

Insights into the Memory Demands of Speech Recognition Algorithms

Rajeev Krishna, Scott Mahlke, Todd Austin
{rkrishna, mahlke, taustin}@eecs.umich.edu

Abstract:

The vision of pervasive computing is one of invisible computers interacting with humans in all aspects of their lives. These invisible computers can be embedded in anything from specialized portable devices to articles of clothing. To make this vision a reality, efficient, convenient, and effective human communication with computers must be provided. Keyboards and mice are effective in a desktop environment, but these are clumsy, awkward, and not well suited for mobile/hidden computers. Personal digital assistants and cellular telephones have added primitive utterance recognition to improve communication, but the interaction efficiency is several orders of magnitude less than that of normal human interaction. Software technologies do exist to support continuous, accurate speech recognition. The problem lies on the hardware side. The speech rate supported by even the most high-end desktop microprocessors is extremely limited due the processing requirements of the software. The performance of processors that are more suitable in a portable environment are even worse. In this paper, we analyze a number of parameters affecting the potential performance of speech recognition software, with particular emphasis on the demands made of the underlying memory system. We observe that the majority of 'poorly behaved' memory accesses are directed to the knowledge based traversed during the search phases of speech recognition, suggesting potential benefit from memory space partitioning.

Section 1: Introduction

Speech recognition software technology has emerged in recent years to present a viable solution to the problems of human-computer interaction [1]. Ideally, such technology would be embedded into all invisible and portable computers to enable efficient human-computer communication. Unfortunately, computing platforms ideally situated to take advantage of speech recognition and other natural I/O capabilities, such as PDAs and other hand held

computing devices, simply do not have the memory systems and processing power to achieve a reasonable trade-off between interaction rate and recognition accuracy. Recent studies have shown that the computational demands of current generation speech recognition technology are enough to tax even high-end desktop processors [1,6]. Obviously, high-end desktop processors do not make sense in the portable domain due to their high power consumption and large size. Thus, we observe a performance gap of about 20x between the current generation of processors usable in portable environments and the computation demands of speech recognition. Further, it is reasonable to surmise that demands for performance will continue to grow with increased speech recognition accuracy and robustness.

In this paper, we evaluate the memory behavior of a standard speech recognition infrastructure. The following section will provide some introductory details on the SPHINX-II speech recognition engine from CMU, as well as a brief discussion on speech recognition in general. Section 3 covers our analysis of memory behavior of this recognition engine. Section 4 will present some insight on methods of dealing with the memory bottleneck presented by this application space, and section 5 will provide some summarizing and concluding remarks.

Section 2: Background and Related Work

A number of previous works have investigated the general impact of hardware architectures on speech recognition systems [1,5,6]. We attempt to expand on these works by focusing on the memory demands of speech recognition algorithms and exploring this behavior in further detail. Before getting into the actual evaluation, we present some details on the CMU-SPHINX speech recognition software, and on speech recognition algorithms in general.

2.1: CMU-Sphinx

The analysis presented in this paper is performed through use of the Sphinx-II speech recognition system from CMU [2,3]. Sphinx provides an infrastructure for performing large-vocabulary speaker-independent continuous speech recognition from pre-recorded input samples, and is capable of achieving 71-86% recognition accuracies. Internally, Sphinx operates on hidden-markov model representations of language model and dictionary data, utilizing algorithms that are common in this domain[8]. As such, Sphinx represents a state of the art research platform for speaker independent speech recognition, and provides an excellent test base by which to analyze behavior.

For the purposes of this work, we recognize three distinct phases of Sphinx operation: initialization, feature extraction, and search. The initialization phase involves a one time load of knowledge base (KB) data, as well as initialization of various dynamic data structures used during the later search process. While this code represents a notable portion of execution time in the sample trials run in this study, we contend that it would be insignificant (or pre-loaded) in any embedded system. Thus, we discount the effects of this phase in our statistics. The feature extraction phase processes the target sound, dividing the input into frames and producing a set of feature vectors that constitute the base data for the recognition phase of the algorithm. Once again, the operations performed in this phase could be passed off to existing hardware solutions (such as DSPs), and are thus discounted in our evaluation.¹ This leaves the back end search phase, which constitutes the majority of the steady state running time of the program. The first step of the search phase is a tree based beam search. This is followed by an optional flat lexical search, and finally an optional best path search that traverses the nodes (and word solutions) activated by the previous searches and selects the best solution path.

2.2: Speech Recognition Technology

Previous research has shown that the computational behavior of speech recognition algorithms

1. characterization of front end processing can be found in cited works [1]

runs in many ways contrary to that demonstrated by most software applications [1,5,6]. For example, recognition is generally achieved through extensive search of a large, in-memory knowledge base consisting of language models, dictionaries, phonetic information, and the like. This search is strongly dependent on the input, and often exhibits very poor locality and frequent conditional evaluations, resulting in poor cache performance and low instruction level parallelism. While data placement techniques and future advances in search algorithms and knowledge representation techniques will help mitigate this problem, the features leading to poor memory system performance are inherent to this application space and will continue to hurt performance for the foreseeable future.

Thus, we conclude that the current generation of general purpose processors, particularly those for embedded systems, are not designed to meet the needs of applications in this domain. Current memory systems depend on a very high degree of locality to addresses in a relatively confined memory space in order to be effective. Current high end processors go to great lengths to attempt to exploit instruction level parallelism, but have no real facilities by which to exploit the higher-level parallelism often present in recognition algorithms. While this work primarily focuses on memory system performance, we conclude with a more general discussion of possible techniques to improve overall algorithm performance.

Section 3: Performance Analysis

The analysis presented in this paper 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 [9]. All studies presented are based on the forward tree beam search algorithm, which is a standard technique for hidden markov model searches. Analysis of the other search options available in SPHINX-II show very similar memory system behavior (in terms of cache miss rates), with the flat lexical search performing nearly identically, and the bestpath search showing similar patterns with a lower ratio of cache misses. As the bestpath search traverses only a subset of the

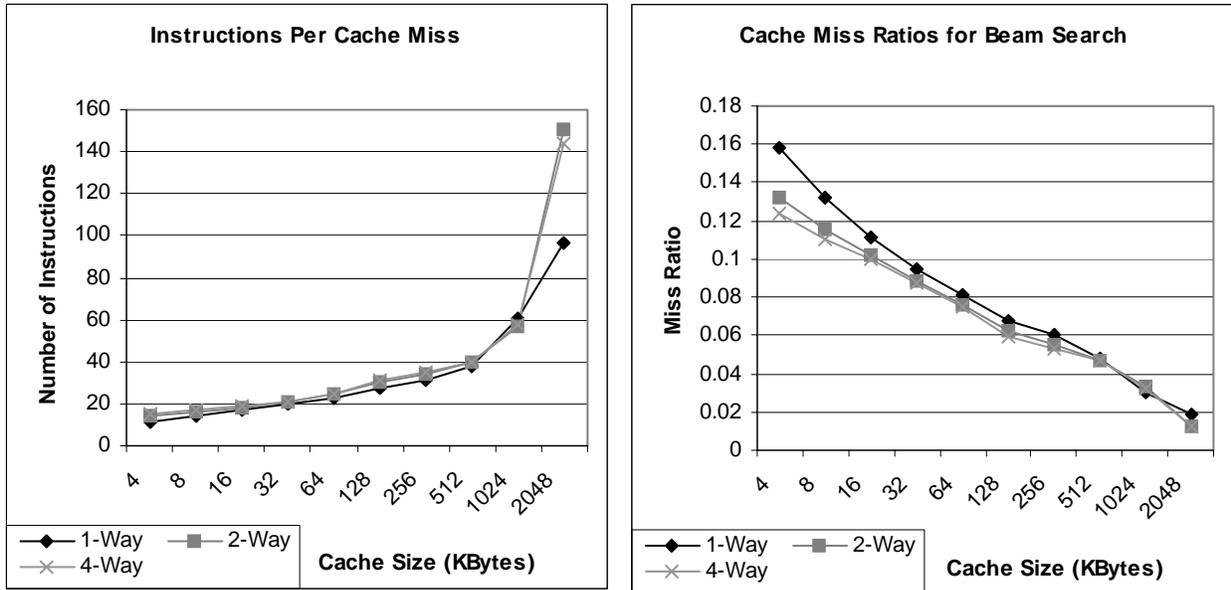


Figure 3.0: Average instructions per miss and cache miss rates of beam search on baseline configuration.

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. Figure 3.0 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 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. [11].

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 as metrics relative to the average duration of time of a spoken syllable (e.g. ‘misses per syllable’). This formulation is chosen to provide both intuitive human understanding (syllables being much more intuitive than the ‘frame’ timeslices 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.

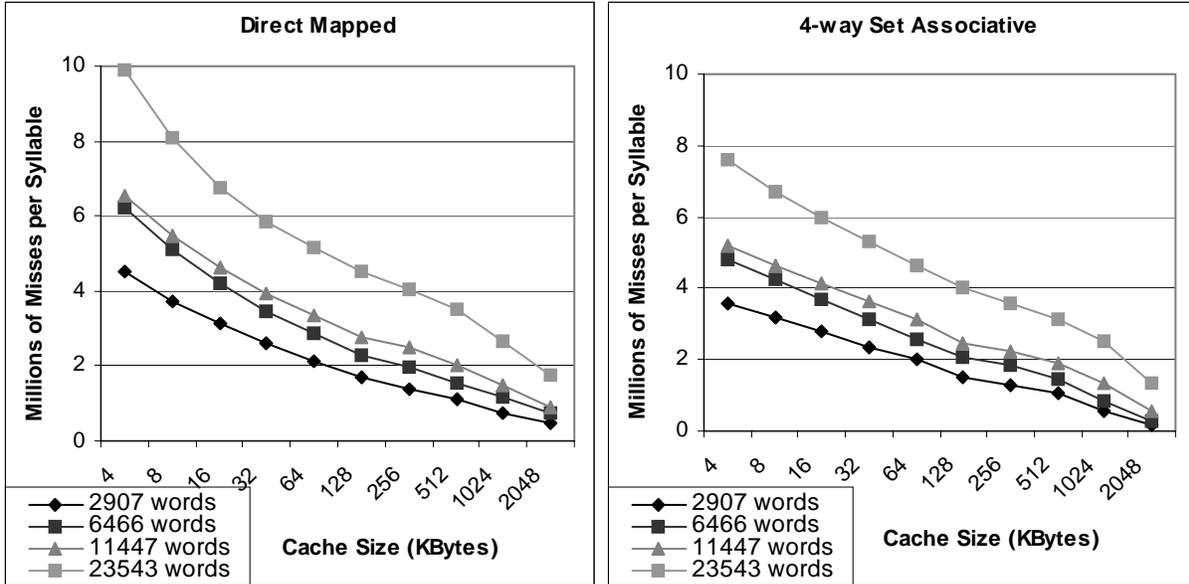


Figure 3.1: Average number of cache misses per syllable for varied dictionary sizes.

3.1: 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 3.1 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 [10]. 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 [5]. It should be noted that the actual cache miss ratios did not vary substantially across dictionary sizes.

3.2: 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 3.2, 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 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.

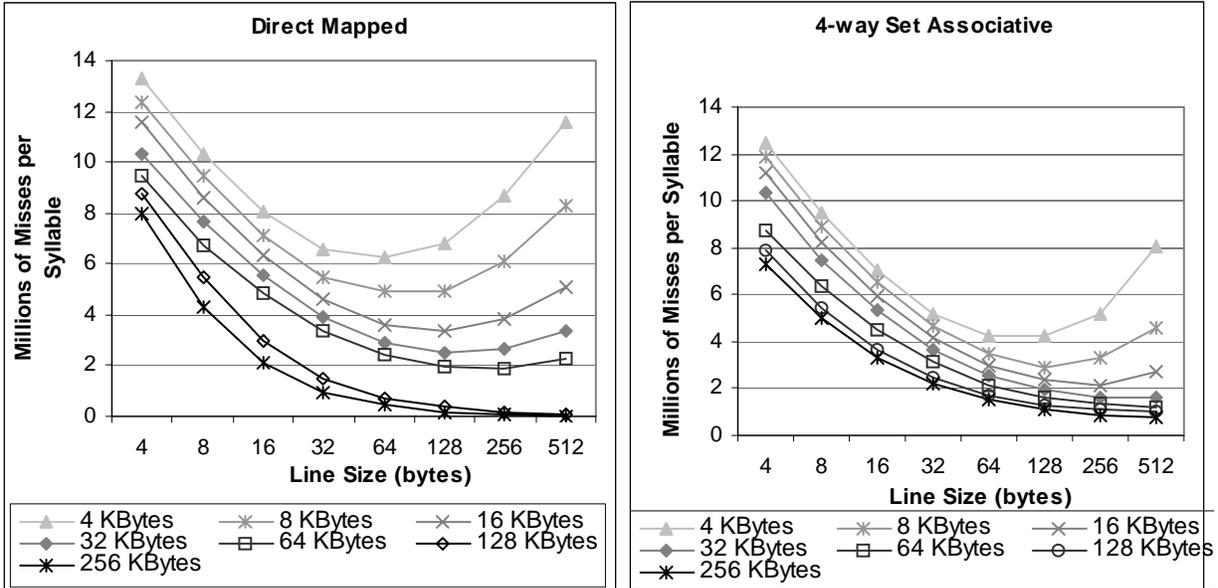


Figure 3.2: Misses per syllable for various cache sizes for a range of cache block sizes.

3.3: 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 than half the pruning threshold of ‘default’). Results of these evaluations are shown in figure 3.3 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.

3.4: 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 3.4 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

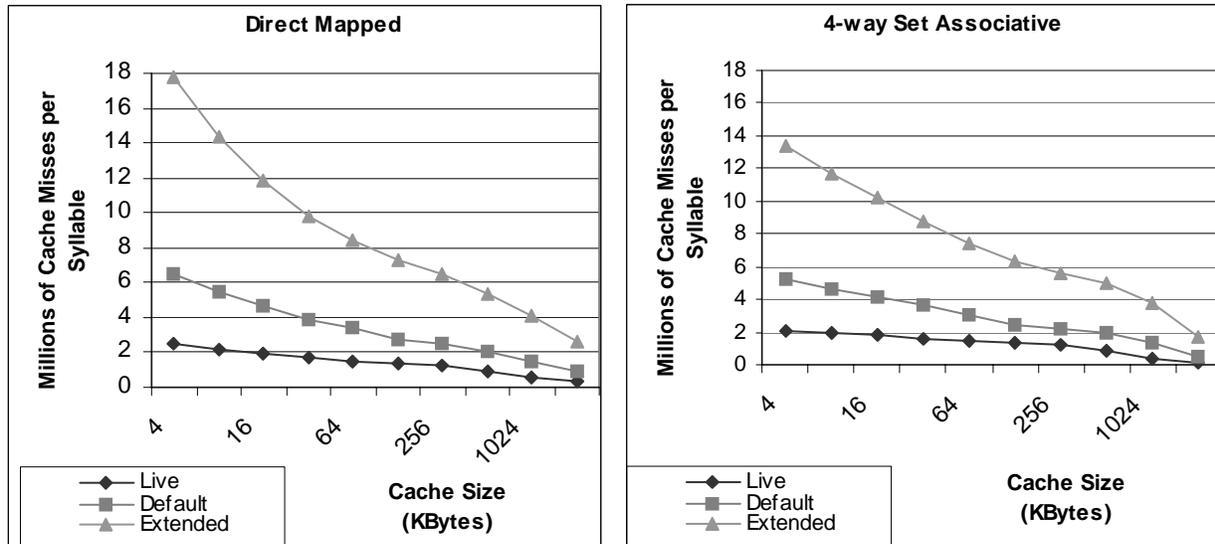


Figure 3.3: Average number of cache misses per syllable for a variety of search beam widths.

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.

3.5: 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 sentences with detailed structure.

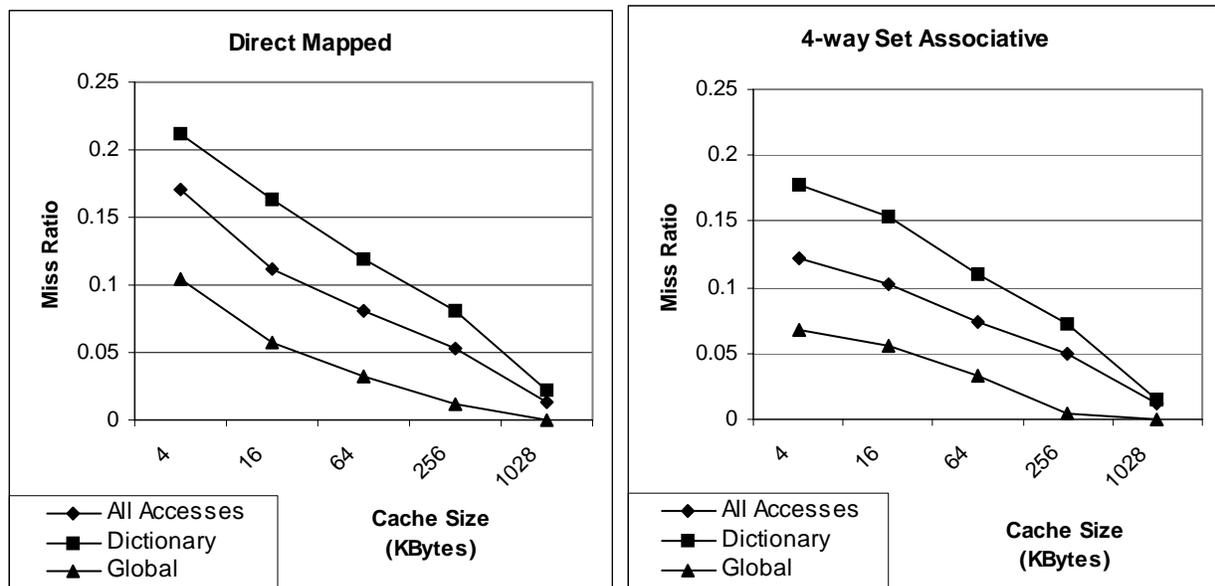


Figure 3.4: Cache miss rates for back end search, partitioned by data access type.

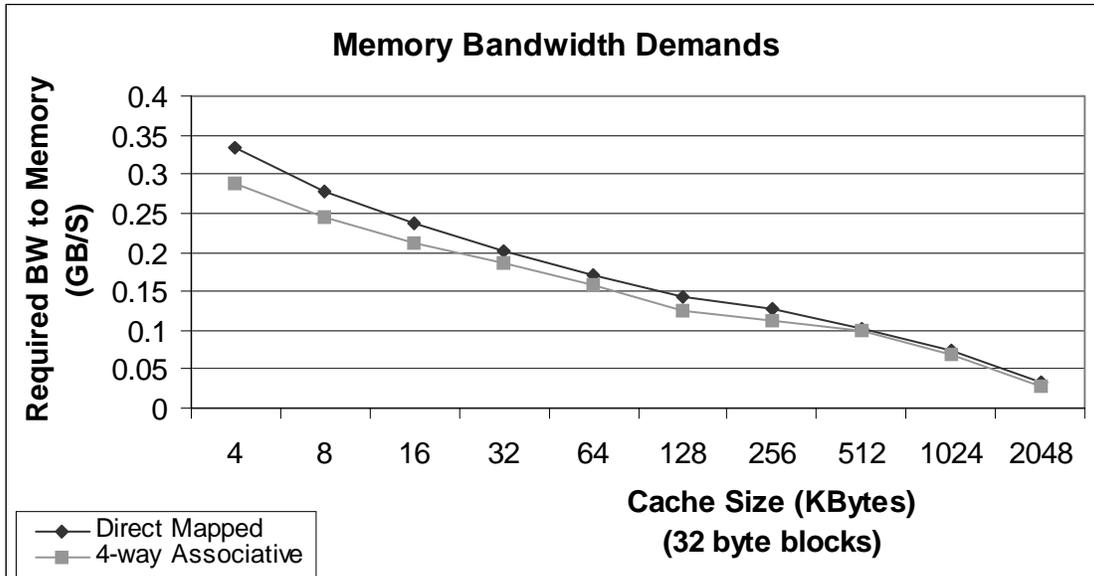


Figure 3.6: Memory bandwidth demands for beam search algorithm across a variety of L1 cache configurations.

We 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.

3.6 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. We assume a 206Mhz processor, which is a reasonable rate for current generation embedded processor systems (e.g.: the Intel SA-1110). 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.

Section 4: Strategies for Improving Speech Algorithmic Performance

The evaluations thus presented clearly demonstrate some of the constraints faced by speech recognition algorithms due to the inherent nature of current memory systems relative to the algorithm reference stream. In particular, these algorithms show poor cache performance out to rather large caches. Clearly, a two or three megabyte L1 cache is impractical and infeasible in most situations, particularly for portable or embedded systems which stand to gain the most from robust speech recognition technology. Thus, we step back and realize that there are really only two methods for dealing with the memory associated latency: the latency must either be reduced or tolerated. Thus, we consider a first level examination of two approaches, one addressing each of these techniques. We investigate the potential for prefetching by examining the memory reference stream, and the potential for parallelization of the algorithms themselves as a means of tolerating latency.

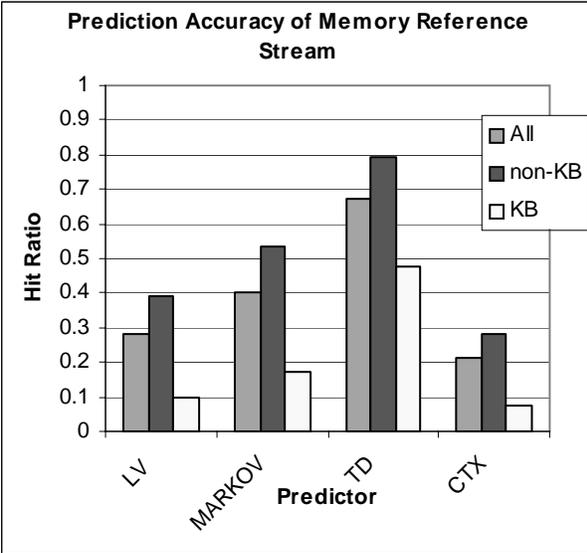


Figure 4.1.1: Prediction accuracy of raw memory reference stream.

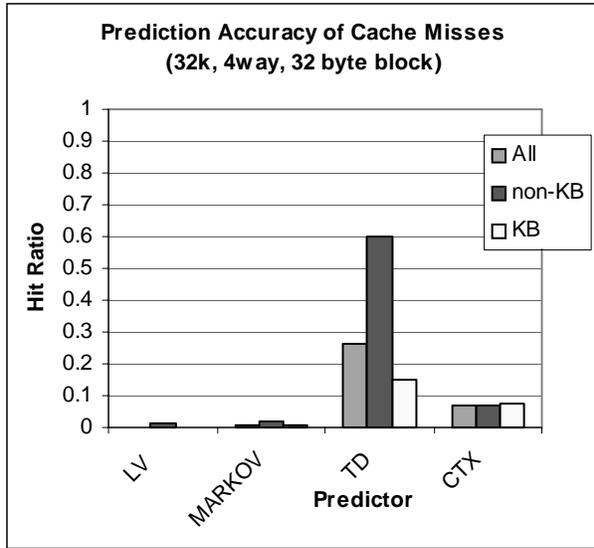


Figure 4.1.2: Prediction accuracy of memory reference stream past L1 cache.

4.1: 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 4.1.1. The “LV” predictor represents a standard last value predictor [12]. The “MARKOV” configuration represents an LRU list based history predictor [14]. The “TD” configuration represents a 2-delta stride predictor [15]. Finally, the “CTX” configuration represents a context based history predictor [13]. 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 a set cache configuration (32k, 4way, 32 byte lines) is shown in figure 4.1.2. 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, an accurate prediction may be needed tens to hundreds of cycles before the data itself is required.

4.2: 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, 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 [1]) is shown in figure 4.2.1 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

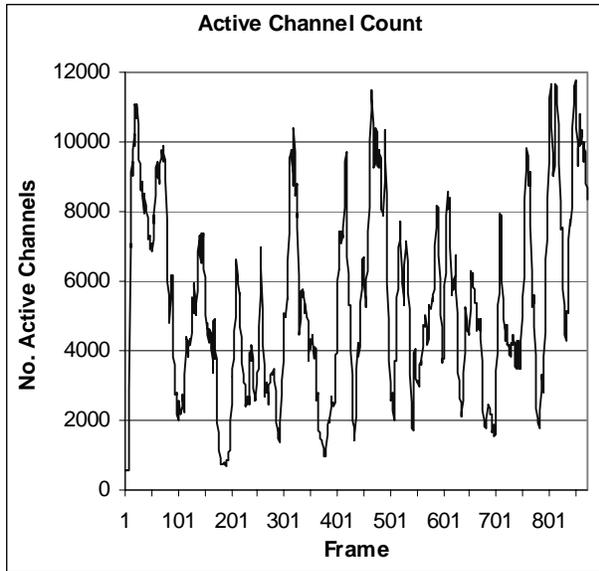


Figure 4.2.1: Map of active channels over the course of a sample run of forward tree search

[4,7]. 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 in 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 instruction 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. This is shown in figure 4.2.2, which depicts the number of memory requests made in a given cycle for 10 and 20 thread runs against the percentage of execution cycles in which that number of simultaneous memory requests was observed. This evaluation was performed on a ver-

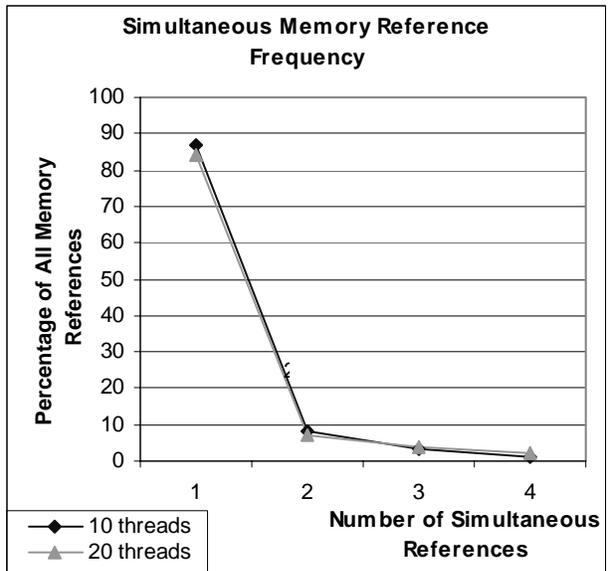


Figure 4.2.2: Percentage of simultaneous memory requests for multi-threaded executions.

sion of the SimpleScalar toolset modified to simulate a basic SMP architecture with very low threading overhead.

Section 5: Summary and Conclusions

We present an evaluation of the memory behavior of speech recognition algorithms through analysis of the SPHINX-II speech recognition software from CMU. Through this analysis, we confirm previous work suggesting that this application domain shows poor memory performance on existing memory infrastructures, and expand upon it through analysis of a wider range of parameters. An interesting finding from this analysis is the marked difference in memory performance between references to the knowledge base and references to other global memory. It is clear that references to this knowledge base are a prime contributor to the poor memory performance of the application as a whole. As expected, memory demand per syllable increases with larger dictionaries and wider searches. The impact of this, however, is mitigated with increasing dictionary size due to knowledge base compression applied by the software. We find that large block sizes matched to dictionary node size also work best. Finally, we also explore potential solutions to the bottleneck presented by memory performance, determining that prediction and pre-fetching of memory accesses at the hardware level is ineffective, but

also revealing much potential in the area of parallelization and concurrent execution.

References:

- [1] 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.
- [2] K. Lee, H. Hon, and R. Reddy, "An Overview of the SPHINX speech recognition system," *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38:35-44, 1990.
- [3] K. Huang, F. Alleva, and H. Hon, "The SPHINX-II Speech Recognition System: An Overview," Technical Report CMU-CS-92-112. Carnegie Mellon University, School of Computer Science, January 1992.
- [4] S. Chatterjee and P. Agrawal, "Connected Speech Recognition on a Multiple Processor Pipeline," *Proceedings of the 1989 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2:774-777, May 1989.
- [5] S. Vlaovic and R. Uhlig, "Performance of Natural I/O Applications," In *Proceedings of 2nd Annual Workshop on Workload Characterization*. October 1999.
- [6] C. Lai, S. Su, and Q. Zhao, "Performance Analysis of Speech Recognition Software," *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*. February 2002.
- [7] M.K. Ravishankar, "Parallel Implementation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition," *Computer science & Automation*, Indian Institute of Science, Bangalore, India., 1993.
- [8] L. Rabiner, "A tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," In *Proceedings of IEEE*, 77(2):257-286, 1986.
- [9] D. Burger and T. Austin, "The SimpleScalar Toolset Version 2.0," Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madison, June 1997.
- [10] P. Clarkson and R. Rosenfeld, "Statistical Language Modeling Using the CMU-Cambridge Toolkit," In *Proceedings of EUROSPEECH'97*, p2707-2710, 1997.
- [11] J. Cantin and M. Hill, "Cache Performance for Selected SPEC CPU2000 Benchmarks", *Computer Architecture News (CAN)*, September 2001.
- [12] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value locality and load value prediction", In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*", December 1996.
- [13] Y. Sazeides and J.E. Smith, "The predictability of data values", In *Proceedings of 20th International Symposium on Microarchitecture*, December 1997.
- [14] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors", In *Proceedings of 24th Annual International Symposium on Computer Architecture*", May 1997.
- [15] B. Calder, G. Reinman and D. Tullsen, "Selective Value Prediction", In *Proceedings of 26th Annual International Symposium on Computer Architecture*", May 1999.