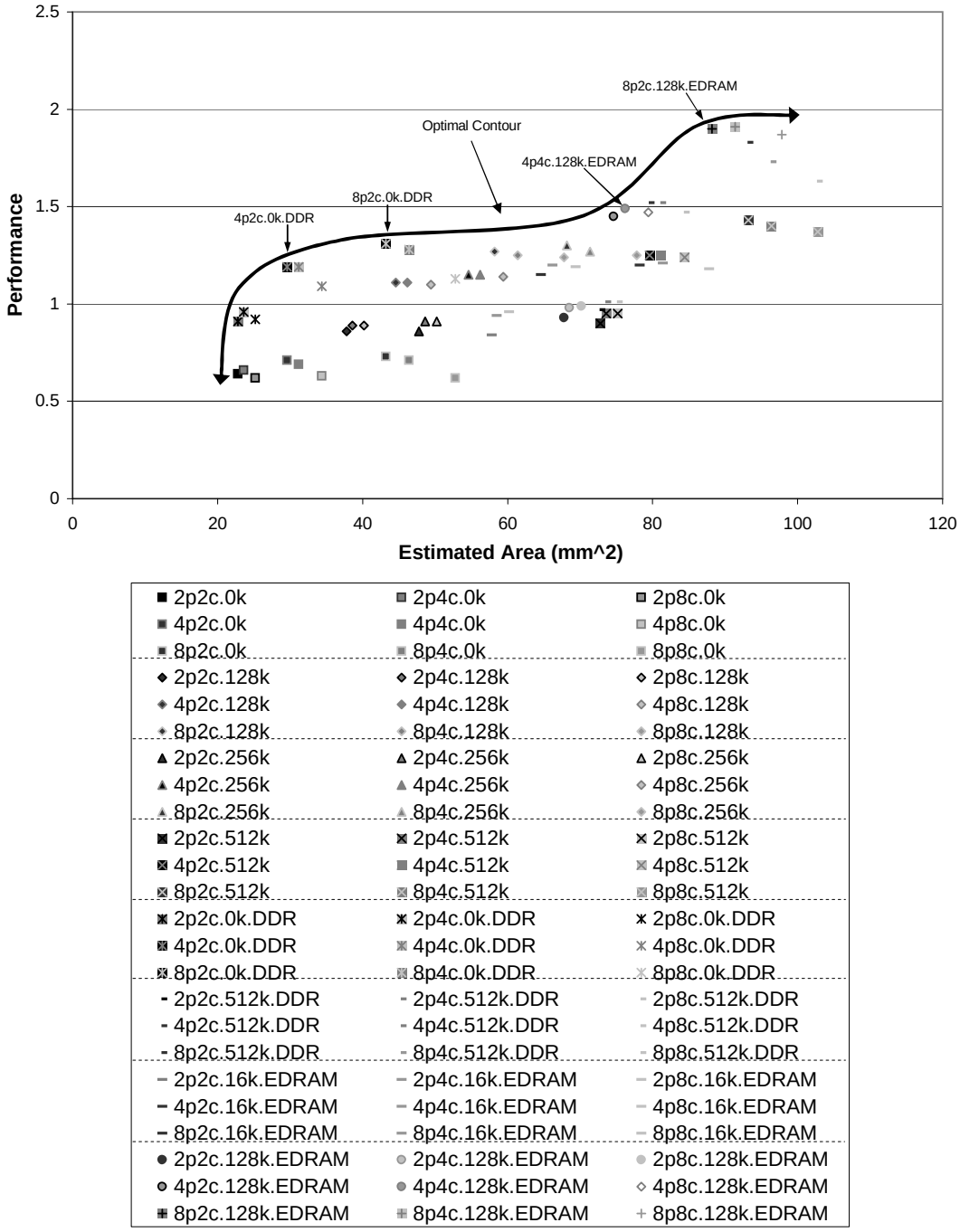


Figure 9.2: Summary Performance vs. Chip Area - This figure depicts performance relative to realtime versus a chip area estimate based on data from our energy estimation framework. The optimal contour by this metric (upper–left) favors DDR systems because the added cost of DDR does not enter our on–chip area model. EDRAM based systems with L2 caches are also seen due to improved performance.

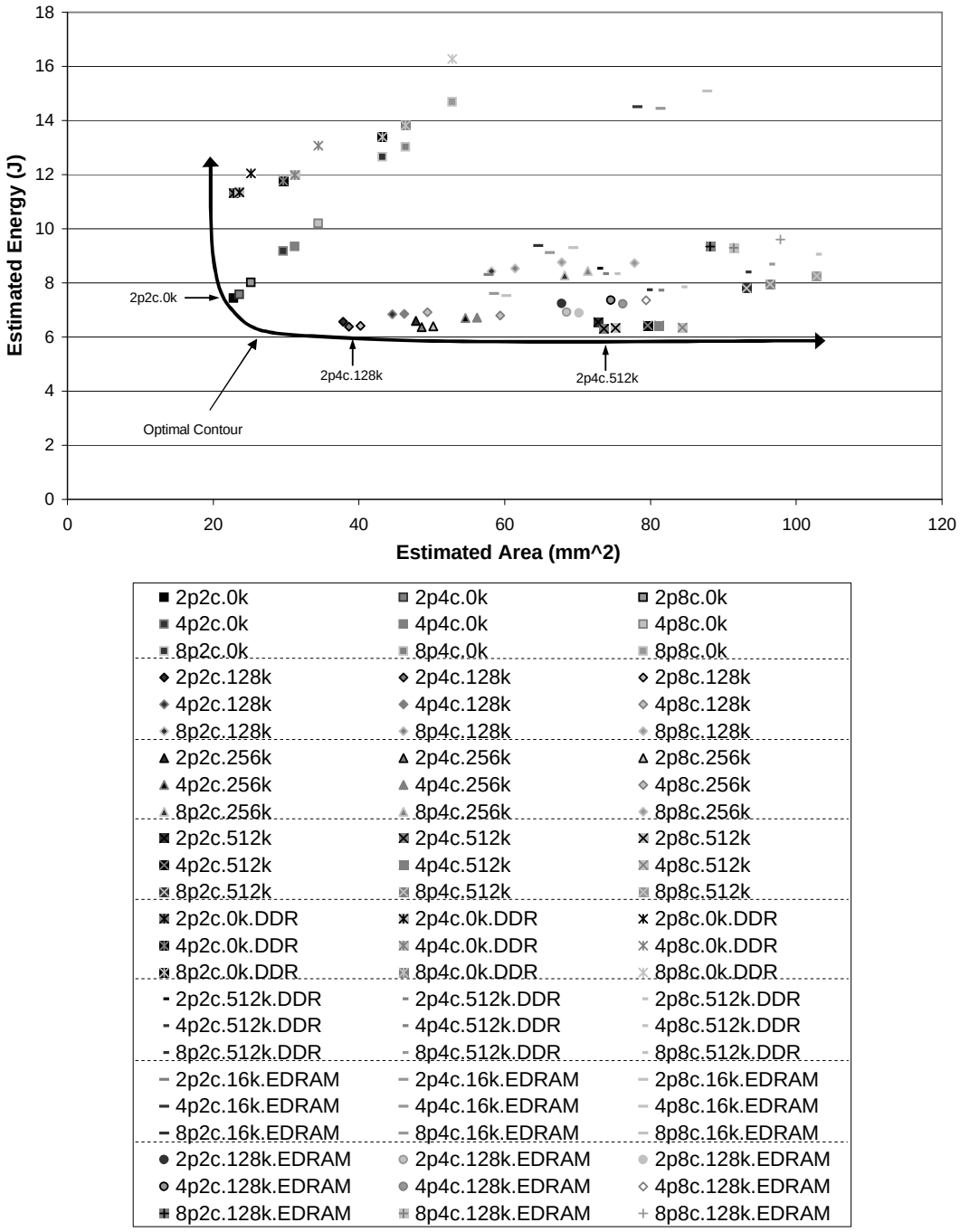


Figure 9.3: Summary Energy vs. Chip Area - This figure depicts an estimate of workload energy consumption versus estimated chip area (as described previously). The optimal contour here (lower–left) favors smaller processor / context configurations with small L2 caches.

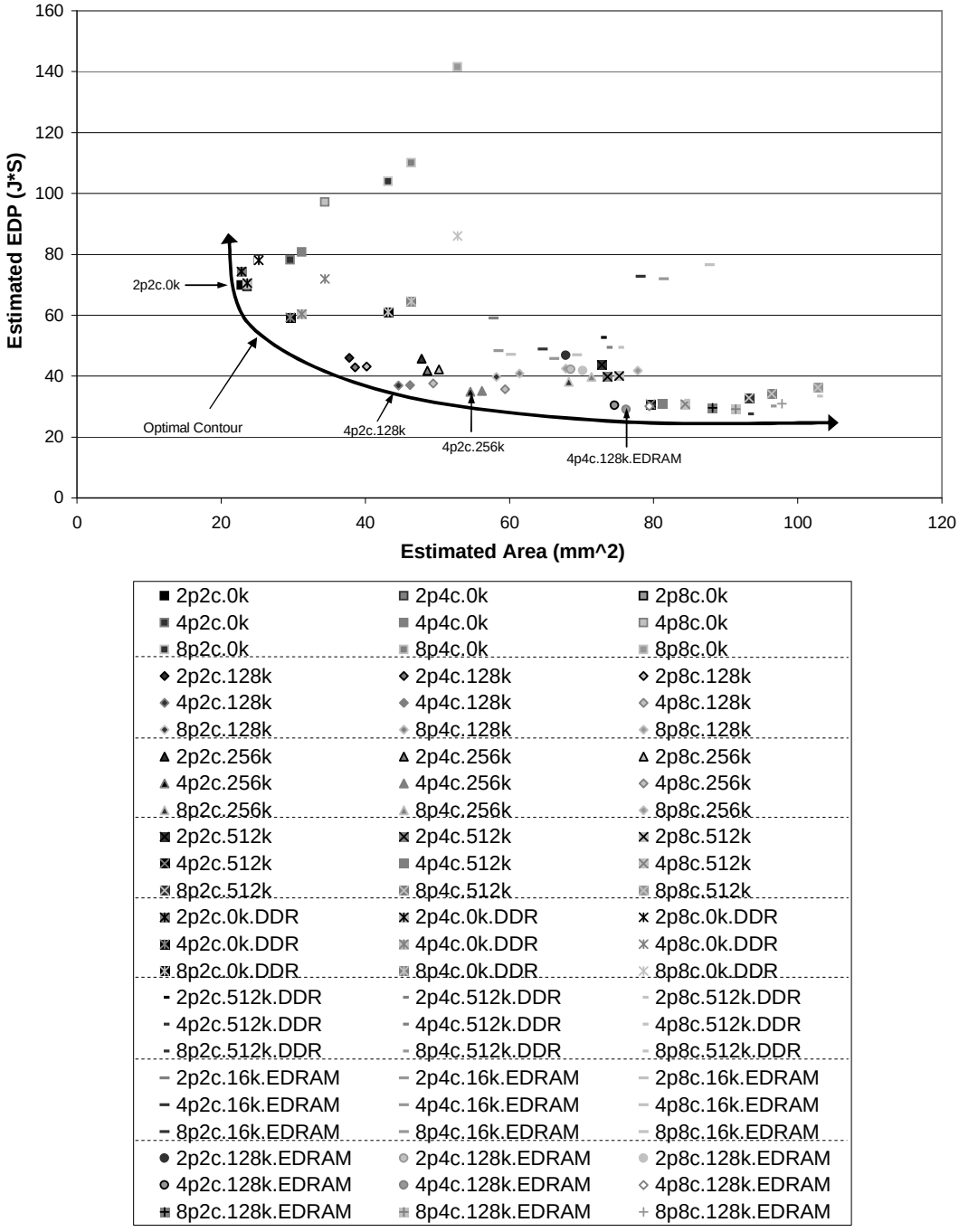


Figure 9.4: Summary EDP vs. Chip Area - This figure depicts estimated EDP versus estimated chip area (as previously described). This is likely the most telling of our summary evaluations, as it considers performance, energy, and cost. The optimal design contour here (lower–left) generally favors L2 based systems.

with a base SDRAM system and moving through a number of L2 cache sizes, the inclusion of DDR memory, and the inclusion of EDRAM (without DDR memory). Figure 9.1 relates workload energy consumption to performance (relative to realtime). Note that systems that did not meet realtime in this evaluation are included because the computation required to achieve realtime can vary substantially in this domain. For the same reason, power management techniques to match realtime performance are also not considered. In this figure, optimal design points are those that achieve higher performance with lower energy consumption, which tends to exclude configurations with poor memory system bandwidth.

Figure 9.2 considers performance relative to an estimate of on–chip area based on data generated for our energy estimation model in Appendix B. This area estimate is used as a proxy estimation of the relative "cost" of individual designs. In this figure, more optimal configurations are those that achieve the highest performance for the lowest area (the "upper–left" contour). Figures 9.3 and 9.4 similarly relate energy and EDP to this area estimate. Of these, the EDP analysis is likely the most telling, as it incorporates performance and power as well. The optimal contour on this graph follows the lower–left surface, generally includes systems with around 4 processors, a standard SDRAM memory system, and an L2 cache, and generally does not include more sophisticated memory systems or configurations with 8 contexts. This tracks well with the intuition developed in earlier chapters. While the specific optimal configuration will depend on the performance necessary to achieve realtime performance goals, these figures provide a general summary of where the performance / energy / cost tradeoffs are for this domain.

Note that this summary evaluation does not consider flash and ROM systems. The area component of these devices can not be easily determined, as the required area is likely to vary with the constraints of current speech recognition technology. Furthermore, choice of memory system components in an actual design will be dependent on the total workload intended to be run, not just a speech recognition task. We note, however, that a flash component will be notably more expensive in monetary cost than a DRAM component, and the design and verification cost of a ROM based system may be higher still.

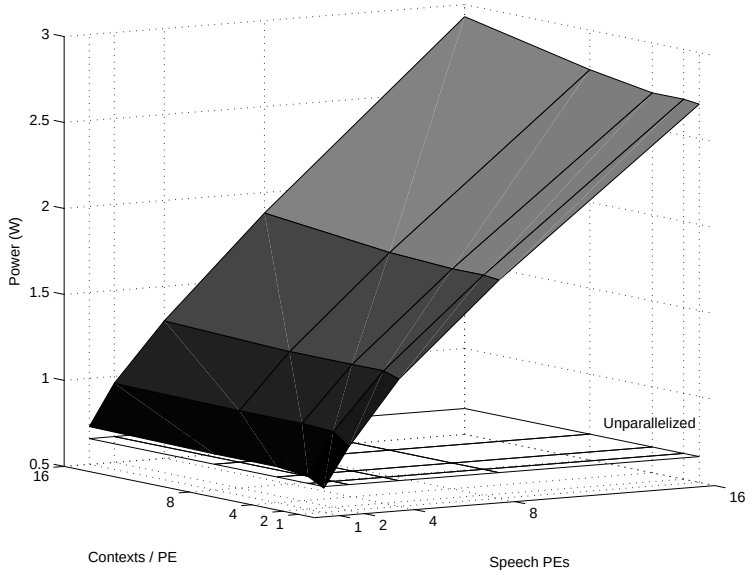


Figure 9.5: Instantaneous Power Estimate - This figure depicts estimated instantaneous power consumption for a range of processor / context configurations assuming a 128k L2 cache and standard SDRAM memory system.

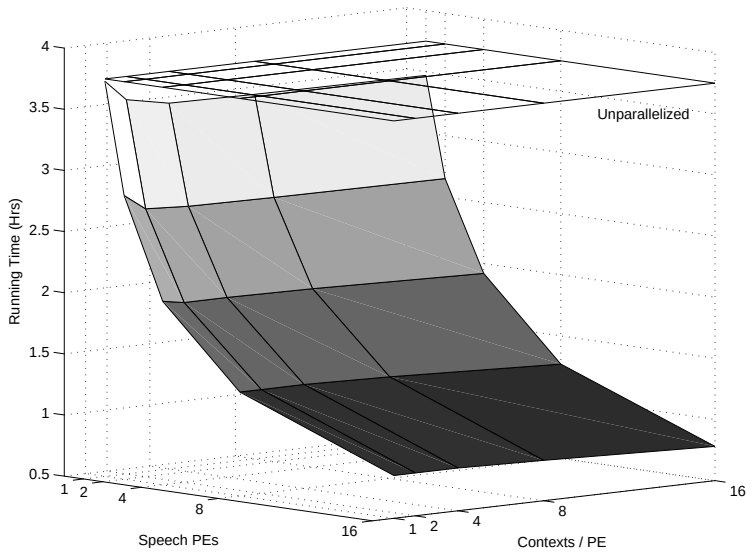


Figure 9.6: Battery Runtime Estimate - This figure depicts estimated total runtime for various configurations assuming a 128k L2 cache and standard SDRAM memory system. These estimates are based delivery characteristics of a single 1600 ma·hr AA battery.

Finally, we consider a high level approximation of instantaneous power consumption and estimated battery life in Figures 9.5 and 9.6 respectively. Once again, our estimate of power consumption and battery lifetime for a standard XScale based system is included as a relative reference point. Runtime estimates are based on lifetime on a single 1600 mA·hr rechargeable AA battery. These figure, taken together and in context with the other metrics provided in this work, suggest that our "sweet spot" designs of 4 processors achieve 3–4x the performance of the base system at around 2x the power dissipation (and consequently half the runtime, around 2 hours in Figure 9.6).

9.1 Future Directions

While this work provides significant intuition into techniques for achieving low–power continuous speech recognition, there are a number of areas remaining to be evaluated. The most obvious of these is memory system effects and constraints across generations of speech recognition systems. As speech recognition is itself an evolving field, these dynamics have the potential to change drastically over time. For example, over the course of this work the CMU Speech group has moved to the Sphinx3 recognition library. This system uses continuous Gaussian models to score input sequences, resulting in considerably greater computational complexity in that phase of the search process. Furthermore, as knowledge bases and training data sets grow, the memory bandwidth problem discussed in this work will only become worse.

While we begin to explore a few potential solutions to future problems with on–package / independent knowledge base streams and embedded DRAM technologies, a more thorough accounting is left to future analysis. The work presented here suggests that our base architecture and programming model will continue to provide impressive performance on future generation speech recognition systems *if* the necessary memory bandwidth can be provided.

More aggressive and domain specific approaches may also be called for. Our approach attempts to maintain a number of general purpose characteristics in the handling and

processing of data in recognition of the continued growth of the speech recognition field as a whole. For example, though not designed for it, the HMM search component of Sphinx3 can be *easily* adapted to our design. Other components of speech recognition, however, may be more amenable to computation specific designs such as the Gaussian Scoring ASIC design by Mathew et al [65]. A more direct use of program metadata to allow the architecture to perform operations such as informed prefetching may also become more useful in dealing with the streaming nature of references to much of the application knowledge base data.

In short, this work opens the door to a range of potential future enhancements. Furthermore, we provide a computational model and infrastructure that handles the demands of this application space quite effectively. The key problem with providing low–power large vocabulary continuous speech recognition is, and will likely continue to be memory system performance. This is likely to be the focus of future efforts in this area.

APPENDICES

APPENDIX A

Experimental Infrastructure

Much of the data presented throughout this work is based on simulated runs of the CMU-Sphinx2 speech recognition engine. Through the course of development of the various architectural and programming models presented in this work, the simulation and experimentation infrastructure evolve in detail and sophistication as well. Any simulation environment that utilizes approximations of true hardware components for performance reasons, however, is bound to incur errors. A full accounting of the features and capabilities of the experimental infrastructure is important to determining the validity and generalizability of the results presented. This chapter attempts to provide such details. We first present details of the Sphinx knowledge base configuration and execution parameters used in all parallel architectures in this work. We will then present a detailed analysis of the final simulation infrastructure, making particular note of optimizations and simplifications that may affect the accuracy of our results.

A.1 Sphinx Configuration

During the initial characterization of the Sphinx speech recognition system, we explore an array of basic memory configurations, as well as variations in speech recognition knowledge base, search parameters, and input types. Due to simulation overhead, a full exploration of sphinx parameters is not practical during the architectural evaluation phase

of this work. Rather, we select a single recognizer configuration for use throughout the work.

The selection of configuration was affect by a number of factors. First and foremost, the configuration must be true to the current and potential future characteristics of speech recognition. Thus, a 200 word vocabulary would not be reasonable. Secondly, we wish to explore a wide array of architectural and memory system variations, requiring extensive labeling of program data and equally extensive simulation efforts. Thus, we wish to select a configuration that results in reasonable data export properties and simulation time. Based on this set of factors, we settle on a language model based on a 11440 word vocabulary. This is large enough to expose the properties of the application, but small enough to lead to reasonable simulation performance.

The knowledge base itself is built from input sentences taken from famous speeches and selected text excepts from Project Gutenberg. This set of English sequences is passed through the Cambridge Statistical Modeling toolkit [22], producing a trigram language model. A list of specific dictionary words is extracted independently from the full CMU dictionary (v0.6).

All analysis is performed utilizing the Sphinx2 speech recognition library. We utilize default recognizer configurations for search beam threshold and pruning characteristics. We also evaluate only the forward lexical tree search algorithm, disabling flat model analysis and the post–processing bestpath analysis. Note that, as previously described, the flat algorithm shows similar characteristics to the lexical tree approach, and this lexical tree is more practical for speech recognition. A set of initial static partitions for various processor configurations is generated by exporting profile–annotated knowledge base information and utilizing the HMetis graph partitioning toolset [42]. A set of speech inputs were evaluated to verify that the choice of input sequence did not alter the relative performance characteristics of various design options, and the speech input that resulted in worst raw performance was selected for use in subsequent evaluations. Each simulation result is based on 600 frames of speech input, starting after the first 200. This allows the recognizer itself to

"warm up", allowing us to evaluate the steady state operation. While this corresponds to only 6 seconds of speech input, it covers an array of per iteration active node counts, and is sufficient to expose the primary characteristics of the application. Under this set of configuration parameters, simulation runs required 6 – 30 hours on a 3GHz Pentium 4 processor, depending on processing element count / configuration.

A.2 Simulation Infrastructure

The simulation environment utilized in this work is based on a modified version of the SimpleScalar Toolset [6]. The basic in–order processor model was modified and extended to model processor and system characteristics in significantly greater detail. The memory system delay model utilized in non–ideal memory delay simulations is provided by Wang and Jacob [96]. This section will detail modifications made to the simulator to increase accuracy, and attempt to account for potential areas of inaccuracy. Specific details regarding the memory system model itself are left to cited works.

A.2.1 XScale Modeling

The main XScale processor is modeled as a 7–stage, in–order execution core with a bi-modal, 128 entry BTB branch predictor and 32k, 32way set associative data and instruction caches. The basic simulator execution core is modified to accurately model branch resolution in the execute stage (introducing stall cycles to account for pipeline flush and restart). Pipeline related latencies are also introduced on all processor restarts (from long latency stalls or explicit idle time in the instruction flow such as while waiting for speech coprocessor completion). Instruction latencies and inter–instruction dependencies are tracked to allow scheduling of instructions on different functional units such that instruction complete in–order and meet necessary dependencies. Thus, a dependent add following a long latency multiply will be held in issue until dependencies could be satisfied by a hypothetical bypass network. A non-dependent add would be held until it's completion time would occur immediately after the multiply, maintaining in–order properties.

There are a number of points where this modeling diverges from the actual XScale processor model. Most relevant is the assumption of an FPA style floating point unit, such as those available as coprocessors on ARM SA-1110 [3] systems. Beyond this, our implementation does not model a number of the nuances of the XScale pipeline. For example, all delays are based on a 7 stage pipeline, while the XScale processor utilizes a 8th stage for memory operations. We also do not account for hazards incurred by the split shifter/ALU configuration in the XScale, nor do we explicitly account for run–ahead effects of the two stage fetch engine or deferred register dependency stalls within the bypass network. It should be noted that simplification of these nuances is only likely to have any effect on unparallelized performance estimates, as the amount of work performed by the XScale processor in parallel configurations is minimal.

Cache and memory system modeling is based off of standard SimpleScalar cache models. Instruction and data cache hits are assumed to take a single cycle, and do not affect pipeline flow (note again that the actual XScale processor includes an extra pipeline stage for memory operations). Non–blocking caches are assumed, with the ability to buffer up to 8 outstanding read requests. While the actual XScale provides an 8-entry writeback buffer, we assume essentially infinite writeback buffers (though system contention due to writes are accounted for, as discussed elsewhere). Furthermore, the processor is stalled upon issue of the 9th outstanding read request (not when the read buffers are full). In practice, dependency constraints stall the processor well before this point. The SimpleScalar cache model itself is modified to accommodate unknown memory latencies (necessary for accurate modeling of modern DRAM systems with request reordering), and perform all state updates on request completion (as opposed the original state modifications on request submission). Note that the XScale processor maintains two 2KB mini-caches, which are not modeled here. Instruction and data TLBs are also not modeled.

The final significant divergence from a base XScale processor is the inclusion of a second "background processing" context. As discussed, this context is used to perform data preparation tasks for subsequent frames of execution while the speech coprocessor component is

operating on the current frame. Utilizing a second independent context allows the main processor to return to the main parallelism generating execution thread on an interrupt signal from the speech coprocessor, maximizing the overlap of computation. Our analysis suggests that this background thread could be entirely virtualized with minimum hardware support due to low register pressure. In order to simplify simulation, however, we maintain data for this thread in a full replica of the main processor register file. Program code is constrained at compilation time to use half the available register file, and latency is introduced in the simulation environment to switch between the main and background context. This effectively account for the performance effects of this optimization.

A.2.2 Speech Processor Modeling

The simulation model of the base execution core of a speech coprocessor element is nearly identical to that of the main XScale processor. The main differences are the assumption of a 5–stage pipeline model with branch resolution in execute, and four MSHR equivalent memory request buffering components per thread context. In order to improve simulator efficiency, outstanding memory request tracking in performed in an interface layer in the memory system model, and the processor only tracks it's number of currently outstanding memory requests. Otherwise, the execution model for these processing elements matches that already described for the XScale processor, executing beyond cache misses until dependencies, stalling instructions as necessary to ensure in–order commit (except for load misses), and introducing stall cycles to account for pipeline bubbles. We once again assume an infinite writeback buffer, but account for latencies and bottlenecks introduced by writebacks at the memory system level.

The primary functionality present in the speech processor pipelines that is not found in the XScale is multiple execution contexts utilized to tolerate latencies. While the main processor is given two execution contexts, a switch from the main context to the background contexts is performed by an explicit instruction, and return of control to the main context is performed on an interrupt from the speech coprocessor system indicating the completion

of a parallel section. By contrast, contexts in the speech processing elements are activated and deactivated in response to system latencies. In order to model this accurately, latencies incurred during execution of an instruction are divided into two categories: maskable, and non–maskable. Maskable latencies include those that could potentially involve a substantial number of stall cycles, and could be identified during execution. For example, execution of an instruction that has a register dependency with a currently outstanding load would be considered a maskable delay. Non-maskable delays are predominantly related to execution hazards such as branch misprediction.

In the case of a maskable latency (currently unknown or otherwise), a check is performed for a ready context from the pool of currently available contexts. It is assumed that this check can be performed in the same cycle the maskable latency is discovered. As this check could be an ongoing operation of a small component of control logic, this is not an unreasonable assumption. If another context is active and ready to execute, the processor is configured to begin executing from that new context on the equivalent of the next cycle. The result in terms of instruction throughput is that the latency incurring instruction in the first context is considered flushed from the pipeline, and sufficient stall cycles are introduced to account for pipeline stages already traversed by that instruction. So, if a register dependency violation is discovered in the RF stage, the result would be two stall cycles between the last instruction of the first context and the first instruction of the second context. Thus, the remaining cycles needed before the first context is ready to execute again may potentially be hidden behind execution of the second. If a second execution context is not available, the processor is stalled with a special flag to indicate that is is awaiting maskable latencies. Thus, events that can provide new work for the pipeline (such as a migrated job), can begin executing immediately (with latencies for their own execution appropriately accounted for).

By contrast, non-maskable delays are exposed in their entirety. An obvious example of such a delay is a branch misprediction. In order to minimize the need for physical resources, speech coprocessor pipelines assume a branch not taken prediction scheme. Actual branch resolution is assumed to take place in the execute stage. Even if the processor were to

choose to switch contexts, it would not realize the need to do so until the branch instruction reached that stage. As such, the introduction of three stall cycles in inevitable. Generally, delays needed to insure in–order commit or serial functional unit execution also fall into this category.

It should be clear that the goal of this modeling system is to simulate the behavior of a multi-stage pipelined processor without actually tracking instructions in explicit pipeline stages. With appropriate latency metrics, and the use of these latency categories, we believe the resulting observed stream of completed instruction should track quite well with a more detailed micro-architectural implementation.

A.2.3 Communication Modeling

This architecture consists of two independent communication networks that must be modeled. The first is the control bus used to issue commands to the speech coprocessors, migrate jobs, and pass other signal and status information. The second is the memory network, which consists of the bus interfacing with the memory system proper, and possibly the bus connecting speech processors to an L2.

The control bus is modeled as a simple resource latency. Operations utilizing the control bus are designed to issue the communication request at the time the would first attempt to. The bus then returns a delay corresponding to the point when the message would have been completely sent. This completion time is equal to the next cycle at which the communication bus is "free" plus the number of cycles required to send the request (modeling an 8–bit bus with 2 cycle protocol overhead). Subsequent transmissions are simply stacked at subsequent free intervals. The largest model error between this and a real system is the conflict free, FIFO nature of transmission. In reality, a component wishing to use the bus would need to wait until the resource was free and then follow some protocol to take control of the bus and perform it's transmission. The extra latency for this synchronization and conflict resolution is hidden in the simplicity of the bus model, but should be somewhat adjusted for by the two protocol overhead cycles associated with each communication. The important

aspect of bus communication, ensuring serial, non–overlapping utilization of the resource, is accounted for by this model.

The memory network is modeled somewhat similarly to the control network, but simulates a 64–bit bus infrastructure. If a L2 cache is present, the bus between the L1 and L2 caches is modeled identically to the control bus. In the absence of an L2 (or from the back of the L2 to the memory system), modeling is modified slightly to account for the lower clock rate of the memory interface. As mentioned previously, we make use of a memory system model by Wang and Jacob [96]. This memory system model contains an internal bus model for communication to and from the memory system proper. In situations where we are only utilizing a standard DRAM memory system, requests into the memory space utilize this bus directly and all associated delays and latencies are accounted for. This straightforward approach is not feasible when utilizing flash and ROM based memory systems, because these models are achieved by re-directing reference streams for particular address ranges to other memory latency models. In order to account for latency in these situations, we return to a basic resource utilization model as with our other bus models. In this instance, latency is incurred when the request is issued into the memory system, and when the data is returned. This approach is verified against the standard SDRAM bus system, and we find good correlation in overall performance metrics under both constraint mechanisms. In either case, these systems fully account for resource utilization, ensuring that bandwidth is limited to available memory bus cycles and no two communication events occur simultaneously.

A.2.4 Memory System Modeling

While we utilize an external memory latency model for the DRAM component of our memory system simulation, there are a number aspects to the memory system beyond the SDRAM. The two of particular relevance are our address remapping infrastructure, and latency model for flash and ROM components, both of which are discussed here. Note that cache modeling was discussed in our main processor description above.

Address Remapping

As discussed in our Sphinx description, data segments of the application are exported to the simulator for the purpose of isolating specific data streams and properties. This allows us to perform dynamic data address remapping without modifying the properties of the base program. This is done by building up a mapping table of data addresses as the data is exported by the software into the simulator. This mapping table is then referenced when memory operations occur to determine the translated address. The original data address is used to reference real program data in the virtual memory space, while the translated address is passed to the memory latency model (caches, etc).

The primary uses of this data remapping capability come from the use of translated addresses to identify properties of the data stream. For example, access to static data components must be identified at the memory system model (below the cache level) in order to perform split data stream simulations such as those involving flash and ROM based static knowledge base storage. This is achieved by remapping static data into a high, otherwise unused address range. As the memory latency model only sees translated addresses, stream partitioning becomes trivial. Another use for this address translation mechanism is found in our compressed memory model estimates. In this case, the remapping infrastructure maintains both a uncompressed translated address, and a compressed translated address. The uncompressed address is used by all levels of the memory hierarchy that, for a given architecture configuration, are supposed to operate on raw addresses, while the compressed address is utilized by all levels that operate on the compressed data.

It should be noted that the obvious use for this capability is in exploring the potential for data placement optimizations. Such optimizations are not discussed in significant detail in this work, as they were found to be generally unsuccessful at making significant impacts on performance.

Flash and ROM Component

Modeling of flash and ROM components are performed by a much more straightforward means than DRAM modeling. Note that our basic assumptions regarding the flash and memory components make little use of factors that would affect access latency such as request reordering and open page usage. As such, these components are essentially modeled as a resource with a utilization latency and an associated "next free cycle" identifier. Upon a request to the flash model, the latency returned corresponds to the next free cycle of the selected flash bank (identified by data address) plus the latency of performing the request itself. Contexts waiting on the data for such a request are released when the critical word is returned, and communication network time is allocated to transfer all of the required data. Note that the flash / ROM model does not take steps internally to prevent simultaneous data transfer from multiple independent banks, relying instead of the bus model to prevent such overlap.

APPENDIX B

Power Audit

We generate power estimates by combining power / energy information from a number of data sources. While this is not ideal, it is necessitated by the fact that we are attempting to perform full system power estimates across a wide variety of designs. As such, we must develop a way to generate respectable power/energy estimates quickly, despite variations in the underlying architecture.

We recognize that any method of power estimation based on multi–source data is likely to see significant error with respect to an actual running system. We believe, however, that a detailed power model infrastructure will lend sufficient credibility to individual energy estimates that relative energy evaluations between similarly estimated designs will reflect trends seen on actual hardware. We will therefore spend this chapter considering a detailed virtual hardware implementation of our design space, discussing how the runtime power consumption of individual components will be computed to predict final results.

As a final note, be aware that all equations presented here are in simplified form, and ignore many constants such as conversion factors between metric values (e.g. watts vs. milliwatts). This is done for simplicity and should not be taken to imply a lack of such conversions in our actual computations.

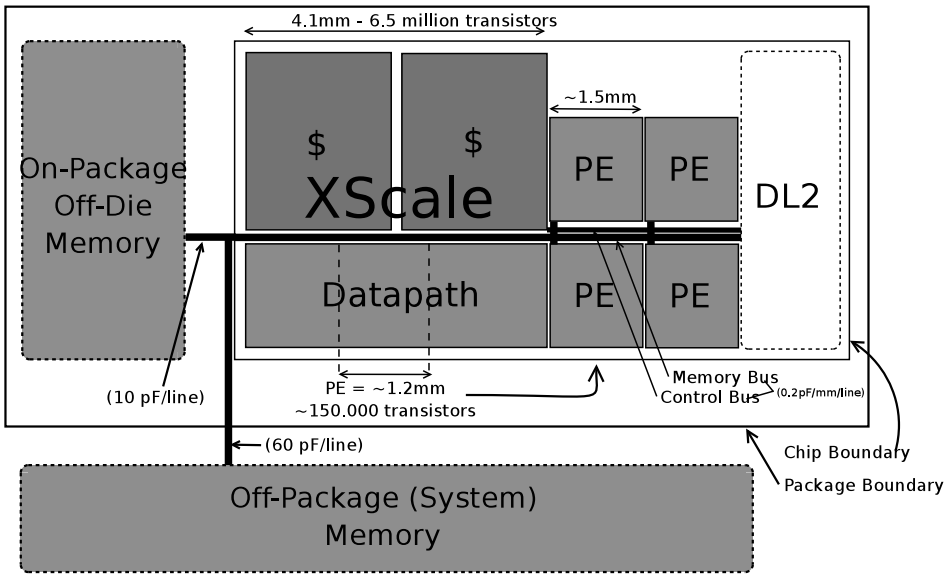


Figure B.1: Virtual Layout of System Architecture - This figure depicts a "virtual" processor layout, as well as some of the components considered in the derivation of our power estimation model.

B.1 Processor Level

The majority of our evaluations will assume a system–on–chip design philosophy, keeping the main processor and all components of the speech co–processor on the same die, or at least within the same packaging. This has the effect of drastically reducing communication and interconnect cost between the main processor and the co–processor, as well as amortizing pin counts and board space for such common facilities as the memory interface.

We will estimate processor power consumption by building up a virtual layout of processor components, and evaluating power consumption for all components. An overview of this virtual layout is shown in Figure B.1, and depicts an arrangement of a central communication network with the main processor on one end, global control logic on the other, and speech co-processor processing elements flanking the central communication system along the middle. In this case, we have depicted a processor design with 4 processing elements.

As is evident from this diagram, the key power components we recognize here are computation resources, cache resources, and communication resources. Together, these constitute a system–on–chip design. Existing methods of system on chip design generally involve

layout level power estimates based on standard cell layouts, or high level summations of individual component power estimates. In our environment, the standard cell approach would fall far short of an adequate estimation. The base XScale processor is itself designed with a careful eye to the overall energy constraints of embedded systems at the circuit level, and it seems reasonable to assume that a simple extension processor for a similar environment would be similarly constructed. By contrast, the often recognized failing of high level power summation based estimates for system–on–chip designs is the lack of consideration of system activity or software contributions to actual power dissipation.

Given these constraints, we will attempt to estimate component level power based on combined power modeling from a number of source. We will utilize design documentation for the XScale processor, and derive speech processing element power estimates based on area / component based scaling from the XScale design. We use similar derivation techniques to estimate bus, I/O, and memory configuration power. All of these individual power models will then be integrated into an activity based energy estimation framework.

B.1.1 Computation Resources

The computational resources of this design include the main XScale processor, the data path components of the speech co-processor pipelines, and the added register storage required to implement hardware contexts. The basis of our computational energy models is the design documentation for the XScale processor [2, 21]. Considerable effort in the design of this processor went into providing performance with minimal energy consumption. Thus, the XScale incorporates a number of advanced architectural features such as aggressive clock gating / banking in the cache, and careful datapath design. We believe that on–chip speech co–processor capabilities would likely use similar architectural and design techniques to minimize power consumption, and therefore choose to utilize conservative area–scaled power estimates from the XScale rather than develop independent pipeline models.

In order to appropriately attribute power dissipation to various components of our design, we begin by breaking down average power dissipation of the XScale processor by

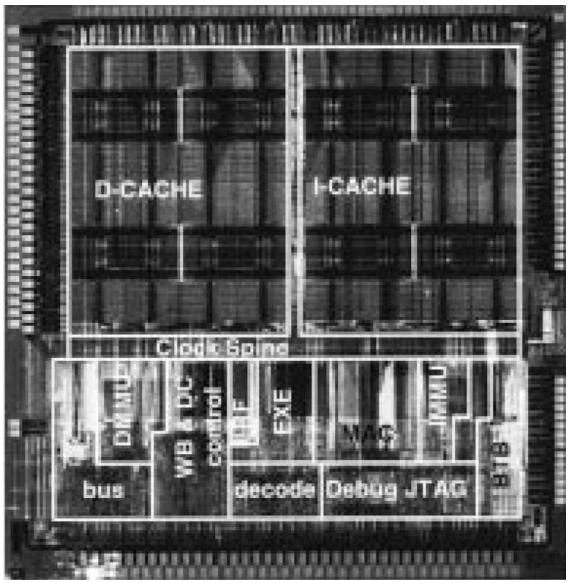


Figure B.2: XScale Die Photograph - This figure depicts a die photograph of the Intel XScale processor with a overlaid floorplan. It is taken from cited works [21]. In order to generate estimates in line with low–power architecture and processing technologies employed on the XScale, we generate power consumption figures for the datapath of co–processor elements through a conservative area scaling technique. Given the relatively lower complexity of processing element pipelines, we estimate a single context pipeline to be equivalent to the decode, execution, and a portion of the control logic area of the XScale processor.

components. We primarily want to generate an estimate for the power dissipation of the datapath component, as we will utilize this later in estimating the power dissipation of our added processing resources. We assume a typical active power dissipation for the entire chip of 450 mW @ 400 MHz, which is in line with a conservative (high) reading of power consumption estimates provided in the design documentation. The Intel PXA250 documentation rates typical power consumption at around 410mW, but includes and LCD display and some (minimal) memory communication. As the parameters for this evaluation involve a 98% icache hit rate and a 95% dcache hit rate, we discount the off–chip memory communication and attribute all energy consumed to the processor. By comparison, the base design documentation for the device [21] rates typical power consumption at 450mW, but at 600MHz (under similar program characteristics). Thus, an estimate of 450mW should be greater than the actual power consumption of the processor itself at 400MHz. This, in the end, should translate to a conservative estimate of processing element power dissipation, keeping with a general theme of targeting the higher end of power consumption when in doubt. In order to generate a reasonable datapath power estimate from this number, we will approach the estimate by three different methods, and consider how well they correlate.

Datasheet Based Estimate

Our first approach to estimating datapath power is based entirely on the base design documentation for the XScale device [21]. This document provides estimated percentage power distributions for a simulated run of the Drystone benchmark, which correlates well to the measured energy dissipation in their environment. They break down power by datapath, cache, clock, and control components. Based on our 450mW total power estimate, the contribution of each component according to the data sheet reference is as follows:

Component	Power
Datapath	121.5 mW
Cache	67.5 mW
Clock	103.5 mW
Control	121.5 mW

In our analysis, we wish to incorporate the power dissipated by the clock and control logic into either the datapath or cache component. As extensive clock gating techniques are applied to the cache in this environment, we believe a valid approach to distributing the clock and control component across the datapath and cache components is to distribute based on die area. In this case, the caches take up 60% of the chip area, while the datapath takes up 40%. Based on this distribution, we arrive at:

Component	Power
Datapath	217 mW
Cache	211 mW

Cache Energy Estimate

Our second approach to modeling component energy dissipation is to estimate energy for the cache structures utilizing the Cacti 3 [89] toolset, and subtracting this value from the total energy to arrive at a datapath estimate. The XScale design utilizes very aggressive cache clock gating and banking to reduce power consumption, resulting in cache design (not including the 2k minicache data) of 32KB, 32-way set associativity, with 32 banks, and 32 byte cache lines. The Cacti toolset is not able to generate an identical model, and furthermore does not assume the degree of gating and power mitigation utilized on the actual chip. Thus, we focus on Cacti models that match the physical cache area on the chip, and choose a base Cacti configuration of:

Parameter	Value
Size	32 KB
Associativity	2
Banks	16
Technology	.18μm

We find that variations in the number of banks in this range does not substantially alter individual bank power, and recognizing the type of power mitigation techniques used on the actual processor we favor designs that moderate routing power. Furthermore, as the actual XScale cache design targets the necessary cache line for activation through clock gating techniques, we utilize the bank access and total routing power from cacti to estimate the energy of a cache access (as opposed to the total energy to activate all cache banks). We generate this configuration for a .18um processing technology, and then perform voltage scaling down to the 1.3V voltage of the processor. Using the energy and access time values provided by Cacti to estimate power dissipation, we arrive at a value of:

$$P_{cache} \approx 400 \frac{\text{mW}}{\text{cache}}$$

We now assume 1/3 utilization of the dcache (a load/store every three instructions) and a 1/3 utilization of the icache (each cache line is eight instructions, so this assumes the cache is accessed once per instruction block, and that three instructions are used on average per block, which is likely quite low, but only adds to our error margin). We deduct the resulting power estimate from our total power estimate to arrive once again at a datapath estimate. Thus:

$$\begin{aligned} P_{caches} &\approx \frac{1}{3}P_{cache}(Icache) + \frac{1}{3}P_{cache}(Dcache) \\ P_{caches} &\approx 264\text{mW} \\ P_{datapath} &\approx P_{total} - P_{caches} \\ P_{datapath} &\approx 185\text{mW} \end{aligned}$$

These numbers compare reasonably with the 210mW (cache) and 216mW (datapath) estimates generated from the original documentation. This lends some further credence to the processing element datapath power estimates discussed shortly, and suggests that in this environment, power values generated by Cacti will, if anything, provide conservatively high estimates of actual achievable power dissipation for on–chip memories.

SPICE Transistor Switching Estimate

As a final approach to estimating the datapath power, we note from the XScale design documentation [21] that the datapath is made up of approximately 650,000 transistors. Previous estimates suggest that around 10% of these transistors switch on a given active cycle [15]. Given an estimate of the switching power for transistors in this process technology, it should be possible to develop an estimate of the total power dissipation of these the switching transistors in the datapath component.

In order to generate such as estimate, we utilize a SPICE model for a simple inverter. We generate transistor model files with the Berkeley Predictive Technology Model [1] using technology parameters from XScale design documentation [21] (for a .18μm process) or default values where necessary. The inverter model assumes a 4x minimum size inverter driving an output load equivalent to four 4x minimum size inverters. While this is likely large for many of the components in the datapath, we are not considering interconnects and other datapath components, and therefore go with this estimate. We drive this SPICE model with pulse voltage transitions from 0 V to 1.3V. The average power dissipation as reported by SPICE is:

$$P_{ave}(trans) \approx 6.83x10^{-6}\text{W}$$

As an inverter represents two transistors, and assuming 10% switching, we can estimate the switching power dissipation of the datapath as:

$$\begin{aligned} P_{datapath} &\approx P_{ave}(trans) \cdot \frac{num\_transistors \cdot 10\%}{2} \\ P_{datapath} &\approx 220\text{mW} \end{aligned}$$

Summary

Thus, from our three estimation frameworks, we have:

$$\begin{aligned} P_{datapath}(Datasheets) &= 217\text{mW} \\ P_{datapath}(AreaScale) &= 185\text{mW} \\ P_{datapath}(Spice) &= 220\text{mW} \end{aligned}$$

Based on this analysis, we hereby adopt the datasheet power estimate (217 mW) as necessary for any further calculations.

Speech Processing Element Datapath

We now estimate speech processing element active power dissipation by performing area scaling of the relevant components of the XScale die (as shown in Figure B.2). To arrive at a reasonable area estimate, we begin by recognizing that the instruction and datapath complexity of the processing elements are far less than the corresponding architectural components of the XScale processor. Furthermore, the processing elements utilize a branch–not–taken prediction scheme, and are assumed not to require memory address translation (or to utilize the memory management capabilities of the main processor, since memory is a shared resource on most of our evaluation configurations). On the other hand, the processing element will require some logic to perform bus communication and internal scheduling, which does not exist on the main processor.

Given these complexity trade-offs, we take the relative datapath area of each processing element to be roughly equivalent to the area corresponding to decode and execute logic on the XScale processor, along with a small chunk of the writeback/control logic. A contiguous slice from the corresponding datapath component includes one full register context, and a linearly area scaled power estimate based on the 216mW figure for the entire datapath of

the XScale results in a active power dissipation estimate of:

$$P_{PE}(\text{Area Scaled}) \approx 50 - 60 \text{ mW active}$$

By comparison, assuming an even transistor density, we can estimate 150,000 transistors in our target slice. Utilizing the same spice model based evaluation discussed previously, and once again assuming 10% switching, we arrive at:

$$P_{PE}(\text{Spice}) \approx 51.2 \text{ mW active}$$

Given these values, we incorporate a reasonable conservative error margin and utilize a active processing element power dissipation value of:

$$P_{PE} = 70 \text{ mW active}$$

B.1.2 Storage Resources

Having established a estimate for datapath and control, we must now consider the effects of added hardware contexts, and of the cache component of each processing element.

Thread Contexts

An added hardware context is essentially a register file (along with some extra routing). We therefore begin by once again generating an area scaled estimate of the register file from the XScale processor. We add to this area estimate using register bit equivalent techniques [67] to help account for the added routing area. This results in an approximation of around 5mW per active cycle. Recognizing that the register file is relatively high density and a high utilization component of the datapath, we double this estimate and arrive at:

$$P_{context} \approx 10 \text{ mW active}$$

Configuration	n_{banks}	$E_{access}(total)$	$E_{access}(bank + route)$	t_{access}	Area
2K, 4–way	1	0.5 nj	0.5 nj	0.9 ns	$1mm^2$
128K, 4–way	2	1.5 nj	0.7 nj	1.1 ns	$15mm^2$
256K, 4–way	2	1.8 nj	0.9 nj	1.5 ns	$25mm^2$
512K, 4–way	2	2.3 nj	1.2 nj	1.7 ns	$50mm^2$
1M, 4–way	2	3.9 nj	1.9 nj	2.9 ns	$130mm^2$

Table B.1: Cache Energy Dissipation Values - This table lists cache energy dissipation values utilized for a number of the cache configurations considered in this work. The values are derived from information provided by the Cacti3 toolset

Similar techniques are utilized to account for the storage resources represented by the processing element work queues, though these constitute a far smaller number of registers with much lower utilization, and in practice do not contribute significantly to power dissipation. Note that the actual per-cycle energy contribution of hardware contexts and work queue elements that are not in use is not their active power dissipation, but their idle power dissipation (discussed shortly).

Caches

Cache power estimation is somewhat more straightforward. We intend to utilize energy and delay estimates from Cacti 3 [89] which, as discussed previously, should provide a conservatively high estimation of power consumption. In fact, analysis of cache estimations relative to spice simulations of compiled memory netlists for recent technologies shows that Cacti consistently over-estimates per–access energy consumption by considerable amounts (over 2 or 3x for larger caches). In keeping with the conservation techniques utilized on the XScale, we will consider cache access energy to be equivalent to the energy necessary to access a single bank plus routing energy. This divided by the access time for a particular cache configuration produces our estimate of active power dissipation. We utilize this method to estimate activity based power consumption for all data (l1 and l2) and instruction (shared l1) caches. In order to mitigate the effects of Cacti's over-estimation for more recent technologies, we divide our power consumption estimates in half for cache sizes over 32K. This still constitutes a very conservative (high) estimate, but is somewhat more in line with

processing advances.

In order to clarify the methodology discussed, we consider energy estimation for a single 2k, 4–way thread processor DL1 cache. We generate a Cacti configuration for a .18μm process with a single read/write port, and two banks. The results are then scaled from 1.7 V to the 1.3V XScale operating voltage, resulting in:

$$\begin{aligned} E_{access} &\approx 0.5\text{nJ } (E_{bank} + E_{routing}) \\ t_{access} &\approx 0.9\text{ns} \end{aligned}$$

If, over the course of program execution, there are n accesses to the cache, then total *active* energy dissipation is:

$$E_{Active} = n \cdot E_{access}$$

and the total amount of execution time during which this cache was actively accessed is:

$$t_{Active} = n \cdot t_{access}$$

The difference between t_{Active} and total simulation time is accounted for by idle power dissipation, as discussed later. This basic approach is used for all system caches. Note that, if we were considering a cache size larger than 32K, we would utilize $\frac{E_{access}}{2}$ in computations. Note also that the t_{access} generated by cacti and used for active energy consumption is not the same as the access latency (in simulation cycles) utilized for a given cache configuration. This later metric attempts to take communication and internal and external protocol overheads into account. Thus, we do not assume, for example, that a 1MB cache may be accessed in 2 processor cycles (at 400MHz). A listing of a number of cache configurations used in this work, and their power values, is depicted in Table B.1.

B.1.3 Computation Management

Beyond the processor datapaths, registers, and caches, the only other computational or storage components on the processor die are the global lock manager and the dynamic load balancing infrastructure.

The global lock manager is represented in our model as little more than a few registers, a bus encoder to interface with the control bus, and a small amount of control logic. In essence, this device maintains two registers per lock. One tracks the current lock holder, and the second represents a bitfield of lock requesters. All communication with the lock manager is performed over the control bus, and is accounted for in that model (described below). The control logic need do little more than respond to lock requests and release commands, which occur very rarely due to the locking facilities available on the individual processors. We model the active power dissipation of this component as a 10mW register file plus 10mW for logic. Thus:

$$P_{LM} \approx 20\text{mW active}$$

Though both values are generously excessive, the lock manager contributes insignificantly to the overall energy consumption of the system.

Our model for dynamic load balancing does not treat the balancing engine as a physical device on the system. Rather, we implement this component in a distributed fashion. As only one job migration can occur at any given time, the relevant global information is not which processing elements can potentially accept a migrated job, but rather whether such a processing element exists. Since each processing element can independently determine it's own ability to accept a job (a free context and an empty work queue), we assume the presence of a single bit line (similar in orientation to a control bus line) that may be pulled high by any processing element to broadcast such a signal. The logic needed on each processing element to identify if it or another element is activating the global signal is fairly trivial, and can ensure that (barring collisions) only one processing element recognizes

itself as responsible for accepting the next migrated job request. As job migration itself is performed over the control bus, this single line signal is sufficient to achieve the desired purpose. Given this implementation framework, we model the energy consumption of job migration as transitions on this global signal, and as communication on the control bus. The methodology for modeling both is presented in the next section. In practice, however, the energy dissipation of this configuration is insignificant in the overall system.

B.1.4 Communication Network

The final chip level component for which we wish to track energy consumption is the system bus. While the actual control bus does not contribute significantly to overall energy dissipation due to extremely low utilization, we will employ the same methodology to generate energy consumption estimates for the memory communication interface, and thus discuss the process here.

We are interested in the total energy dissipated through communication on system buses. This primarily amounts to accounting for the energy required to drive bus line state transitions, and is given by the equation:

$$\begin{aligned} E_{bus} &= \frac{1}{2} \cdot C_{load} \cdot V_{dd}^2 \cdot n_{trans} \\ C_{load} &= (250\text{fF/mm}) \cdot (4.1\text{mm} + (1.5\text{mm} \cdot \frac{1}{2} num\_PEs)) \end{aligned}$$

where C_{load} is the total capacitative load of a single bus line, V_{dd} is the supply voltage, and n_{trans} is the total number of transitions occurring across all bus lines.

The capacitative load of a bus line is related to processing technology and length. Utilizing the Berkeley Predictive Technology Model [1] for an aluminum top level global interconnect in .18um, we estimate the capacitative load of each bus line to be approximately 250 fF/mm. We utilize our virtual layout to estimate a total bus length based on the specific processor configuration, and use this length to generate a overall line load. Finally, we add 10 fF for each interface point between the bus and a computational component. This value

is somewhat greater than twice the typical input load of a minimum size inverter in .18um, and represents the added load of buffers to tap the bus and pass data into communication logic. We add one such buffer equivalent for each processing element, one for the main processor, and one for the global lock management infrastructure.

Transition counts are generated at simulation time. We implement a fairly simple bus protocol, and accumulate the total hamming distance between all contiguous transitions for the duration of a simulation. We assume that bus charge dissipates between communication events separated by more than a few cycles, causing the initial communication after such a lull to be a transition from zero on all lines. Based on our performance analysis data, we assume a 8–bit control bus in all evaluations.

B.1.5 Idle Power Dissipation

Thus far, we have discussed the power dissipation during active execution or active utilization of a processor component. As we are employing an activity based energy estimation model, this only provides half the picture. When components of the system are idle, they continue to dissipate potentially substantial amounts of static power which must be accounted for. The Intel design documentation for the PXA250 suggests that at idle their typical system dissipates around 120mW, or around 25% of the active power rating. This conforms well with analysis of circuit idle power dissipation in other works [25], and in fact constitutes a high estimate for a .18μm processes (where leakage is minimal). As all of the components discussed thus far are assumed to be on the same die, we adopt this assumption and generally assume that a given component dissipates around 25% of it's active power rating during all idle time (unless explicitly placed into a low power state, as discussed below). While .18μm processes would not actually leak background energy at this rate, we believe this 25% estimate will help to account for non–aggressive clock gating, logic that can not be disabled or must maintain state, and energy dissipated in areas such as the clock

distribution network itself. Thus:

$$E_{Idle} = \frac{1}{4} \cdot P_{Active} \cdot t_{Idle}$$

Note that this excludes the communication bus, which models dissipation of charge by resetting the "last value" of all lines to zero after a idle interval (leading to a potentially larger number of transitions in the future). Also note that, while the active energy dissipation of cache structures is based on activation of single banks, the idle power dissipation is a fraction of the total power that would be dissipated by all banks (not including routing).

This set of idle power estimates makes one further assumption with regards to the main processor which is important to note. The XScale processor itself does not have an "idle" NOP instruction, implementing NOPs as moves from register 0 to itself. As a result, the actual XScale processor sees very little difference in energy consumption between active and "runtime idle" (as opposed to extended standby) states. In our system, most of the work of speech evaluation is performed on our processing elements, leaving the XScale with considerable idle time. We consider it only reasonable to assume that some added energy reduction would take place on the XScale during this idle time, and thus assume idle power consumption (essentially aggressive clock-gating) rather than normal active power consumption.

B.1.6 Standby Power Dissipation

One final point of consideration is the potential of bringing the entire system into a low–power state for lengthy idle periods. For example, as our configurations exceed realtime performance, we must account for the possibility that they will need to await the next realtime interval before they can proceed to the next frame of input data. This constitutes a well–known idle interval, and if long enough, would present an ideal time to employ clock or voltage gating to cut down on energy dissipation. In order to accurately reflect the energy saved during standby intervals, we make a number of assumptions. First, we assume that memory elements such as register files (thread contexts) do not enter a standby state, and

minimally dissipate idle power to ensure state preservation. For the cache components of our design, we assume the use of techniques such as drowsy caches [30] may be employed, and conservatively treat standby power dissipation as 10% of total power dissipation. For the computational components, we assume V_{DD} gating techniques may be employed, and treat standby power dissipation as 1% of active power dissipation. Thus:

$$\begin{aligned} E_{Standby}(context) &= \frac{1}{4} \cdot P_{Active}(context) \cdot t_{Standby}(context) \\ & and \\ E_{Standby}(cache) &= \frac{1}{10} \cdot P_{Active}(cache) \cdot t_{Standby}(cache) \\ & and \\ E_{Standby}(logic) &= \frac{1}{100} \cdot P_{Active}(logic) \cdot t_{Standby}(logic) \end{aligned}$$

Note once again that this excludes communication resources, and that the active power utilized for caches in these computations is based on total cache energy dissipation, not per–bank dissipation.

B.2 System Level

The next step in our full system energy evaluation framework is to account for off–chip communication systems and component devices such as memories. We wish to consider the impact of communication both off chip through the system board, and to components within the same physical package as the processor, though potentially not on the same die. To this end, we once again break down energy into activity based estimates of individual design components.

B.2.1 Memory Communication

One of the major communication components in this infrastructure is the memory communication interface. We wish to model energy cost of memory communication at all levels of the memory hierarchy. At a basic level, the methodology used is identical to that used

Interconnect	Capacitance	Description
Proc – L1	0.05 pF	Processor to any L1 cache structure
Proc – EB	0.1 pF	Processor to any embedded memory device
L1 – EB	0.1 pF	L1 to any embedded memory device
L1 – L2	CB	L1 to On–chip L2 cache
L1 – On-pkg	CB + 10 pF	L1 to On–package, Off–chip memory
L1 – Off-pkg	CB + 60 pF	L1 to Off–chip memory
L2 – On-pkg	CB + 10 pF	L2 to On–package, Off–chip memory
L2 – Off-pkg	CB + 60 pF	L2 to Off–chip memory

Table B.2: Capacitance Computation Method for Various Interconnects - This table summarizes memory system interconnect levels, and the capacitance values used in estimating interconnect energy dissipation. The symbol CB represents the capacitance of the on–chip bus, computed based on the processor configuration as discussed previously. The symbol EB represents on–chip, embedded memories

to model the control bus. In this instance, however, we assume the bus is 64–bits wide between all hierarchy levels. In order to mitigate simulation latency, we perform a series of experiments to calculate the average Hamming distance between communications through the memory system, arriving at a value of 16 bits. We then utilize this average Hamming distance, and computations of total numbers of communication events along each level of the memory hierarchy to determine total transitions.

The capacitative load seen along each level of the hierarchy is dependent on the memory configuration. In general, potential capacitative values can be divided into on–chip communication, on–package communication, and off–package communication. Capacitance for memory communication to on–chip components (such as the communication between L1 and L2 caches) is computed in the same way as the capacitance of the control bus. The capacitance to memory components that are off chip, but are assumed to be on the package, total the on–chip communication discussed previously plus a fixed capacitance of 10pF. Similarly, communications off–package see an added capacitance of 60pF. Both of these values are taken from cited works [31]. The off–chip value is based on the capacitance to memory on an ARM7 system (similar in nature to our environment). The on–package value is based on the cited value for an off–chip L1 cache, and, as with other components, is accepted as a conservative estimate. A summary of these various interconnects and their

capacitance values is presented in Table B,2.

B.2.2 Memory System

The final system level component in our energy evaluation is main memory. We wish to evaluate a number of potential memory systems, ranging from standard SDRAM to on–chip embedded Flash and ROM technologies. As before, we will utilize activity based estimates combined with component specific (in this case, taken from product datasheets) energy values. Most of our memory analysis makes use of existing energy estimation systems, and is thus very straightforward.

DRAM Based Memories

We estimate the energy of DRAM based memory systems by utilizing the SDRAM system power calculator available from Micron Technologies [4]. This system power calculator account for active, precharge, Read/Write, refresh, and background power dissipation based on activity rate for a single DRAM chip. The cycle and access rates for the chip also place limits on the amount of bandwidth each chip can provide. Thus, we divide the number of DRAM requests across the number of chips necessary to hold our datasets and provide the expected bandwidth. Device parameters for SDRAM systems are taken from device datasheets for Micron Technologies 1.7V mobile SDRAM components. Device parameters for DDR-SDRAM systems utilize datasheets for Micron Technologies 2.5V DDR cells.

The Micron Technologies system power calculator bases it's power estimates on a combination of device specifications and usage specifications. As mentioned, we collect device specific information from data sheets for Micron technologies low power embedded SDRAM technology, and standard DDR technology. A list of these parameters, along with values used in our analysis is presented in Table B.3. The second component, usage information, is where our activity based estimation framework comes into play. It utilizes information on the average number of active DRAM cycles, reads, and writes, as well as information on how aggressively the system is placed into a low power state. A listing of usage parameters,

Parameter	SDRAM	DDR	Description
V_{cc}	1.7 V	2.4 V	System Voltage
I_{dd1}	50 mA	200 mA	Maximum Operating Current
I_{dd2}	3.5 mA	3 mA	Maximum Precharge Power-Down Standby Current
I_{dd3}	35 mA	125 mA	Maximum Active Standby Current
I_{dd4}	80 mA	350 mA	Maximum Read Burst Current
I_{dd6}	2 mA	3.5 mA	Maximum Distributed Refresh Current
t_{CKc}	9.5 ns	8 ns	tCK for current measurements
t_{RDD}	20 ns	16 ns	Minimum activate-to-activate timings (diff banks)
t_{RC}	80 ns	60 ns	Minimum activate-to-activate timings (same banks)
t_{CK}	8 ns	5 ns	Minimum clock cycle

Table B.3: DRAM Device Parameters and Descriptions - This table lists device parameters (taken predominantly from device data sheets) for SDRAM devices used in our evaluation, and the meaning of each term.

Parameter	"Calc" / Value	Description
f_{ck}	100 / 200 MHz	System Clock Frequency
C_{load}	25 pF	Output Load
$Pc_{precharge}$	CALC (%)	Percent time all banks on the DRAM are in a precharged state
$Pc_{p\_ck\_low}$	80%	Percent all bank precharge time w/ CKE held LOW
$Pc_{1b\_ck\_low}$	0%	Percent > 1 bank active time w/ CKE held LOW
t_{ave_ACT}	CALC (ns)	Average time between ACT commands
Pc_{RD}	CALC (%)	Percent cycles outputting READ data
Pc_{RD}	CALC (%)	Percent cycles inputting WRITE data

Table B.4: DRAM Usage Parameters and Descriptions - This table lists device usage parameters for our DRAM evaluations. Values marked CALC are calculated based on activity information. More data is available from Micron Technologies

along with their descriptions and annotation as to which are calculated from activity information and which are fixed is presented in Table B.4. Other than the output load, which is left at it's default value (communication back to the processor is accounted for in our memory bus analysis), the remaining fixed values ($Pc_{p\_ck\_low}$ and $Pc_{1b\_ck\_low}$) are related to how aggressively the device maintains a power down state when in precharge mode (CLK held LOW basically equates to disabling of the device), and the values utilized represent a fairly aggressive power minimization scheme.

The remaining values are computed based on simulation information. The first step in this process is to determine how many memory cycles occur over the course of the simulation,

and how many of those cycles are taken up with READ/WRITE operations. In order to determine how many are utilized by read/write operations, it is necessary to know how many memory cycles are taken up by each memory access. Thus:

$$\begin{aligned} total\_mem\_cycles &= f_{ck} \cdot sim\_time \\ & and \\ cycles\_per\_access &= \frac{transaction\_size}{channel\_width} \end{aligned}$$

where sim_time is the reported running time based on simulator cycles, $transaction\_size$ is the cache block size plus protocol overhead, and $channel\_width$ is the width of the memory channel (64–bits in our system). From this, we can calculate the total number of memory cycles for which the device is active as:

$$total\_active\_cycles = (Reads + Writes) \cdot cycles\_per\_access$$

where $Reads$ and $Writes$ are the number of reads and writes that proceed off chip and into the memory system respectively. We are now ready to calculate the parameters necessary to fill in our usage table, as follows:

$$\begin{aligned} Pc_{precharge} &= (0.9 - \frac{total\_active\_cycles}{total\_mem\_cycles}) \\ & and \\ t_{ave\_ACT} &= \frac{sim\_time}{Reads + Writes} \\ & and \\ Pc_{RD} &= \frac{Reads \cdot cycles\_per\_access}{total\_mem\_cycles} \\ & and \\ Pc_{WR} &= \frac{Writes \cdot cycles\_per\_access}{total\_mem\_cycles} \end{aligned}$$

Note in our calculation of $Pc_{precharge}$ that the remaining 10% of time accounts for time

when the system is not "active", but has not yet entered a "low power" state for whatever reason. Furthermore, note that ACT commands are sent at the beginning of an access, so t_{ave_ACT} is based on the number of unique accesses, not the number of cycles those accesses take up.

While a full analysis of the computations used to arrive as power estimates from these inputs is beyond the scope of this work (and available as a technical report from Micron Technologies [4]), the system power calculator uses these device and usage parameters to compute active and background power dissipation for the memory device in milliwatts. Thus, the final step in our calculation of device energy dissipation is arrived at by:

$$E_{total} = (P_{Active} \cdot t_{Active}) + (P_{Background} \cdot t_{Idle})$$

where t_{Active} and t_{Idle} represent the total time the device is active over the course of the workload, and the remaining time. Both may be calculated from information already presented.

FLASH Memories

Energy estimates for FLASH memory are based on Micron Technologies Q-Flash chip datasheets for a 64–MB flash chip and page mode flash datasheets. We assume only reads occur from flash chips during runtime (enforced by data placement), and further assume the transfer of a full cache line on each request. We utilize a much more straightforward energy computation framework for flash chips (relative to DRAM chips).

At a fundamental level, our flash energy computations are based on estimations of the amount of energy required to perform each flash access, over the total number of flash accesses occurring over the course of program execution. Once again, data transfer energy is accounted for in our memory bus communication model. As we are transferring full cache lines, it is possible to assume that a full set of page mode accesses is completed on each flash access. Each read moves two bytes of data off of the flash device, and seven consecutive page reads may be issued once the first page is opened, resulting in a total of 16 bytes per

Parameter	Value	Description
V_{DD}	1.7 V	Device voltage
I_{RD1}	15 mA	Current draw for first page read (random access)
I_{RDp}	5 mA	Current draw for subsequent page reads (sequential)
t_{RD1}	90 ns	Time from request to first read
t_{RDp}	30 ns	Time for subsequent page reads after first
I_{STBY}	50 μA	Current draw in standby mode
I_{Leak}	2 μA	Leakage current

Table B.5: FLASH Device Parameters and Descriptions - This table lists device parameters (taken from device data sheets) for the page mode FLASH devices used in our evaluation, and the meaning of each term.

page. It is assumed that this data is buffered as necessary in the memory controller in order to be transferred along the 64–bit processor to memory bus.

The base information utilized in our energy estimation is presented in Table B.5. Note that the energy cost of initial address decoding and page opening is accounted for by the current and delay of the first page read. Given this information, we compute the total energy per access as:

$$E_{access} = (V_{DD} \cdot I_{RD1} \cdot t_{RD1}) + (V_{DD} \cdot I_{RDp} \cdot t_{RDp} \cdot num\_pages)$$

where num_pages, once again, represents the seven page mode reads that may be completed sequentially after the page is opened. The idle power dissipated by the cache is given by:

$$P_{Idle} = (V_{DD} \cdot (I_{STBY} + I_{Leak}))$$

From this information, total energy dissipation estimates can be derived as follows:

$$\begin{aligned} E_{Active} &= num\_accesses \cdot E_{access} \\ t_{Idle} &= (sim\_time - (num\_accesses * time\_access)) \\ &\textit{and} \\ E_{Idle} &= P_{Idle} \cdot t_{Idle} \end{aligned}$$

Thus, as before, the total energy of the flash system is simply the sum of the active and idle energy dissipation, or:

$$E_{Total} = E_{Active} + E_{Idle}$$

ROM Memories

In order to estimate energy consumption for our ROM based structures, we must make a number of simplifying assumptions. It is generally accepted that the major energy dissipation in large ROM devices comes from charging and discharging large capacity lines such as pre–decoder lines, word–lines, and bit–lines [101]. We therefore begin by discounting the energy dissipation in address decode logic, and consider only the energy dissipated in such data and control lines. We also discount leakage energy dissipation, as we assume ROM designs can be constructed to minimize such leakage to at least the levels of previously discussed FLASH chips, making it essentially irrelevant.

Secondly, in order to make reasonable estimates of line capacitance, it is necessary to estimate line length. We therefore once again generate a virtual layout of the ROM chip, and use parameters from this virtual layout to estimate line lengths. In this layout, we assume that the data arrays constitute the bulk of chip area, and once again ignore area contributions of global logic and routing.

Third, we recognize that simple line gating techniques can be quite effective at reducing the effective capacitance of large global control wires. We thus assume a banking model for the ROM chip, with only the relevant bank being activated for a given access, and line gating techniques utilized to channel control signals to the relevant bank (rather than through the entire chip). The area of each bank will be given by $total\_area/num\_banks$. We assume square blocks at all times, and assume that word–lines and bit–lines are activated on a per–bank basis only. Note that this is really just a conceptual division, and in the end our estimation could be considered as a conservative estimate for a multi–chip solution as well. In that case, what we consider to be energy dissipated in pre–decode lines would be replaced by energy dissipated in bank / chip selection lines.

Parameter	Value	Description
ROM_size	64 MB	Data size of ROM array
n_{banks}	16	Number of "banks" in ROM
n_{rpb}	32768	Number of rows per ROM bank
n_{bpr}	1024	Number of bit lines per row
num_banks	16	Number of "banks" in ROM
V_{DD}	1.7 V	Device operating voltage
V_{ref}	$V_{DD}/2$ V	Reference voltage for sense amps
A_{bit}	$4f^2$	Area of a single bit (f = min. feat. size)
C_{gbl}	250 fF/mm	Capacitance for global lines (eg: pre–decode)
C_{loc}	215 fF/mm	Capacitance for local lines (word–lines, bit–wise)
C_{buf}	10 fF	Capacitance for each buffer on a line

Table B.6: ROM Design Parameters - This table presents a summary of the design parameters used in our ROM energy estimation methodology.

Finally, we recognize the existence of a number of recent techniques to reduce the amount of energy dissipated in ROM systems, and assume the presence of techniques such as charge sharing [101] to help mitigate the energy dissipation of data and control lines. We also assume that sense amplifiers are utilized at the end of the column decoding process to reduce the charge swing necessary on bit–lines.

With this overview, we will now present details of our modeling. In order to simplify evaluation, we assume a 64MB ROM cell for all relevant simulations, which is large enough to hold our datasets (though larger structures may be needed for different vocabularies or different speech recognition infrastructures). We begin by estimating the area of the total data array. Recent work [88] estimates the size of a single ROM bit as:

$$A_{bit} \approx 4f^2$$

where f is the minimum feature size of the technology. Assuming a .18μm process, an

estimation of the total area of the data array can be calculated as:

$$\begin{aligned} num\_bits &= (ROM\_size \cdot 8) \\ A_{ROM} &= num\_bits \cdot A_{bit} \\ A_{ROM} &\approx 70\text{mm}^2 \end{aligned}$$

We now assume the presence of 16 internal banks in the ROM. This assumption is admittedly somewhat arbitrary, but is based on recognizing that a greater number of banks may reduce the energy dissipated by word and bit lines on individual accesses, but will add to the total amount of control and routing area / logic, invalidating some of our area assumptions. Given this, the area of each bank is approximately:

$$\begin{aligned} A_{bank} &= \frac{A_{ROM}}{num\_banks} \\ A_{bank} &\approx 4.5\text{mm}^2 \end{aligned}$$

Utilizing a further assumption of square component blocks, we arrive at the length values necessary to estimate wire length as:

$$\begin{aligned} L_{side}(ROM) &= \sqrt{A_{ROM}} \approx 8.4\text{mm} \\ L_{side}(Bank) &= \sqrt{A_{bank}} \approx 2.1\text{mm} \end{aligned}$$

Based on the analysis thus far, we generate a high–level virtual ROM layout as depicted in Figure B.3. We are now able to start estimating energy dissipation. Our energy estimation is based very much on data presented in cited works [101], and will consider energy dissipated along pre–decode lines, word–lines and bit–lines.

Pre–Decode

As depicted in the figure, we assume the first stage of decoding (and bank selection) is performed within the decoder block. The pre–decode signals are transmitted out to the

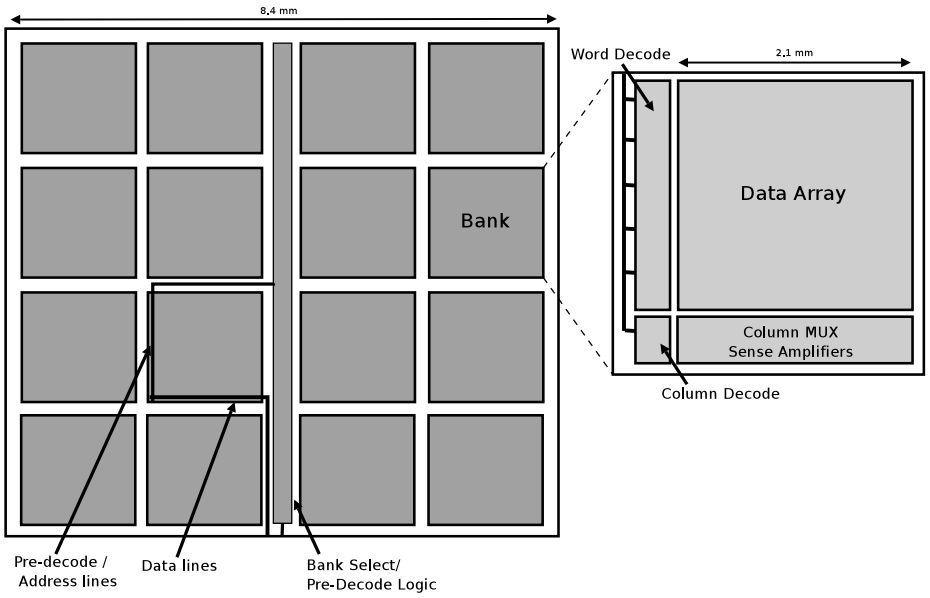


Figure B.3: Virtual ROM Layout - This figure depicts a virtual layout of our ROM chip, which we use to estimate wire lengths in our power analysis.

selected bank. We assume equal distribution of accesses, and thus estimate that on average the signal must traverse one quarter of the chip along each axes to reach the back, and must then traverse the length of the bank. Furthermore, the line capacitance must include the cost of connections with the word–line decoders. As these are wires, the power dissipation is related to the number of transitions. This, in turn, averages to 2 (one line activated, another deactivated) per access. Thus, we can arrive at the energy dissipated on a pre–decode line per access as follows:

$$\begin{aligned} n_{trans} &= 2 \\ L_{line} &= (\frac{1}{4}L_{side}(ROM)) + (\frac{1}{4}L_{side}(ROM)) + L_{side}(bank) \\ C_{line} &= (C_{gbl} \cdot L_{line}) + (C_{buf} \cdot n_{rpb}) \\ E_{predecode\_line} &= \frac{1}{2} \cdot C_{line} \cdot V_{DD}^2 \cdot n_{trans} \end{aligned} \tag{B.1}$$

Note that our capacitance estimates are once again taken from the BPTM [1]. We extend this computation by noting that both row and column select information must be trans-

mitted on a access. We make the assumption that data is arranged in the DRAM to allow a full cache line to be read by cycling through the relevant columns activated by a single word–line in a single bank. We further assume 64–bit rows, implying that for a 32–byte block size, 4 reads must be performed. This results in one set of transitions along the word selection lines, and four sets of transitions along the column selection lines. Finally, we assume a single line is used to activate the bank (and any necessary routing logic). For the sake of simplicity, we assume this control line incurs the same capacitance as the decoder lines, though it interfaces with far fewer buffers (note that we make the same conservative assumption regarding column selection lines). Thus, the total energy dissipated by the pre–decode lines for a single access is:

$$E_{predecode} = E_{control\_tran} + E_{word\_trans} + (4 \cdot E_{col\_trans})$$

Incorporating all of the relevant numbers, we arrive at an estimate of:

$$E_{predecode} \approx 5.7 \frac{\text{nJ}}{\text{access}}$$

We now further assume the use of charge sharing to reduce the explicit energy input needed to switch lines as discussed in cited works [101]. As we assume that, on average, the length of switched lines is approximately the same, we utilize the 1/2 estimate presented in that work, and arrive at:

$$E_{predecode} \approx 2.8 \frac{\text{nJ}}{\text{access}}$$

Note that there are a number of potential other gating techniques that may be used to reduce the total number of buffer capacitances incurred, and that our buffer capacitance itself is a very conservative estimate, so we are once again working off of a conservative estimate of dissipation here.

Word–lines

Given our pre–decode analysis, our word–line analysis is equally straightforward. We once again assume that each access leads to two transitions. The remaining equations are essentially identical except that the wire length is assumed to be the length across a bank, as opposed to a length across some component of the chip. Thus:

$$\begin{aligned} n_{trans} &= 2 \\ L_{line} &= L_{side}(bank) \\ C_{line} &= (C_{loc} \cdot L_{line}) + (C_{buf} \cdot n_{bpr}) \\ E_{wordline} &= \frac{1}{2} \cdot C_{line} \cdot V_{DD}^2 \cdot n_{trans} \end{aligned} \tag{B.2}$$

We again extend this analysis by noting that the column MUX select lines are essentially identical to the word–lines themselves. The word–lines, however, undergo a single set of transitions per device access, while the column select lines once again undergo 4. Thus, the total energy dissipated by the word and column select lines for a single access may be estimated as:

$$E_{wordline} = E_{word} + (4 \cdot E_{col})$$

If we once again incorporate the relevant numbers as presented, we arrive at:

$$E_{wordline} \approx 30.77 \frac{\text{pJ}}{\text{access}}$$

In this case, we do not adopt charge sharing techniques, conservatively assuming that, as different banks are accessed and change in a given bank may dissipate before it is referenced again, charge sharing effects will not be as useful to exploit.

Bit–lines

Once again, our analysis thus far makes estimation of bit–line energy fairly straightforward. The adjustment from previous computations here comes from the assumption of sense amplifiers to reduce the energy that must be committed to a bit–line before it's value can be read. Thus, rather than a full swing from ground to V_{DD}, or the relatively small voltage voltage swing necessary to trigger the sense amplifier, we assume (again, conservatively) that the bit lines must be brought from zero up to a sense amplifier reference voltage of V_{ref}. Based on our assumptions regarding how data is read out of a bank, we assume that this reference charging is performed only once per device access. Note that, to account for the line capacitance of inactive words and the column selection MUX, we assume an added buffer capacitance of half the number of rows. Thus, we are left with:

$$\begin{aligned} n_{trans} &= 1 \\ L_{line} &= L_{side}(bank) \cdot n_{bpr} \\ C_{line} &= (C_{loc} \cdot L_{line}) + (C_{buf} \cdot \frac{1}{2} n_{rpb} \\ E_{bitline} &= \frac{1}{2} \cdot C_{line} \cdot V_{ref}^2 \cdot n_{trans} \end{aligned} \tag{B.3}$$

Incorporating the relevant numbers, we arrive at:

$$E_{wordline} \approx 226 \frac{\text{pJ}}{\text{access}}$$

Data Lines

The final component of our energy estimate for the ROM is the energy dissipated in transferring data from a bank to the chip edge. We assume that the first data element causes transitions on all 64–bits of our presumed data lines, and that all subsequent elements to complete a cache block equivalent read cause our previously computed average of 16 bit transitions. Further, we assume that the data must move to the central axis of the

chip, and then to an edge. We therefore estimate line distance as one quarter of the side length to reach the center, and another one half to reach the edge. As before, we cover communication back to the processor via our bus model. Thus, maintaining our 32–byte transfer size assumption:

$$\begin{aligned} n_{trans} &= 64 + (3 \cdot 16) \\ L_{line} &= \frac{1}{4} L_{side}(ROM) + \frac{1}{2} L_{side}(ROM) \\ C_{line} &= (C_{gbl} \cdot L_{line}) \\ E_{dataline} &= \frac{1}{2} \cdot C_{line} \cdot V_{DD}^2 \cdot n_{trans} \end{aligned} \tag{B.4}$$

Incorporating the relevant numbers, we arrive at:

$$E_{dataline} \approx 254 \frac{\text{pJ}}{\text{access}}$$

Total

Combining all of these computations, we generate a total per–access (i.e. per–cache–miss) energy estimate for the ROM chip:

$$\begin{aligned} E_{access} &= E_{predecode} + E_{wordlines} + E_{bitlines} + E_{datalines} \\ E_{access} &\approx 4.31 \frac{\text{nJ}}{\text{access}} \end{aligned}$$

Which directly leads to a total ROM energy estimate of:

$$E_{ROM} \approx E_{access} \cdot num\_accesses$$

APPENDIX C

SPHINX Phoneme Set

The following is a list of phonemes recognized by the CMP-SPHINX speech recognition engine. This list represents version *cmudict*.0.1 and was compiled by Jerry Quinn (jquinn@bnr.ca).

Phoneme	Example	Translation
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
IH	it	IH T
IY	eat	IY T
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
OW	oat	OW T
OY	toy	T OY
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
UH	hood	HH UH D
UW	two	T UW
V	vee	V IY
W	we	W IY
Y	yield	Y IY L D
Z	zee	Z IY
ZH	seizure	S IY ZH ER

APPENDIX D

HMetis Partitioning Statistics for Evaulation Vocabulary

The following sections contain hMetis output for various degrees of static graph partitioning utilized in our evaluation. Note that a "balance" of 1 implies equal weight partitions. All partitions were generated for a usage weighted node list on a 11447 word vocabulary.

D.1 2–way Partition

```
**********************************************************************
  METIS 4.0.1 Copyright 1998, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
  Name: 11447wd.dat, #Vertices: 40117, #Edges: 39554, #Parts: 2

Recursive Partitioning... -------------------------------------------
  2-way Edge-Cut:       0, Balance:  1.00

Timing Information --------------------------------------------------
  I/O:                          0.050
  Partitioning:                 0.080   (PMETIS time)
  Total:                        0.130
**********************************************************************
```

D.2 4–way Partition

```
**********************************************************************
  METIS 4.0.1 Copyright 1998, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
  Name: 11447wd.dat, #Vertices: 40117, #Edges: 39554, #Parts: 4

Recursive Partitioning... -------------------------------------------
  4-way Edge-Cut:       0, Balance:  1.00

Timing Information --------------------------------------------------
  I/O:                          0.050
```

```
  Partitioning:                    0.160   (PMETIS time)
  Total:                           0.210
**********************************************************************
```

D.3 8–way Partition

```
**********************************************************************
  METIS 4.0.1 Copyright 1998, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
  Name: 11447wd.dat, #Vertices: 40117, #Edges: 39554, #Parts: 8

K-way Partitioning... -----------------------------------------------
  8-way Edge-Cut:       0, Balance:  1.00

Timing Information --------------------------------------------------
  I/O:                             0.060
  Partitioning:                    0.090   (KMETIS time)
  Total:                           0.150
**********************************************************************
```

D.4 16–way Partition

```
**********************************************************************
  METIS 4.0.1 Copyright 1998, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
  Name: 11447wd.dat, #Vertices: 40117, #Edges: 39554, #Parts: 16

K-way Partitioning... -----------------------------------------------
  16-way Edge-Cut:       0, Balance:  1.24

Timing Information --------------------------------------------------
  I/O:                             0.060
  Partitioning:                    0.090   (KMETIS time)
  Total:                           0.160
**********************************************************************
```

D.5 32–way Partition

```
**********************************************************************
  METIS 4.0.1 Copyright 1998, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
  Name: 11447wd.dat, #Vertices: 40117, #Edges: 39554, #Parts: 32

K-way Partitioning... -----------------------------------------------
```

```
  32-way Edge-Cut:       0, Balance:  1.09

Timing Information -------------------------------------------------
  I/O:                            0.050
  Partitioning:                   0.110   (KMETIS time)
  Total:                          0.160
*********************************************************************
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Berkeley predictive technology model. http://www-device.EECS.Berkeley.EDU/ ptm/.

[2] Intel PXA250 processor. http://developer.intel.com/.

[3] Intel SA-1110 processor. http://developer.intel.com/design/pca/ applicationsprocessors/1110_brf.htm.

[4] Micron Technologies. http://www.micron.com/.

[5] NEC Embedded DRAM Technology. http://www.necelam.com/ASIC/edram.cfm.

[6] SimpleScalar toolset. http://www.simplescalar.com.

[7] B. Abali, H. Franke, X. Shen, D. Poff, and T. Smith. Performance of hardware compressed main memory. Available from: http://www-124.ibm.com/developerworks/opensource/mxt/publications/mxtperformance.pdf.

[8] S. Aditya, B. Rau, and V. Kathail. Automatic architectural synthesis of vliw and epic processors. In *International Symposium on System Synthesis*, pages 107–113, November 1999.

[9] K. Agaram, S. Keckler, and D. Burger. A characterization of speech recognition on modern computer systems. In *Proceedings of 4th Annual Workshop on Workload Characterization*, December 2001.

[10] V. Akinen. Front-end feature extraction with mel-scaled cepstral coefficients. http://www.hut.fi/u/vmakine2/mscc.doc [MS-Word], 2000.

[11] T. Anantharaman and B. Bisiani. A hardware accelerator for speech recognition algorithms. In *Proceedings of the 13th Annual Intl. Symposium on Computer Architecture*, pages 216–223, 1986.

[12] G. Antoniol, F. Brugnara, M. Cettolo, and M. Federico. Language Model Representation for Beam-Search Decoding. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 588–591, May 1995.

[13] E. Birney. Hidden Markov Models in Biological Sequence Analysis. *IBM Journal of Research and Development*, 45(3/4), 2001.

[14] M. Bohm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.

[15] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[16] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proceedings of 26th Annual International Symposium on Computer Architecture*, pages 64–74, May 1999.

[17] J. Cantin and M. Hill. Cache performance for selected SPEC CPU2000 benchmarks. In *Computer Architecture News (CAN)*, January 2001.

[18] S. Chatterjee and P. Agrawal. Connected speech recognition on a multiple processor pipeline. volume 2, pages 774–777, May 1989.

[19] G. Churchill. Hidden markov chains and the analysis of genome structure. *Computers & Chemistry*, 16(2):107–115, 1992.

[20] C.Lai, S.Su, and Q.Zhao. Performance analysis of speech recognition software. In *Proceedings of 5th Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[21] L. Clark, E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. Velarde, and Mark A. Yarch. An Embedded 32-b Microprocessor Core for Low-Power and High-Performance Applications. *IEEE Journal of Solid–State Circuits*, 36(11), November 2001.

[22] P. Clarkson and R. Rosenfeld. Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings of EUROSPEECH'97*, pages 2707–2710, 1997.

[23] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill Book Company, 1998.

[24] H. Corporaal and R. Lamberts. *TTA processor synthesis*. May 1995.

[25] C.Zhang, F. Vahid, and W. Najjar. A Highly–Configurable Cache Architecture for Embedded Systems. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[26] R. Dujari. Parallel viterbi search algorithm for speech recognition. Master's thesis, MIT, 545 Technology Square, Cambridge MA 02139, 1992.

[27] D. Grunwald et al. Policies for dynamic clock scheduling. In *4th Symposium on Operating Systems Design and Implementation*, 2000.

[28] B. Falsafi and D. A. Wood. Parallel dispatch queue: A queue-based programming abstraction to parallelize fine-grain communication protocols. In *HPCA*, pages 182–192, 1999.

[29] P. Faraboschi, G. Brown, G. Desoli, and F. Homewood. Lx: A technology platform for customizable vliw embed-ded processing. In *Proceedings of 27th Annual International Symposium on Computer Architecture*, pages 203–213, June 2000.

[30] K. Flautner, N. Kim, S. Martin, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. *ISCA'29*, May 2002.

[31] W. Fornaciari, D. Sciuto, and C. Silvano. Power Estimation for Architectural Exploration of HW/SW Communication on System-Level Busses. In *Seventh International Workshop on Hardware/software Codesign*, 1999.

[32] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed–setting of a low–power cpu. In *1st Intl. Conf. on Mobile Computing and Networking*, 1995.

[33] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1995.

[34] H. Hon. A survey of hardware architectures designed for speech recognition. Technical Report CMU-CS-91-169, August 1991.

[35] C. Hsu and U. Kremer. Compiler-directed dynamic voltage scaling for memory-bound applications. Technical Report DCS-TR-498, Department of Computer Science, Rutgers University, August 2002., 2002.

[36] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2):137–148, 1993.

[37] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *MICRO-34*, December 2001.

[38] R. Hughey and A. Krogh. Hidden markov models for sequence analysis: extension and analysis of the basic method. *CABIOS*, 12(2):95–107, 1996.

[39] C. Im, H. Kim, and S. Ha. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *2001 International Symposium on Low power electronics and design*, pages 34–39, 2001.

[40] D. Jagger and D. Seal. *ARM Architecture Reference Manual (2nd edition)*. Addison–Wesley, 2000.

[41] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[42] George Karypis. Metis family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/ karypis/metis/metis/index.html.

[43] S. Kaxiras, G. Narlikar, A. Berenbaum, and Z. Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, November 2001.

[44] P. Keltcher, S. Richardson, and S. Siu. An equal area comparison of embedded dram and sram memory architectures for a chip multiprocessor. Technical Report HPL-2000-53, 2000.

[45] N. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12), December 2003.

[46] N. Kim, T. Austin, and T. Mudge. Low–energy data cache using sign compression and cache line bisection. In *2nd Annual Workshop on Memory Performance Issues*, May 2002.

[47] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *International Symposium on Microarchitecture*, pages 204–213, 1997.

[48] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. *Journal of System Architecture*, (45):571–590, 1999.

[49] R. Krishna, S. Mahlke, and T. Austin. Insights into the memory demands of speech recognition algorithms. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues*, May 2002.

[50] M. Krishnan, C. Neophytou, and G. Prescott. Wavelet transform speech recognition using vector quantization. Preprint: http://citeseer.nj.nec.com/krishnan94wavelet.html, 1994.

[51] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler. Hidden Markov Models in Computational Biology: Applications to Protein Modeling. Technical Report UCSC-CRL-93-32, 1993.

[52] D. Kulp, D. Haussler, M. G. Reese, and F. H. Eeckman. A generalized hidden markov model for the recognition of human genes in DNA. *ISMB-96*, pages 134–141, 1996.

[53] J. Laudon, A. Gupta, and M. Horwitz. Interleaving: A multithreading technique targetting multiprocessors and workstations. In *6th ASPLOS*, pages 308–318, October 1994.

[54] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[55] K. Lee, H. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34:35–44, 1990.

[56] K.-F. Lee. *Automatic Speech Recognition: The Development of the SPHINX System.* Klewer Academic Publishers, 1989.

[57] Y.-H Lee and C. Krishna. Voltage-clock scaling for low power energy consumption in real–time embedded systems. In *6th Intl. Conf. on Real–Time Computing Systems and Applications*, 1999.

[58] T. R. Leek. Information extraction using hidden Markov models. Master's thesis, UC San Diego, 1997.

[59] Charles Lefurgy, Eva Piccininni, and Trevor N. Mudge. Evaluation of a high performance code compression method. In *International Symposium on Microarchitecture*, pages 93–102, 1999.

[60] Haris Lekatsas and Wayne Wolf. Random access decompression using binary arithmetic coding. In *Data Compression Conference*, pages 306–315, 1999.

[61] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, December 1996.

[62] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *ACM–SIGPLAN*, pages 260–267, June 1988.

[63] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control theoretic dynamic frequency and voltage scaling for multimedia workloads. In *CASES-2002*, 2002.

[64] J. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, San Jose, CA, October 2002.

[65] B. Mathew, A. Davis, and Z. Feng. A low-power accelerator for the sphinx 3 speech recognition system. In *Proceedings of the International Conference on Compiler, Architecture, and Synthesis for Embedded Systems (CASES 2003)*. ACM Press, October 2003.

[66] IBM Memory eXpansion Technology (MXT). http://www-124.ibm.com/developerworks/projects/mxt.

[67] J. Mulder, N. Quach, and M. Flynn. An Area Model for On-Chip Memories and its Applications. *IEEE Journal of Solid–State Circuits*, 26(2), February 1991.

[68] H. Murveit, P. Monaco, V. Digalakis, and J. Butzberger. Techniques to Achieve an Accurate Real-Time Large-Vocabulary Speech Recognition System. In *Proceedings of ARPA Human Language Technology Workshop*, pages 368–373, March 1994.

[69] H. Ney, R. Hæb-Umbach, and B.-H. Tran. Improvements in Beam Search for 10000-Word Continuous Speech Recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume I, pages I–9–I–12, March 1992.

[70] Y. Normandi, D. Bowness, R. Cardin, C. Drouin, and R. Lacouture. CRIM's November 94 Continuous Speech Recognition System. In *Proceedings of ARPA Speech and Natural Language Workshop*, pages 153–155, January 1995.

[71] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Intl. Symposium on Low Power Electronics and Design*, 2000.

[72] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM Symposium on Operating Systems Principles*, pages 89–102, 2001.

[73] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Intl. Simposium on Low Power Electronics and Design*, 2001.

[74] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2):257–286, February 1989.

[75] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[76] M. Ravishankar. Parallel implementation of fast beam search for speaker-independent continuous speech recognition. *Computer Science & Automation*, 1993.

[77] M. Ravishankar. *Efficient Algorithms for Speech Recognition*. PhD thesis, Computer Science Department, Carnegie Mellon University, May 1996.

[78] M. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[79] M. Rinard and P. Diniz. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. In *International Conference on Supercomputing*, pages 83–92, 1999.

[80] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. In *15th International Parallel and Distributed Processing Symposium*, pages 630–636, April 2001.

[81] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[82] H. Sakoe and S. Chiba. A dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(1):43–49, February 1978.

[83] P. Sanders. Parallelizing np-complete problems using tree shaped computations. Available at: http://citeseer.nj.nec.com/sanders99parallelizing.html, May 1999.

[84] R. Sasanka, S.V. Adve, Y.K. Chen, and E.Debes. Comparing the Energy Efficiency of CMP and SMT Architectures for Multimedia Workloads. Technical Report UIUCDCS-R-2003-2325, 2003.

[85] Y. Sazeides and J. Smith. The predictability of data values. In *Proceedings of 20th International Symposium on Microarchitecture*, December 1997.

[86] A. Senior. A hidden markov model fingerprint classifier. In *Proceedings of the 31st Asilomar conference on Signals, Systems and Computers*, pages 306–310, 1997.

[87] K. Seymore, A. McCallum, and R. Rosenfeld. Learning hidden Markov model structure for information extraction. In *AAAI 99 Workshop on Machine Learning for Information Extraction*, 1999.

[88] T. Sherwood and B. Calder. Patchable instruction rhttp://www.necelam.com/asic/edram.cfm. In *Proceedings of the International Conference on Compiler, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, pages 24–33. ACM Press, November 2001.

[89] Premkishore Shivakumar and Norman Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, August 2000.

[90] E. Sonnhammer, G. Heijne, and A. Krogh. A hidden Markov model for predicting transmembrane helices in protein sequences. In *Proceedings of 6th International Conference on Intelligent Systems for Molecular Biology*, pages 175–182. AAAI Press, 1998.

[91] J. Srinivasan and S. Adve. Predictive dynamic thermal management for multimedia applications. In *ICS'03*, June 2003.

[92] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA-22*, June 1995.

[93] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *MICRO-33*, 2000.

[94] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, volume IT-13, pages 260–269, April 1967.

[95] S. Vlaovic and R. Uhlig. Performance of natural i/o applications. In *Proceedings of 2nd Workshop on Workload Characterization*, October 1999.

[96] David Wang and Bruce Jacobs. MASE DRAM memory simulator manual. http://www.ece.umd.edu/courses/enee759h.S2003 /references/mase_dram.pdf.

[97] R. Williams. *Electrical Engineering Probability*. West Publishing Company, 1991.

[98] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. pages 101–116, 1999.

[99] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO-25*, pages 81–91, December 1992.

[100] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *Proceedings of 28th Annual International Symposium on Computer Architecture*, pages 110–119, June 2001.

[101] B. Yang and L. Kim. A Low–Power ROM using charge recycling and charge sharing techniques. *IEEE Journal of Solid–State Circuits*, 38(4), April 2003.

[102] J. Yang and R. Gupta. Energy efficient frequent value data cache design. In *MICRO-35*, 2002.

[103] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value–centric data cache design. In *ASPLOS-9*, November 2000.

[104] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions Information Theory*, IT-23(3):337–343, 1977.