MiBench: A free, commercially representative embedded benchmark suite

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst,
Todd M. Austin, Trevor Mudge, Richard B. Brown
{mguthaus,jringenb,ernstd,taustin,tnm,brown}@eecs.umich.edu
The University of Michigan
Electrical Engineering and Computer Science
1301 Beal Ave., Ann Arbor, MI 48109-2122

Abstract

This paper examines a set of commercially representative embedded programs and compares them to an existing benchmark suite, SPEC2000. A new version of SimpleScalar that has been adapted to the ARM instruction set is used to characterize the performance of the benchmarks using configurations similar to current and next generation embedded processors. Several characteristics distinguish the representative embedded programs from the existing SPEC benchmarks including instruction distribution, memory behavior, and available parallelism. The embedded benchmarks, called MiBench, are freely available to all researchers.

1. Introduction

Performance based design has made benchmarking a critical part of the design process [1]. A wide variety of benchmarks have been proposed including Dhrystone [2], LINPACK [3], Whetstone [4], CPU2 [5], MediaBench [6] and many others. Most of these benchmarks are targeted towards specific areas of computation. Dhrystone, for example, is for system (integer) performance; LINPACK is for vectorizable computations; Whetstone and CPU2 are for numerical (floating point) intensive applications; and MediaBench is for multimedia applications. Other benchmarks are available to stress network TCP/IP stacks, data input/output and other specific tasks.

The most widely used benchmarks are the Standard Performance Evaluation Corporation (SPEC) CPU benchmarks [7]. They are now in their third revision (SPEC2000). They characterize a workload for general-purpose computers by providing a self-contained set of programs and data divided into separate integer and floating-point categories. The popularity of the SPEC benchmarks as a measure of performance has heavily influenced the design of general-purpose microprocessors, particularly those employed in servers and high-end desktop systems.

Among the common features of these microarchitectures are deep pipelines, significant instruction level parallelism, sophisticated branch prediction schemes, and large caches.

Although this class of machines has been the chief focus of the computer architecture community, relatively few microprocessors are employed in this market segment. The vast majority of microprocessors are employed in embedded applications [8]. Although many are just inexpensive microcontrollers, their combined sales account for nearly half of all microprocessor revenue. Furthermore, the embedded application domain is the fastest growing market segment in the microprocessor industry.

The wide range of applications makes it difficult to characterize the embedded domain. In fact, an embedded benchmark suite should reflect this by emphasizing diversity. The applications range from sensor systems on simple microcontrollers [9] to smart cellular phones that have the functionality of a desktop machine combined with support for wireless communications. Perhaps the only common denominators are: 1) that embedded processors often require power to be considered at the same time in the design process as performance [11]; and 2) that there is not a significant legacy code base that would favor a standard instruction set architecture (ISA) and operating system, as has happened in the desktop world. This has led to a remarkable increase in the number of ISAs for embedded applications and this number continues to grow.

There have been some efforts to characterize embedded workloads, most notably the suite developed by the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [10]. They have recognized the difficulty of using just one suite to characterize such a diverse application domain and have instead produced a set of suites that typify workloads in five embedded markets. Unfortunately, the EEMBC benchmarks, unlike SPEC, are not readily accessible to academic

researchers because of the high cost of joining the consortium.

In this paper, we present a set of 35 embedded applications for benchmarking purposes called MiBench (pronounced "My Bench"). Following EEMBC's model, these benchmarks are divided into six suites with each suite targeting a specific area of the embedded market. The six categories are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. All the programs are available as standard C source code. Since many past embedded applications have been written directly in assembly language, it has been difficult to collect a portable set of benchmarks for the embedded domain. However, the current trend in the embedded domain shows compilers being used for even the simplest microcontrollers and the highest performance DSPs. MiBench is thus portable to any platform that has compiler support.

The rest of this paper is organized as follows. Section 2 describes the benchmarks and data sets in MiBench. Section 3 validates the microarchitecture model of the ARM SA-1 core; an important step that is often omitted from the discussion of benchmark performance on benchmarks. Section 4 provides an analysis of the MiBench benchmarks and compares them to the SPEC2000 benchmarks. As one would expect from the way they were selected, the MiBench programs exhibit a much wider variance in behavior across the set of suites as well as within individual domains. This suggests that SPEC is not the appropriate workload to drive the design of future microprocessors intended for many of the embedded application categories found in MiBench. Instruction distribution, branch predictability, and memory accesses are all examined. Section 5 provides a summary of the characteristics of embedded workloads found in our experiments.

2. Benchmark Descriptions

MiBench has many similarities to the EEMBC benchmark suite as described on their web site (http://www.eembc.com). However, MiBench is composed of freely available source code. All web sites and authors are maintained with each package, but slight modifications may have been made to the source code to promote the portability of the benchmark and the extensibility of the data set. Where appropriate, we provide a small and large data set. The small data set represents a light-weight, useful embedded application of the benchmark, while the large data set provides a more stressful, real-world application. MiBench consists of six categories including: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. These categories offer different program characteristics that enable researchers in architecture and compilers to examine their designs more effectively for a particular market segment.

2.1. Automotive and Industrial Control

The Automotive and Industrial Control category is intended to demonstrate use of embedded processors in embedded control systems. These processors require performance in basic math abilities, bit manipulation, data input/output and simple data organization. Typical applications are air bag controllers, engine performance monitors and sensor systems. The tests used to characterize these situations are a basic math test, a bit counting test, a sorting algorithm and a shape recognition program.

basicmath: The basic math test performs simple mathematical calculations that often don't have dedicated hardware support in embedded processors. For example, cubic function solving, integer square root and angle conversions from degrees to radians are all necessary calculations for calculating road speed or other vector values. The input data is a fixed set of constants.

bitcount: The bit count algorithm tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers. It does this using five methods including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, non-recursive bit count by bytes using a table look-up and shift and count bits. The input data is an array of integers with equal numbers of 1's and 0's.

qsort: The *qsort* test sorts a large array of strings into ascending order using the well known quick sort algorithm. Sorting of information is important for systems so that priorities can be made, output can be better interpreted, data can be organized and the over-all run-time of programs reduced. The small data set is a list of words; the large data set is a set of three-tuples representing points of data.

susan: *Susan* is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Images of the brain. It is typical of a real world program that would be employed for a vision based quality assurance application. It can smooth an image and has adjustments for threshold, brightness, and spatial control. The small input data is a black and white image of a rectangle while the large input data is a complex picture.

Table 1: MiBench Benchmarks

Auto./Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	GSM dec.
	typeset				

2.2. Network

The Network category represents embedded processors in network devices like switches and routers. The work done by these embedded processors involves shortest path calculations, tree and table lookups and data input/output. The algorithms used to demonstrate the networking category are finding a shortest path in a graph and creating and searching a Patricia trie data structure. Some of the benchmarks in the Security and Telecommunications category are also relevant to the Network category: *CRC32*, *sha*, and *blowfish*. However, they are separated for organization.

dijkstra: The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm. Dijkstra's algorithm is a well known solution to the shortest path problem and completes in $O(n^2)$ time.

patricia: A Patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the trie to reduce traversal time at the expense of code complexity. Often, Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a 2 hour period. The IP numbers are disguised.

2.3. Security

Data Security is going to have increased importance as the Internet continues to gain popularity in e-commerce activities. Therefore, Security is given its own category in MiBench. The Security category includes several common algorithms for data encryption, decryption and hashing. One algorithm, *rijndael*, is the new Advanced Encryption Standard

(AES) [12]. The other representative security algorithms are Blowfish [13], PGP [15] and SHA [14].

blowfish encrypt/decrypt: *Blowfish* is a symmetric block cipher with a variable length key. It was developed in 1993 by Bruce Schneider. Since its key length can range from 32 to 448 bits, it is ideal for domestic and exportable encryption. The input data sets are a large and small ASCII text file of an article found online.

sha: *SHA* is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures. It is also used in the well-known MD4 and MD5 hashing functions. The input data sets are the same as the ones used by *blowfish*.

rijndael encrypt/decrypt: *Rijndael* was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks. The input data sets are the same as the ones used by *blowfish*.

pgp sign/verify: Pretty Good Privacy (PGP) is a public key encryption algorithm developed by Phil Zimmerman. It allows you to communicate securely with people you've never met using digital signatures and the RSA public key cryptosystem. The input data for both the large and small tests is a small text file. This is because *PGP* is usually only used to securely exchange a key for a block cipher which can then encrypt/decrypt data at a much faster rate.

2.4. Consumer Devices

The Consumer Devices benchmarks are intended to represent the many consumer devices that have grown in popularity during recent years like scanners, digital cameras and Personal Digital Assistants (PDAs). The

category focuses primarily on multimedia applications with the representative algorithms being jpeg encoding/decoding, image color format conversion, image dithering, color palette reduction, MP3 encode/decoding, and HTML typesetting. Several of the algorithms are from the SGI TIFF utilities [16]. All of the image benchmarks use small and large images as data input.

jpeg encode/decode: JPEG is a standard, lossy compression image format. It is included in MiBench because it is a representative algorithm for image compression and decompression and is commonly used to view images embedded in documents. The input data are a large and small color image.

tiff2bw: *Tiff2bw* converts a color TIFF image to black and white image.

tiff2rgba: *Tiff2rgba* converts a color image in the TIFF format into a RGB color formatted TIFF image.

tiffdither: *Tiffdither* dithers a black and white TIFF bitmap to reduces the resolution and size of the image at the expense of clarity.

tiffmedian: *Tiffmedian* converts an image to a reduced color palette by taking several medians of the current color palette.

lame: *Lame* is a GPL'ed MP3 encoder that supports constant, average and variable bit-rate encoding. It uses small and large wave files for its data inputs.

mad: *Mad* is a high-quality MPEG audio decoder. It currently supports MPEG-1 and the MPEG-2 extension to Lower Sampling Frequencies, as well as the so-called MPEG 2.5 format. All three audio layers (Layer I, Layer II, and Layer III a.k.a. MP3) are fully implemented. It uses small and large MP3s for its data inputs.

typeset: *Typeset* is a general typesetting tool, that has a front-end processor for HTML. The benchmark captures the processing required to typeset an HTML document, without any rendering overheads. This benchmark is representative of a core component of a web browser that might be used in a consumer device. The small and large inputs are a GCC release announcement and the SimpleScalar main web page.

2.5. Office Automation

The Office applications are primarily text manipulation algorithms to represent office machinery like printers, fax machines and word processors. The PDA market mentioned in the Consumer category also relies heavily on the manipulation of text for data organization.

ghostscript: *Ghostscript* [17] is a postscript language interpreter without its graphical interface. This benchmark is included to represent the growing importance of postscript capable embedded devices like printers.

stringsearch: This benchmark searches for given words in phrases using a case insensitive comparison algorithm.

ispell: *Ispell* is a fast spelling checker that is similar to the Unix spell, but faster. It supports contextual spell checking, correction suggestions, and languages other than English. The input consists of a small and large document from web pages.

rsynth: *Rsynth* is a text to speech synthesis program that integrates several pieces of public domain code into a single program. The small and large input are excerpts from an online news article.

sphinx: *Sphinx* is a speech decoder that operates on finite-length segments of speech or utterances, one utterance at a time. An utterance can be up to some tens of seconds long. The small and large inputs are a simple command and a long sequence of speech.

2.6. Telecommunications

Close beside the Consumer category for importance in modern embedded processors is the Telecommunications category. With the explosive growth of the Internet, many portable consumer devices are integrating wireless communication. The Telecommunication benchmarks are given a separate category to stress the importance of this step. These benchmarks consist of voice encoding and decoding algorithms, frequency analysis and a checksum algorithm.

FFT/IFFT: This benchmark performs a Fast Fourier Transform and its inverse transform on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.

GSM encode/decode: The Global Standard for Mobile (GSM) communications [18] is the standard for voice encoding/decoding in Europe and many countries. It uses a combination of Time- and Frequency-Division Multiple Access (TDMA/FDMA) to encode/decode data streams. The input data is small and large speech samples.

ADPCM encode/decode: Adaptive Differential Pulse Code Modulation (ADPCM) [19] is a variation of the well-known standard Pulse Code Modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The input data are small and large speech samples.
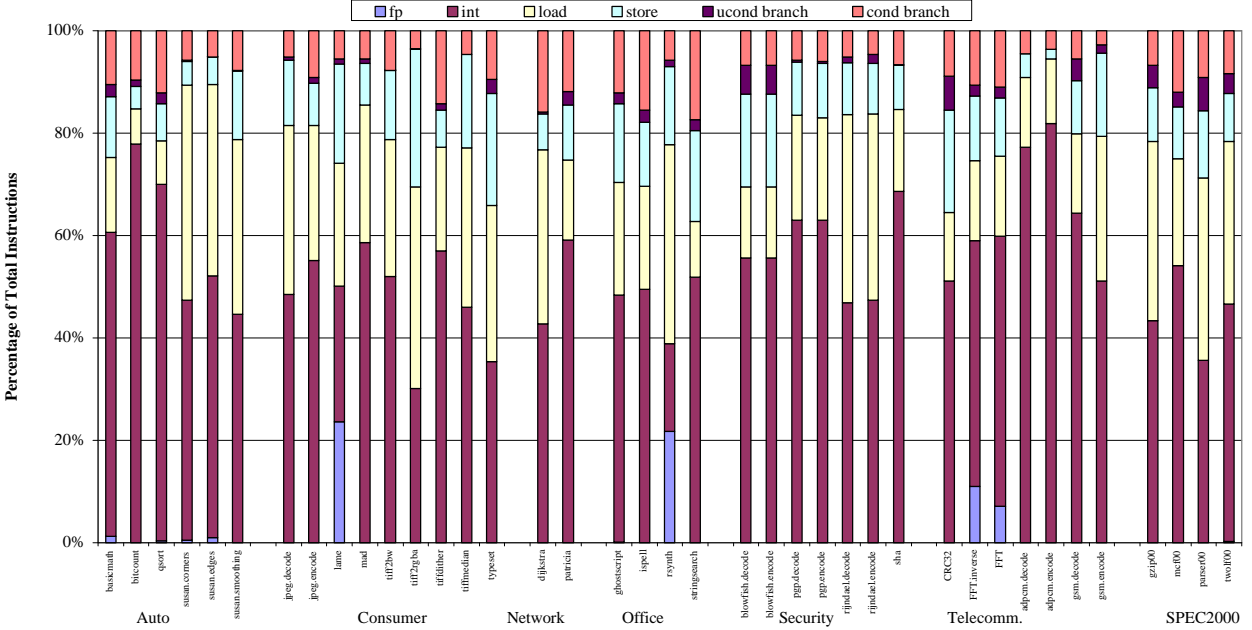
Figure 1: Dynamic Instruction Distribution for large data set.

CRC32: This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. The data input is the sound files from the ADPCM benchmark.

3. Microarchitecture Model Validation

The Current configuration in Table 3 is modeled after Intel's SA-1 StrongARM pipeline [22], found in the SA-11xx series of embedded microprocessors. Intel has released few details of the SA-1 pipeline; our model was constructed using pipeline timing characteristics given in the SA-110 compiler writers' guide [24]. In addition, we used microbenchmarks to accurately measure fully exposed pipeline latencies such as branch mispredictions and cache misses. We validated our model against a Rebel NetWinder Developer workstation [25]. The NetWinder contains a 275 MHz StrongARM SA-110 microprocessor, 128 MB of DRAM, and an Ethernet interface. It runs the Linux operating system (version 2.2.13) with a standard GNU tool chain including GCC (version 2.95.1). The run times of integer microbenchmarks, kernels (e.g., *FFT*), and large benchmarks (e.g., *bzip* and *GCC*) were measured on the NetWinder and compared to their simulated performance on the SA-1 ARM model. The simplicity of the SA-1 pipeline and memory system permitted us to construct an extremely accurate timing model with only a few modifications to the SimpleScalar/ARM performance simulator. The

largest measured error in performance (CPI) was only 3.2%. We were unable to fully validate our floating point co-processor model because the NetWinder does not include floating point support in hardware. We will address this validation effort when reference platforms and suitable floating point benchmarks become available.

4. Benchmark Analysis

All benchmarks in SPEC2000 and MiBench were compiled using GCC version 2.95.2 on a Debian Linux 2.2.18 workstation with optimizations enabled. All the benchmarks were simulated using the SimpleScalar/ ARM [20] performance simulator with a configuration similar to the Intel XScale microcontroller. Only the integer SPEC2000 benchmarks were used for comparison, because most embedded processors do not have significant floating point capabilities. A limited set of the integer SPEC benchmarks ran correctly on ARM, so these were used as data points. Up to 1 billion dynamic instructions were simulated for all benchmarks. The reference data set was used for the SPEC input. The small data sets for MiBench are approximately 50 million dynamic instructions while the large data set has more than 750 million dynamic instructions as shown in Table 2. Cache performance data was gathered by simulating the memory references of all the benchmarks using Cheetah [21]. Cheetah is able to simulate multiple cache

Table 2: Benchmark Sizes

Benchmark	Small Instruction Count	Large Instruction Count	Benchmark	Small Instruction Count	Large Instruction Count
basicmath	65,459,080	1,000,000,000	ispell	8,378,832	640,420,106
bitcount	49,671,043	384,803,644	rsynth	57,872,434	85,005,687
qsort	43,604,903	595,400,120	stringsearch	158,646	38,960,051
susan.corners	1,062,891	586,076,156	blowfish.decode	52,400,008	737,920,623
susan.edges	1,836,965	732,517,639	blowfish.encode	42,407,674	246,770,499
susan.smoothing	24,897,492	1,000,000,000	pgp.decode	85,006,293	259,293,845
jpeg.decode	6,677,595	990,912,065	pgp.encode	38,960,650	824,946,344
jpeg.encode	28,108,471	543,976,667	rijndael.decode	23,706,832	140,889,705
lame	175,190,457	544,057,733	rijndael.encode	3,679,378	24,910,267
mad	25,501,771	272,657,564	sha	13,541,298	20,652,916
tiff2bw	34,003,565	697,493,266	CRC32	52,839,894	61,659,073
tiff2rgba	36,948,939	1,000,000,000	FFT.inverse	65,667,015	377,253,252
tiffdither	273,926,642	1,000,000,000	FFT	52,625,918	143,263,412
tiffmedian	141,333,005	817,729,663	adpcm.decode	30,159,188	151,699,690
typeset	23,395,912	84,170,256	adpcm.encode	37,692,050	832,956,169
dijkstra	64,927,863	272,657,564	gsm.decode	23,868,371	548,023,092
patricia	103,923,656	1,000,000,000	gsm.encode	55,361,308	472,171,446
ghostscript	286,770,117	673,391,179			

configurations in a single pass. Branch prediction was simulated using sim-bpred.

4.1. Instruction Distribution

There are four main classes of instructions: control (unconditional and conditional branches), integer, floating point and memory (load and store). In embedded applications, there are computation intensive, control intensive and I/O intensive applications. Control intensive programs will have a much larger percentage of branch instructions. Computation intensive applications will have a larger percentage of integer or floating point ALU operations. I/O applications depend on how the data is manipulated during its transfer. Figure 1 shows the distribution of all the MiBench programs and SPEC2000.

From the figure, the benchmark categories demonstrate some of these distinctive characteristics. The Telecommunication and Security benchmarks all have more than 50% integer ALU operations. These

applications tend to find or generate entropy in a set of data and is done by repeated operations on a datum. Benchmarks like *ADPCM encode/decode* have approximately 80% integer ALU operations compared to a maximum of 57% for any of the SPEC benchmarks. The Consumer category has relatively few integer ALU operations, but performs many memory operations. This is because of the large image data that must be processed. The actual operation on each part of the image is relatively straightforward and few control instructions are needed. The Office Automation benchmarks have many control and memory operations. These programs use function calls to string libraries to manipulate ASCII data. Because the data is text, it occupies quite a bit of memory and many memory operations are needed to reference it. The SPEC benchmarks have approximately the same distribution for all the benchmarks.

As shown previously, the MiBench categories are representative of different embedded applications. The
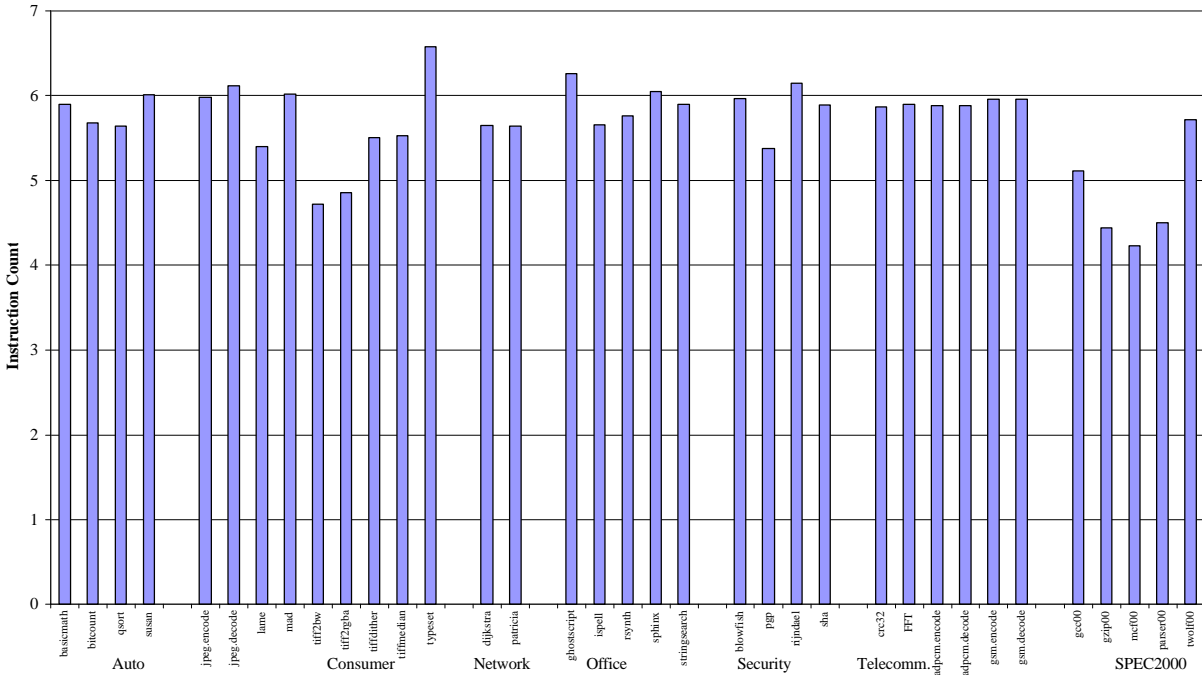
Figure 2: Static basic block length of benchmarks.

entire benchmark suite shows even more variation when considered as a whole. For example, the number of branches varies quite a bit in MiBench. *Bitcount*, *ADPCM encode* and several others also have fewer than 10% branch operations for a very computation intensive operation. Benchmarks like *Blowfish*, *tiff2rgba* and *tiffmedian* have less than 6% branches. The largest number of branches come from the text benchmarks, *stringsearch* and *ispell*, and the telecommunication benchmark, *CRC32*, which have branches ranging from 18 to 20%. SPEC typically has greater than 15% branches except *gzip00* which has only 9%. MiBench also has more variation in the memory operations. Some MiBench benchmarks, like *GSM encode*, *tiff2rgba* and typset, contain more than 50% memory operations, while others, like *Bitcount* and *ADPCM encode*, contain very few. Most of the SPEC benchmarks have about 40% memory operations.

The distribution graph also shows that MiBench has a few floating point instructions in *lame*, *rsynth* and the *FFT* benchmarks. These are not intended to stress floating point operations, but they demonstrate typical situations in which some floating point calculations might be used to control road speed, a vector direction or other data needed to determine a control action. DSP and numerical intensive processors should use a

specific floating point benchmark to analyze the performance in detail.

4.2. Branches

MiBench has quite a variation in the number of branches. The number of branches is small for a number of benchmarks which leads us to Figure 2. This figure shows that the static basic block size in the MiBench programs is approximately 1 instruction longer than SPEC. SPEC's basic block length is normally around 4.5 with *twolf00* being the exception at over 5.5. MiBench, however, has several programs above 6 and almost all are above 5.5. There are a few of the Consumer benchmarks that are below 5 like SPEC.

Now that we have seen that MiBench has more variation than SPEC2000 in the frequency of branches, we can determine how well these branches can be predicted. Simulations were run using a not-taken prediction scheme, an 8k gshare predictor, an 8k bimodal predictor and an 8k combined bimodal/2-level predictor. The direction prediction rates are given in Figure 3. All predictors used a 2k BTB except the not-taken strategy. There was no appreciable increase in prediction misses due to address mispredictions by the BTB so this data is not shown.

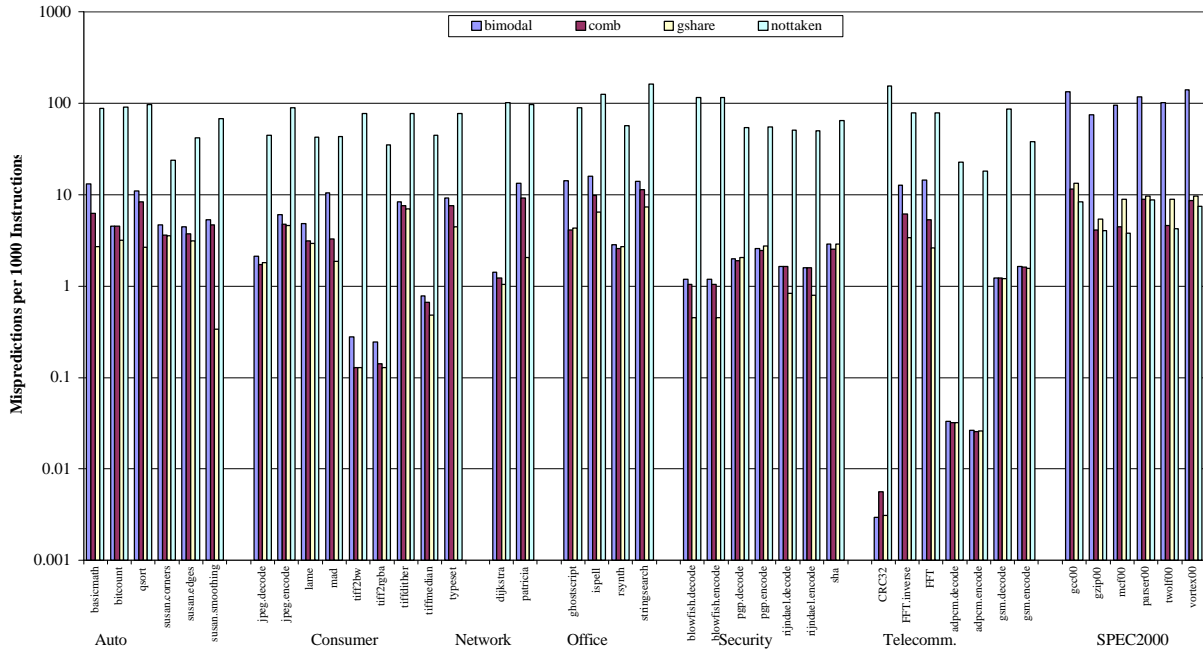Like SPEC2000, MiBench has many correct prediction rates well over 90%. The branch predictors

Figure 3: Branch prediction rates for several schemes.

are very large and, therefore, this shows that most branches in both benchmark suites are predictable. MiBench does have a few benchmarks with interesting branch characteristics though. The Telecommunications and Security benchmarks have fairly high miss rates due to randomness of data, but very few misses per thousand instructions due to the large number of integer ALU operations. The infrequency of branches dilutes the penalty due to mispredicted branches. The other categories like Automotive/Industrial look very similar to SPEC.

4.3. Memory

Besides instruction distribution and branch predictability, memory behavior is another important consideration when evaluating an embedded workload. Static memory size and memory cachability were compared with SPEC. In Figure 4, the text and data segment sizes of MiBench and SPEC2000 are shown. The two benchmark suites are similarly sized, but SPEC2000 has slightly larger segments in most cases. Again, though, MiBench contains more variation. It has a benchmark with a large 1 Mb text segment (*Ghostscript*) like the large 2 Mb text segment of *gcc00*. It also has several benchmarks with several megabyte data segments (*typeset*, *sphinx*, *PGP*). The largest SPEC data segment in these benchmarks is around 0.5 Mb (*gzip00*). It is not known why the *lame* segments are so large, but it may be that large tables are stored rather than recomputed during compression.

The large variations in MiBench's data segment is due to the large number of immediate values (constants) embedded in the source code. Embedded applications will sometimes have large data tables for lookups or interpolation. Generally though, embedded applications have small memory for both data and instruction segments which can be seen in the common case text and data segment sizes. Text sizes around 175-200 kilobytes are very normal and mostly due to inclusion of C standard libraries. Data segments are generally even smaller at several kilobytes.

As shown above, the benchmarks in MiBench have a variety of data set sizes. Because of this, MiBench will have similar cache miss rates on some benchmarks and much less cache miss rates on other benchmarks. Figure 5 and Figure 6 show the cache miss rates with varied associativity and number of sets for some benchmarks in MiBench. *gzip00*'s miss rate (not shown) levels off at just over 0.11 while *tiff2rgba* levels off around 0.05. As mentioned previously, these are the maximum miss rates for any program simulated in the benchmark sets. Other benchmarks have fewer misses and look more like Figure 5. The most frequent miss rate plots have negligible miss rates with associativity greater than 4-way or size greater than 8 kilobyte.

Also from Figure 5, the miss rates drop drastically to less than 2% around 4 to 8 kilobytes for most of MiBench. Some SPEC2000 benchmarks have more of a capacity issue and don't reduce miss rates to below

2% until around 16 to 32 kilobytes which is slightly larger. Some benchmarks like *gcc00* and *patricia* require more associativity due to random access patterns. These working sets require more associativity, but 8-way is sufficient to lower the miss rate substantially.

Embedded processor caches are typically small except in multimedia applications. Embedded applications reuse data so that cache performance is good. Data sets are also typically either fixed or stream based. The 32-way cache used in the Current and Next Generation configurations is unnecessary for the benchmarks simulated. All the benchmarks in MiBench have very few misses with more than 4- or 8-way caches and the number of sets is sufficient at 256-512 as described previously.

4.4. Benchmark Performance

To do an analysis of IPC in MiBench, three different microarchitectures were simulated with SimpleScalar/ARM. The configurations of these machines are all shown in Table 3. The "Current" configuration is modeled after published information on the Intel ARM SA-1 microarchitecture [22]. Similarly, the "Next Generation" configuration is modeled after published information on the next generation Intel ARM Xscale microarchitecture [23] and the "High-end" configuration is modeled after the Compaq Alpha 21264 microarchitecture.

The results of the simulations with each different configuration is shown in Figure 7. The greatest IPC values come from the image manipulation and multimedia related applications like *tiff2rgba*, *JPEG decode*, *tiffmedian*, *gzip00* and *mcf00*. The lowest IPC values are *Blowfish*, *typeset* and *CRC32*, which have many data dependencies due to the nature of the encryption, encoding and hashing algorithms. Surprisingly, *ADPCM* and Sphinx do relatively well even though they should have similar dependencies. The High-end architecture does considerably better than the Current or Next Generation embedded architectures. It normally achieves 2 to 3 times the IPC of the Current and Next Generation configurations.

The Current and Next Generation configurations have very similar performance on most of the benchmarks. The Next Generation architecture has a deeper pipeline, a Bimodal branch predictor and double the cache of the Current architecture. Since most of the branches in MiBench and SPEC are easily predictable, the Current configuration suffers a loss of parallelism by predicting not taken. This can be seen slightly in Figure 7, but it doesn't account for the poor performance of the Next Generation system. As shown earlier, most of the benchmarks are easily cachable, therefore, the performance loss cannot be due to cache problems. The loss in performance must be due to the in-order execution and lack of functional units. Since most benchmarks in MiBench have large basic blocks
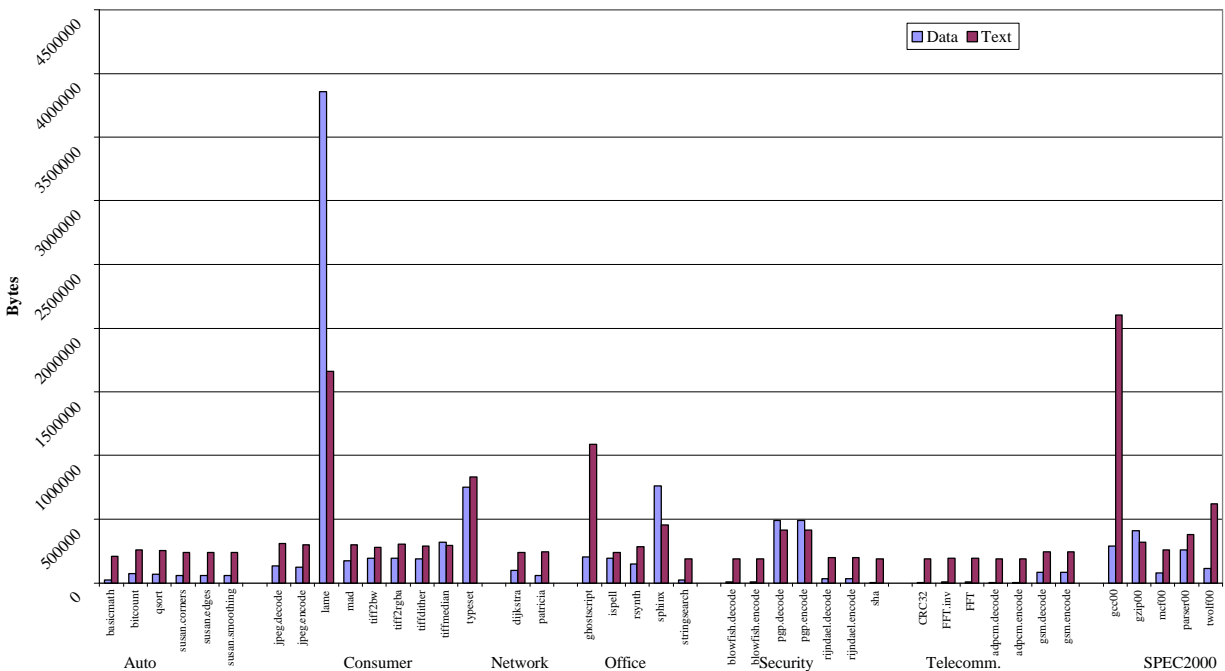

Figure 4: Text and data segment sizes. prediction rates for several schemes.
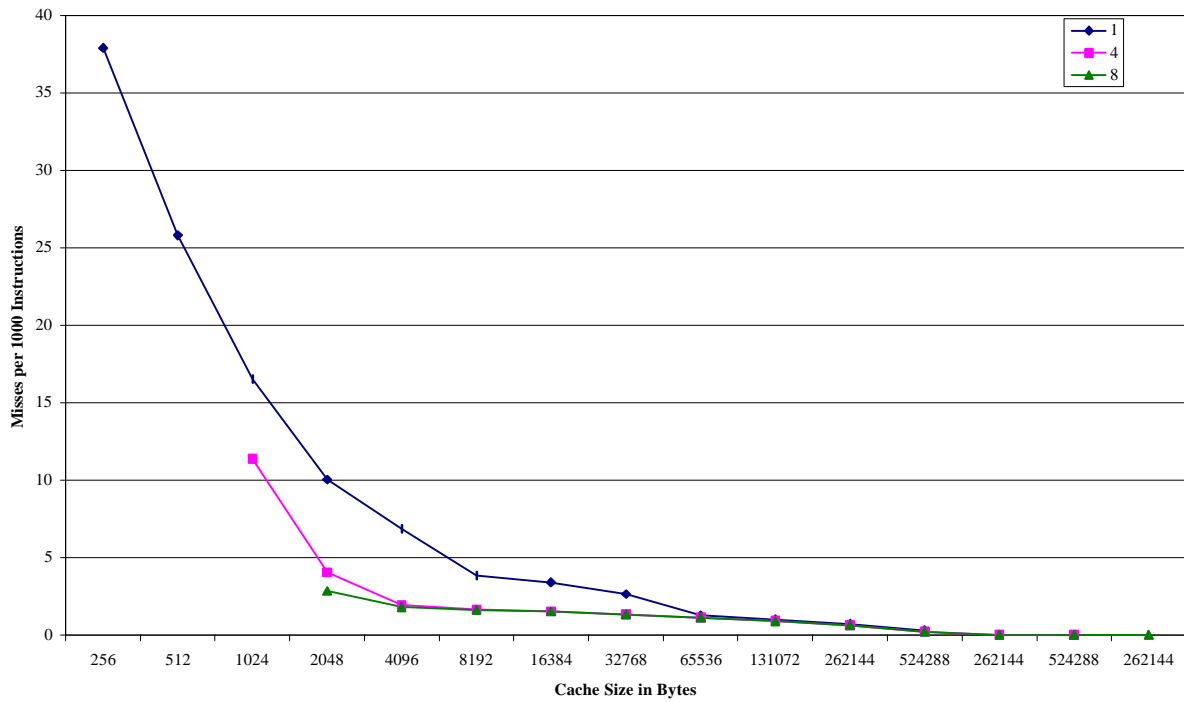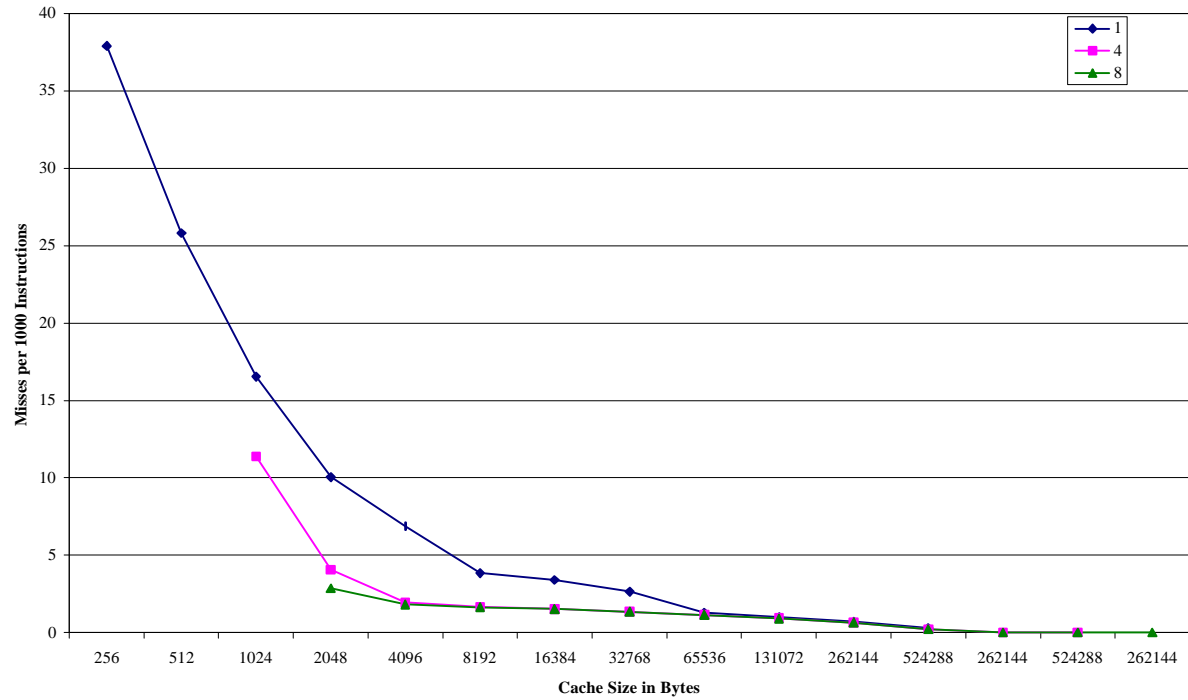
Figure 5: Cache miss rates for *Rijndael (top)* and *ispell (bottom) with 16 byte lines*.

and easily predictable branches, there is likely just not enough resources to execute all the parallel instructions.

5. Conclusions

Embedded processor design requires knowledge of the embedded task to develop an efficient microarchitecture. MiBench shows considerably
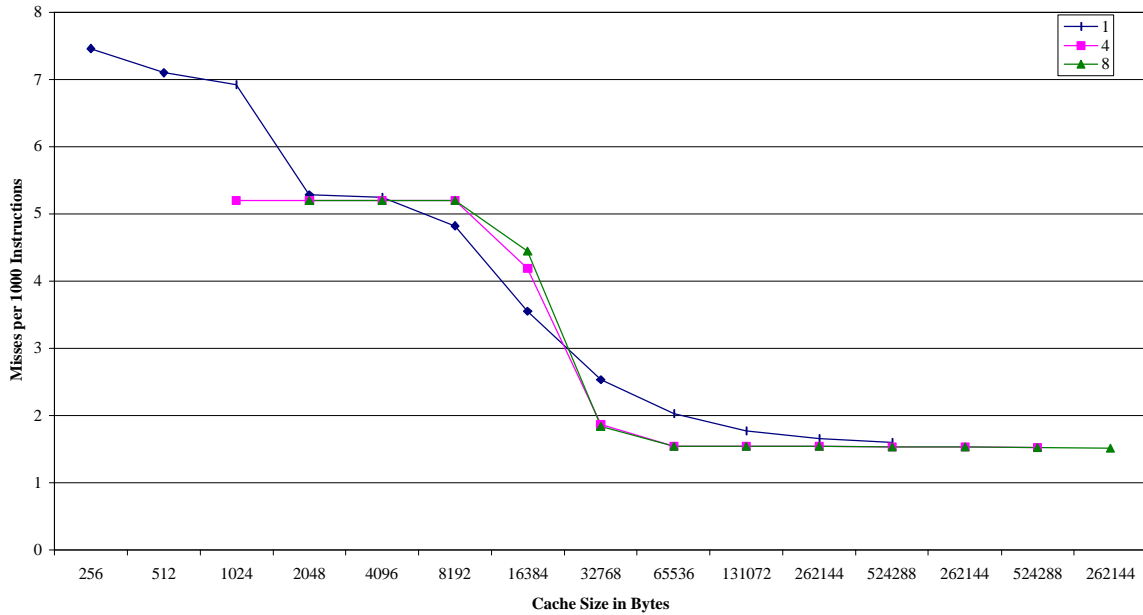
Figure 6: Cache miss rates for *tiff2rgba with 16 byte lines.*

Table 3: ARM Configurations

	Current	Next Generation	High-end
Fetch queue (instructions)	2	4	32
Branch Predictor	Not-taken	8k Bimodal, 2k 4-way BTB	Combining: 4k Bimodal, 4k Gshare, 1k 4-way BTB
Fetch & Decode width	1	1	4
Issue width	1 (In-order)	1 (In-order)	4 (Out-of-order)
Functional units	1 int ALU, 1 FP mult, 1 FP ALU	1 int ALU, 1 FP mult, 1 FP ALU	1 int ALU, 1 FP mult, 4 FP ALU
Instruction L1 Cache	16 k, 32-way	32 k, 32-way	64 k, 2-way
Data L1 Cache	16 k, 32-way	32 k, 32-way	64 k, 2-way
L2 Cache	None	None	512 k, 4-way, unified
Memory (bus width, first block latency)	4-byte, 12 cycle	4-byte, 12 cycle	8-byte, 18 cycle

different characteristics than the SPEC2000 benchmarks when analyzing the static and dynamic characteristics of embedded processor performance. The dynamic instruction profile has more variation in the number of branch, memory, and integer ALU operations. It also has more variable text and data memory segment sizes, but the data tends to be more cachable. MiBench and SPEC2000 both have very predictable branches. The variation in the number of instructions per cycle also shows that the benchmarks

fall into the expected control and data intensive categories.

In the future, more benchmarks will be added to the MiBench benchmark suite. Future Automotive and Industrial benchmarks will include software pulse width modulation (PWM), virtual environment simulation and a real-time operating system scheduler. New Network benchmarks will include defragmenting TCP/IP packet streams and other packet manipulations.
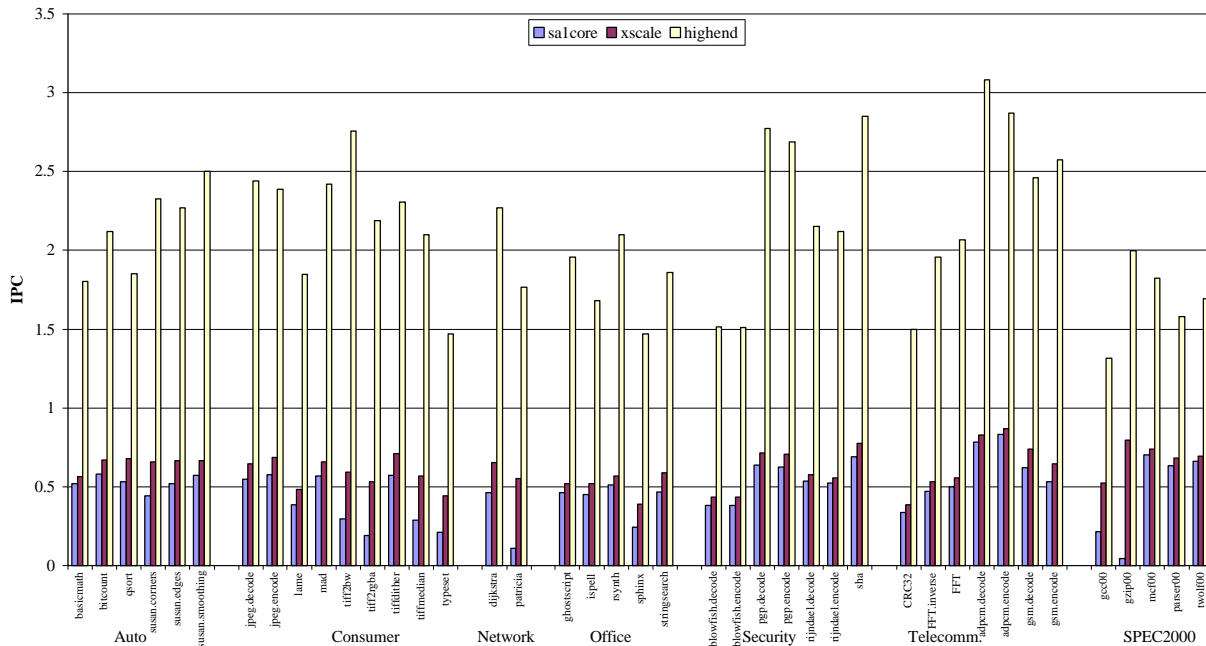
Figure 7: Instructions per Cycle (IPC).

6. References

[1]   J.L. Hennessy and D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, San Francisco, CA, 1996.

[2]   R. Weicker and S. Nixdorf. Dhrystone, CACM, vol. 27, num. 10, October 1984.

[3]   J.J. Dongarra, J.R. Bunch, C.B. Moler and G.W. Stewart. LINPACK Users Guide, SIAM Pub, Philadelphia, PA, 1979.

[4]   H.J. Curnow and B.A. Wichmann. A Synthetic Benchmark, The Computer Journal, vol. 19, num. 1, 1976.

[5]   Digital Review. CPU2, ftp://swedishchef.lerc.nasa.gov/drlabs/cpu.

[6]   C. Lee, M. Potkonjak and H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, Micro-30, November 1997.

[7]   B. Case. SPEC2000 Retires SPEC92, The Microprocessor Report, vol. 9, 1995.

[8]   J. Turley, Embedded Processors by the Numbers, Embedded Systems Programming, http://www.embed-ded.com/1999/9905/9905turley.htm, May 1999.

[9]   K.L. Kraver, M.R. Guthaus, T.D. Strong, P.L. Bird, G.S. Cha, W. Hold, R.B. Brown. "A mixed-signal sensor interface microinstrument," Sensors and Actuators A, vol. 91, pp. 266-277, 2001.

[10] EDN Embedded Microprocessor Benchmark Consortium, http://www.eembc.org.

[11] T. Mudge, Power: A First Class Design Constraint for Future Architectures, IEEE Computer, April 2001, to appear.

[12] Advanced Encryption Standard, http://www.nist.gov/aes.

[13] Counterpane Internet Security, Inc. The Blowfish Encryption Algorithm, http://www.counterpane.com/blowfish.html, 1993.

[14] National Institute of Standards and Technology. Secure Hash Standard, http://www.itl.nist.gov/fipspubs/fip180-1.htm, April 1995.

[15] P.R. Zimmermann, The Official PGP User's Guide. MIT Press, 1995.

[16] Silicon Graphics. Tiff Utilities, ftp://ftp.sgi.com/graphics/tiff, May 1999.

[17] Aladdin Software. Aladdin Ghostscript, http://www.cs.wisc.edu/~ghost/aladdin, April 2000.

[18] International Telecommunication Union. Global Standard for Mobile Communication, http://www.itu.int, February 2000.

[19] International Telecommunication Union. Recommendation G.726 (12/90) - 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM), http://www.itu.int, December 1990.

[20] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[21] R. Sugumar and S. Abraham. cheetah - Single-pass simulator for direct-mapped, set-associative and fully associative caches, Unix Manual Page, 1993.

[22] Intel Corporation, "SA-110 Microprocessor Technical Reference Manual," ftp://download.intel.com/design/strong/applnots/27819401.pdf.

[23] Intel Corporation, "The Intel XScale Microarchitecture Technical Summary," ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf.

[24] Intel Corporation, "Intel StrongARM SA-110 Microprocessors Instruction Timing," ftp://download.intel.com/design/strong/applnots/27819401.pdf.

[25] Rebel.com, NetWinder Family, http://www.rebel.com/netwinder.