Appears in the Proceedings of the 34^{th} Annual Workshop on Workload Characterization

A Characterization of Speech Recognition on Modern Computer Systems

Kartik Agaram Stephen W. Keckler Doug Burger
Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Abstract

In this paper we describe and characterize the speech recognition process, and assess the suitability of current microprocessors and memory systems for running speech recognition applications. We use representative benchmark applications — RASTA [7] to characterize the signal-processing on the front end, and SPHINX [13] for the graph search on the back end. Recognition time is dominated by the back end, which substantially exercises the memory system and exhibits low levels of instruction-level parallelism (ILP). As a result, SPHINX yields an average instructions per cycle (IPC) of 0.64 on a simulated 4-issue out-of-order microprocessor. We identify intelligent layout and thread-level parallelization as the primary methods to improve throughput, showing upper bounds on the performance improvements that these methods can achieve.

1. Introduction

In recent years, speech recognition technology has matured from an area of pure academic research to one with growing use in the marketplace. A variety of software packages for speech recognition are available in the mass market today, such as Dragon Systems' Dragon Naturally Speaking, IBM's ViaVoice, Lernout & Hauspie's Voice Xpress, and Philips FreeSpeech98. Vocabularies in commercial systems today range from 20,000 to 150,000 words. Recognition accuracies have been steadily improving as well, though current systems are still not sufficiently accurate to easily take dictation. Coupled with improvements in processor speeds and trends of ubiquitous computing, these developments promise to make speech a primary human/machine interface in the near future. However, real-time speech recognition requires substantial resources. Future speech applications such as real-time translation will demand even greater computational power. The growing importance of this application suggests a detailed study of its characteristics. In this paper, we describe and characterize the process of speech recognition, and suggest some methods for accelerating speech recognition on general-purpose platforms.

Speech recognition can be broadly broken down into a signal-processing kernel on the front end and a back end that performs a graph search on a large state space that is quadratic in the size of the vocabulary. The front end takes an audio signal as input and preprocesses it into a stream of feature vectors. The back end then uses this stream to perform the graph search. In this paper we characterize both the front end and the back end. To characterize the front end, we study RASTA [7], a Mediabench [12] benchmark. The benchmark we use for the back end is SPHINX [13], a system for large vocabulary continuous speech recognition. On a vocabulary of over 21,000 words, SPHINX achieves speaker-independent word recognition accuracies of 71-96%, depending on the complexity of the grammatical structure in the sentences. Both benchmarks are typical in terms of the algorithms used in modern recognition systems; SPHINX uses HMM-based algorithms that are currently prevalent [16], and RASTA processing is also widely used [8].

In general, current desktop machines have sufficient resources to perform dedicated large vocabulary speech recognition in real-time. However, this performance is attained at the expense of substantial memory capacity and bandwidth requirements that are not presently available in mobile devices. In addition, throughput-oriented server applications such as travel reservation systems provide rewards for further optimizing speech recognition, so as to be able to simultaneously support multiple recognition channels. Thus, speech recognition currently presents to system designers the twin challenges of attaining real-time performance in the context of a resource-constrained mobile device, and of maximizing performance and resource utilization in the context of a throughput-oriented server application.

Our results show that current architectures are poorly tuned to run either RASTA or SPHINX. Since RASTA takes up only 6.7% of the total recognition time and has a negligible memory footprint, the graph search performed by SPHINX presents the real obstacle to accelerating speech recognition. SPHINX has a large working set with highly irregular control and data access patterns.

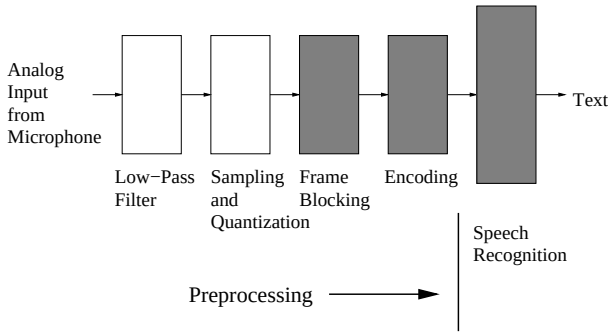

Figure 1: The various stages in converting speech to text. Shaded stages are studied in this paper. The first two shaded regions make up the front end, while the final stage constitutes the back end.

Cache performance is poor, and improves only slowly as cache sizes increase. Levels of available ILP are low. As a result of the poor locality and ILP, we measured an IPC of 0.64 on a simulated 4-issue microprocessor running the Alpha ISA, with 4 KB IL1, 64 KB DL1 and 512 KB L2. To run in real time, the minimum instruction throughput required in the Alpha ISA is 100 million instructions per second. The bandwidth required is 156 MB/s, both between the processor and DL1 and to main memory with an L2 capacity of 512 KB. Bandwidth required from main memory reduces to 16 MB/s with an L2 capacity of 8 MB. To improve speech recognition performance, we show that the memory access patterns of SPHINX have significant regularity that could be exploited by prefetching strategies, resulting in speedups of over 2.5, while the absence of ILP can be compensated for by exposing thread-level parallelism in the graph search, resulting in speedups of over 5.5.

The rest of this paper is organized as follows. Section 2 describes the process of speech recognition from end to end, including the algorithms used in preprocessing as well as the graph search. Experimental methodology and infrastructure is presented in Section 3. Sections 4 and 5 characterize the behavior of RASTA and SPHINX respectively, quantifying the minimal requirements for real-time performance in each. Section 6 suggests strategies for enhancing speech recognition performance. Section 7 summarizes related work.

2. The Process of Speech Recognition

Automatically converting speech into a textual representation requires several stages, as shown in Figure 1. First, a microphone converts the acoustic vibrations into an analog signal. This analog signal is then filtered to eliminate the high frequency components of the signal that lie outside the range of frequencies that the human ear can detect. The filtered signal is then digitized using a sampling and quantization phase. The digitized waveform is then partitioned into fixed-duration time-slices called *frames*, which are then compressed using one of several encoding schemes, to yield a stream of feature vectors. At this point, preprocessing is complete, and recognition techniques can be applied to this representation of the audio input. These typically involve a search to determine the optimal path through a graph, and constitute by far the most time-consuming and complex stage of the process. In the rest of this section we describe the stages in preprocessing and recognition in the context of RASTA and SPHINX, respectively.

2.1. Preprocessing and Feature Generation

The first two stages in Figure 1 are usually performed by a hardware A/D converter, and are common to the process of recording an acoustical signal to a digital format. The preprocessor takes as input an audio file sampled at 8 kHz, partitions the signal into frames, and encodes each frame into a sequence of feature vectors. We study RASTA as an example of such a preprocessor. RASTA performs perceptual linear predictive (PLP) [6] processing to analyze speech.

RASTA is structured as a loop that, in each iteration, operates on the vector of data corresponding to a sample, and generates a frame. Each frame is transformed from the time domain to the frequency domain by an FFT, operated on by a sequence of kernels, and transformed back to the time domain using an inverse Discrete Fourier Transform (DFT). Successive phases transform one vector into another. The entire process is loop-oriented, with fixed loop bounds. All data structures are arrays that are accessed sequentially. None of the inner loops have any loop-carried dependences.

2.2. Recognition

The back end accepts a stream of frames from the front end, and converts them to a textual representation. Much of the difficulty in performing speech recognition stems from the fact that a short sequence of frames may be locally interpreted in many different ways. Deciding between the various interpretations requires global analysis. The predominant technique for performing recognition is based on a stochastic framework called the *Hidden Markov Model* (HMM). An HMM is a graph of states with arcs weighted by transition probabilities between the states, and recognition is performed by determining the most probable path through the HMM graph corresponding to a given input sequence of frames. We study SPHINX as a representative example of such a speech recognition back end.

SPHINX performs recognition with the help of a dictionary broken down into four knowledge bases. The first is

a set of *phone models*. A phone is typically a single vowel or consonant sound. A typical 15-word sentence is composed of approximately 65 phones. A phone model for a specific phone lists the different sequences of frames that can represent that phone, along with probabilities for each sequence. The second knowledge base is an *acoustic model* that specifies for every pair of phones the probability of a transition between them, across all words in the dictionary. The third knowledge base is the *language model* that provides probabilities of transitions between words. Finally, a *pronunciation dictionary* maps phone sequences to word spellings. At runtime, SPHINX combines the four probabilistic knowledge bases, comprising over 90 MB, into a single large HMM graph 6.7 MB in size. The HMM is a forest of trees, with arcs between the roots and leaves of the various trees. Each tree represents all the words in the dictionary that begin with a specific phone.

Recognition is now reduced to finding the highest probability path through this graph corresponding to a given input sequence of frames, using a beam search. The beam search is structured as a loop that reads in one frame in each iteration. SPHINX maintains a list of currently active states in the HMM graph. As each frame is read, SPHINX propagates the arcs leaving the active states to generate the set of active states for the next iteration and tag them with probabilities. Low-probability candidates are eliminated in each iteration. Thus the beam search consists of alternating *evaluate* and *prune* phases, that access the HMM graph in a data-dependent manner. Both these phases provide ample opportunity for parallelization as individual states can be processed independently. At the end of each sentence, the beam search yields a set of candidate "last frames". An *answer-builder* routine then selects the candidate with the highest probability, and retraces its path to recreate the constructed sentence. This process is inherently sequential.

2.3. Combining Preprocessing with Recognition

A speech recognition system needs to have both the preprocessing and recognition phases. Between the two, recognition is far more time-consuming; RASTA takes only 6.7% of the total recognition time on the simulated microprocessor. In addition, since the initialization phase of SPHINX occurs only once and takes constant time for a given vocabulary, in steady state its contribution to the processing time for a single sentence is negligible, and the recognition process is almost entirely characterized by the beam search phase.

3. Experimental Infrastructure

This section describes the methodology used to study speech recognition. We select RASTA and SPHINX for this study because both use algorithms that form the basis for current research in speech recognition. In addition, their source code is accessible to us, allowing us to correlate their characteristics with features of the algorithms.

Feature	Baseline/Range
Out-of-order Processor	
Issue/decode width	4
Int ALUs	4
FP ALUs	1
FP multipliers	1
Branch predictor	Tournament, 1 KB x 1 KB local, 4 KB global, 4 KB choice
Memory Hierarchy	
DL1	64 KB $\rightarrow$ 64 MB (Baseline – 64 KB)
IL1	4 KB (Larger sizes are unnecessary.)
L1 cache latency	3 cycles (1 cycle for sizes less than 32 KB)
Unified L2	256 KB $\rightarrow$ 8 MB (Baseline – 512 KB)
L2 latency	12 cycles
TLBs	128 entries
Latency to DRAM	62 cycles
DRAM	128 MB

Table 1: Machine configurations used in sim-alpha simulations. Wider pipeline widths and more functional units were not found to improve performance.

RASTA takes an audio file as input and generates a feature vector stream. SPHINX takes a feature vector stream as input and generates text. RASTA generates the feature vectors in a format that is not compatible with the input of SPHINX, though the two formats are comparable. Further, the input to both corresponds to the same text. We run SPHINX for 12 sentences in all experiments except when we vary the L2 cache sizes, when we perform functional and cache simulation for 4 sentences to warm up the caches and then perform timing simulation for 3 sentences. When running RASTA, we perform RASTA filtering with a simple logarithmic bandpass filter. In particular, the JaH-RASTA filtering method, specified in the Mediabench benchmark suite and constituting an extra phase in the front end, was not performed.

For the majority of our simulations we use the sim-alpha simulator [4], an extension of the SimpleScalar tool set [3]. This simulator has been validated against an Alpha 21264-based DS-10L workstation, including the memory hierarchy. We use this simulator to study fine-grained microarchitectural details within the processor pipeline. Table 1 describes the machine configurations used in our experiments. The sim-alpha simulator does not model the operating system; all system calls take zero cycles. In order to model coarse-grained system-level effects we use SimOS-PPC [18], a port of SimOS [9, 17] for the PowerPC platform developed at IBM's Austin Research Laboratory.

SimOS-PPC performs full-system simulation, including the application, OS and all peripheral hardware. However, microarchitectural details are not modeled – SimOS-PPC performs only functional simulation of the processor. We use SimOS-PPC to measure the overhead due to the OS, and to examine the dynamic behavior of the various levels of the memory hierarchy and correlate it with the source code. Since we do not use SimOS-PPC to study details of the microarchitecture, differences in the ISAs do not substantially influence the results. Our results regarding OS overhead are specific to AIX, but are unlikely to be substantially different for other variants of Unix. SimOS-PPC uses the same memory parameters as sim-alpha, as given in Table 1.

4. The Front-end: RASTA

RASTA's characteristics are summarized in Table 2. RASTA is an extremely regular, loop-oriented program with a small working set. Processing 12 sentences on the baseline configuration in sim-alpha results in an IPC of 0.59. This IPC is sufficient to process speech in 3% of real-time, and 6.7% of the total speech recognition time when considering RASTA and SPHINX together. RASTA is compute-intensive and performs both integer and floating-point addition, multiplication and division. All data structures are easily accomodated in our simulated memory hierarchy. Each input vector has 160 floating-point values (640 bytes), corresponding to 20 ms of speech sampled at 8 kHz, with a floating-point value for each sample. The intermediate vectors are not large. The FFT of each frame is a vector of 128 elements. Internal vectors beyond this step contain 17 elements (68 bytes). The size of these vectors is again dependent on the sampling frequency. The elements of this vector are then transformed in parallel through the stages until the linear predictor equation, which is solved to yield 12 coefficients. During initialization 2 KB of memory is allocated on the heap for various data structures, after which 516 bytes are allocated and freed each iteration during the FFT. The source code reveals that this periodic allocation is not essential, permitting the number of frames to be changed at runtime. Thus, RASTA has a small working set and memory footprint. The bulk of the recognition time is spent in the graph search phase, which we study next.

5. The Back-end: SPHINX

On modern desktop systems SPHINX processes sentences at a rate greater than real-time; native execution on an Alpha DS-10L workstation processes sentences in 35% of the typical time taken to say them. SPHINX stresses memory system substantially, the baseline system yielding an IPC of 0.64. Table 2 summarizes the characteristics of

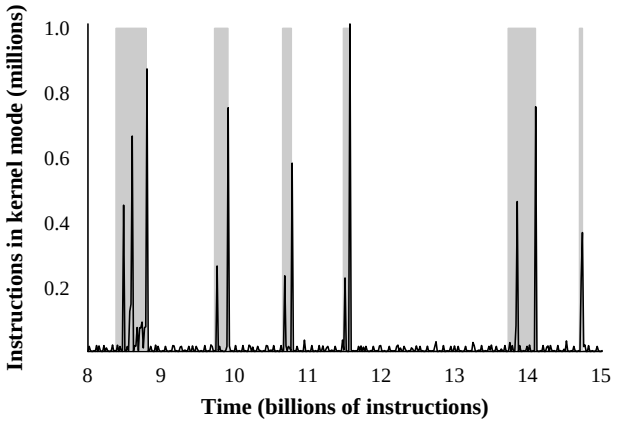

Figure 2: Kernel-mode instruction counts. Shaded regions denote execution of the answer-builder function. The end of this function coincides with sentence boundaries.

SPHINX when processing 12 sentences, and compares the behavior of the baseline machine configuration with that of several SPEC2000 benchmarks run for the first 500 million instructions. In the course of execution, SPHINX has a maximum memory footprint of 110 MB, most of which is used only during initialization to create the HMM graph. On the Alpha ISA, SPHINX needs an instruction throughput of at least 100 million instructions per second for real-time recognition rates with currently achievable accuracies.

SPHINX performs both integer and floating-point addition, multiplication and division. However, floating-point operations constitute only 5% of the instructions executed in steady state. The stream of feature vector inputs to SPHINX are floating-point values, and floating-point computation is required only to convert these to integers. The HMM graph has no floating-point data, so the rest of the beam search consists of only integer operations. Multiplication and division instructions, both for integer and floating-point, constitute less than 0.6% of the instructions executed in steady state, suggesting that hardware multipliers and dividers would be under-utilized.

Branches constitute 14% of instructions executed in steady state; on average 1 in 7 instructions is a branch. On sim-alpha, we measured a branch misprediction rate of 10%. The high misprediction rate is due to data-dependent branches in both the evaluate and prune phases of the beam search. In the evaluate phase, these branches arise because computing the score of each HMM involves obtaining the best score among all the nodes in the HMM. In the prune phase these branches arise when comparing the scores to a global best score and deciding on the nodes to prune.

Overhead due to the operating system is negligible. Experiments on SimOS-PPC show that, with a memory capacity of 128 MB, kernel and user level instructions executed

	RASTA	SPHINX	gcc	gzip	vpr	mesa	art	equake	eon	twolf
Execution aggregates										
Instructions (10^9)	2.4	16.2	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Cycles (10^9)	4.0	25.4	0.7	0.4	0.6	0.7	1.0	0.7	0.7	0.7
IPC	0.6	0.6	0.7	1.3	0.8	0.8	0.5	0.8	0.8	0.8
Instruction mixes (%)										
Loads	26.6	23.9	14.8	22.7	18.2	10.2	21.5	20.1	18.3	9.8
Stores	6.9	6.4	7.3	7.1	8.6	6.1	9.6	4.0	25.9	9.1
Branches	8.8	14.3	13.4	10.8	15.4	14.3	4.4	12.0	14.3	14.0
Integer ops	42.2	50.5	63.1	59.3	47.5	47.7	39.9	34.8	25.6	59.3
FP ops	10.4	4.8	0.1	0.0	10.4	21.5	24.6	29.1	15.8	7.8
Branch misprediction rates = predictor hits / predictor updates (%)										
	5.3	9.4	9.6	11.3	7.2	3.6	10.5	3.4	9.7	4.1
Cache and TLB miss rates (%)										
DL1	0.5	15.8	2.5	7.9	2.2	0.6	32.7	0.1	0.1	0.6
IL1	3.4	3.2	14.7	10.1	17.3	19.3	0.1	15.4	16.1	16.7
L2	1.9	41.9	4.4	32.9	3.1	0.8	71.7	0.5	0.1	1.4
DTLB	0.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 2: Summary of SPHINX's characteristics, and comparison with several SPEC2000 benchmarks

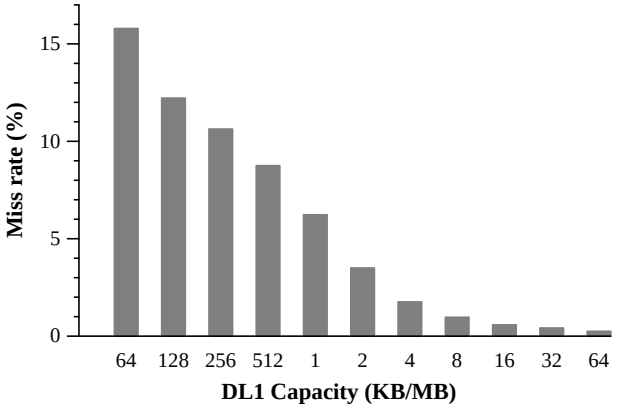

Figure 3: Miss-rate vs DL1 Capacity.

have an average ratio of 1:15. Figure 2 shows the time-dependent behavior of kernel-mode instructions while processing the last 7 of 12 sentences. Each point on the graph is a sample of 16.7 million cycles. The shaded regions denote the times of execution of the answer-builder function. The end of this function corresponds to a sentence boundary. The primary maxima in this graph correspond to the answer-builder function. From our analysis of disk activity, we determine that most of the OS activity during execution is due to paging. SPHINX alone is responsible for the strain on the memory system. On a larger configuration with 256 MB of main memory, paging is eliminated.

5.1. Memory System Behavior

SPHINX needs a larger cache capacity than any of the SPEC benchmarks. Memory operations constitute 32% of instructions executed in steady state; 25% of instructions executed are loads. Stores occur only when updating the dynamic HMM data structures during the beam search, when updating scores of the active states, and when pruning. The large working set of SPHINX causes substantial traffic to main memory. In steady state, SPHINX requires a minimum of 156 MB/s from the entire hierarchy to process sentences in real-time. On the memory hierarchy of the baseline machine configuration, the bandwidth required from the baseline 512 KB L2 to main memory is 161 MB/s. With an L2 capacity of 8 MB, the bandwidth required from L2 to main memory reduces to 16 MB/s.

Figure 3 plots the miss rates for a range of cache sizes. The baseline DL1 capacity is 64 KB. A 2 MB cache has a miss rate of 3.5%. This large working set is a result of the data structures used in SPHINX. During initialization SPHINX processes the four knowledge bases to form a more compact representation of the HMM graph. The main data structures used by SPHINX can be divided into two categories. The first category consists of data structures occupying 7.2 MB that are traversed in a regular pattern while processing each frame. However, fewer than 1% of these structures are accessed in any single frame. These data structures consist mainly of the input data structures to the HMM graph, and the back-pointer table that keeps track of the path taken by each of the active states. This path is retraced to recreate the sentence during the answer-builder function. Thus, even though these data structures are accessed regularly, the accesses are spaced sufficiently far apart for the data to be evicted from the cache in the interim. The second category constitutes 6.7 MB and con-

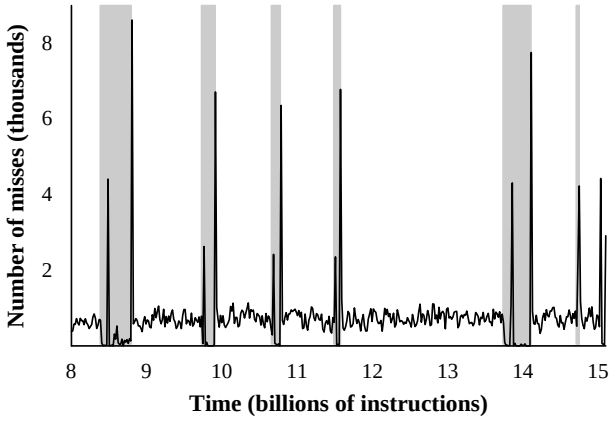

Figure 4: Dynamic IL1 misses.

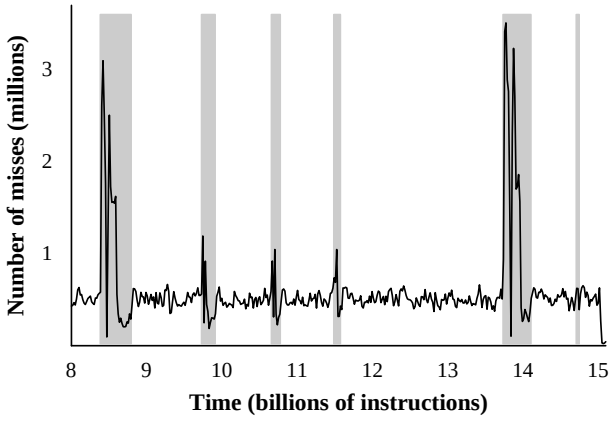

Figure 5: Dynamic DL1 misses.

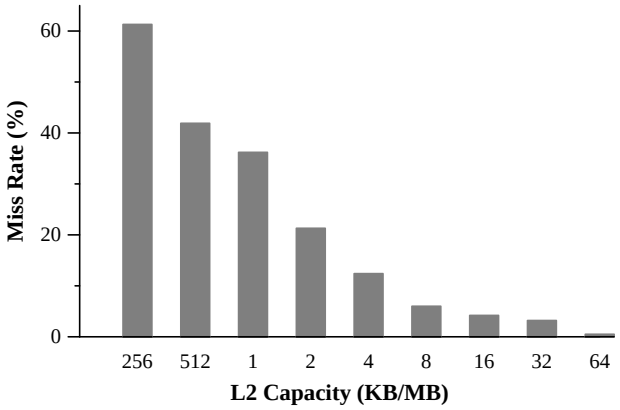

Figure 6: L2 Miss Rate vs Capacity, with a constant 64 KB DL1

L1 blk size → L2 blk size ↓	16	32	64	128	256	512
	DL1 miss rate					
1024	0.23	0.18	0.15	0.13	0.12	0.11
	L2 miss rate					
32	0.39	-	-	-	-	-
64	0.34	0.39	-	-	-	-
128	0.30	0.36	0.40	-	-	-
256	0.28	0.32	0.36	0.35	-	-
512	0.26	0.29	0.32	0.32	0.35	-
1024	0.25	0.27	0.29	0.29	0.30	0.32

Table 3: Sensitivity of cache miss rates with block size. DL1 and L2 capacities remain constant at 64 KB and 512 KB, respectively.

sists of the static and dynamic HMM graph data structures. The access pattern here is extremely irregular and data-dependent, as states of the graph are activated and deactivated. Of these accesses, 98% go to half of the 6.7 MB. We examine the distribution of access patterns of nodes within the HMM graph in more detail in Section 6, when we examine the opportunity to improve spatial locality in SPHINX.

SPHINX has a small code footprint; the baseline level-1 instruction cache capacity of 4KB shows a miss rate of 0.2%. Figure 4 shows the time dependent instruction cache performance of SPHINX, in a manner similar to the behavior of kernel-mode instructions. The maxima correspond to the exit of the answer-builder function. These are attributed to the initialization SPHINX performs at the start of each sentence, before diving into the innermost loop. The secondary spikes correspond to the start of this function. This figure shows that SPHINX goes through 2 phases in the course of processing a sentence: the beam search and the answer-builder. The instruction caches are turned over at each phase boundary. Figure 5 shows the dynamic performance of the data cache when running SPHINX. Once again, the maxima of the graph correspond to the answer-builder function phase boundary. The baseline cache size of 64 KB has a miss rate of 15.8%. Note that Figure 3 already shows the variation in miss rate with DL1 capacity. Figure 6 shows the variation in the L2 miss rate with capacity, with the DL1 at a constant 64 KB, when simulating SPHINX for 3 sentences after fast-forwarding through 4 sentences to prime the caches.

Finally, the sensitivity of miss rate to L1 and L2 block-sizes is summarized in Table 3. This table shows L2 block-sizes in the rows, and L1 block-sizes in the columns. The first half of this table shows the variation in DL1 miss rate with DL1 block size (DL1 miss rate is constant across all L2 configurations), while the second half shows the variation in L2 miss rate with DL1 and L2 block size. Increasing each of the DL1 and L2 block sizes causes the corresponding miss rate to decrease. This is due to the sequence in which the nodes of the dynamic HMM graph are allocated. In the next section we show that changing the layout of the HMM graph in memory can systematically exploit this spatial locality to improve cache miss rates and performance.

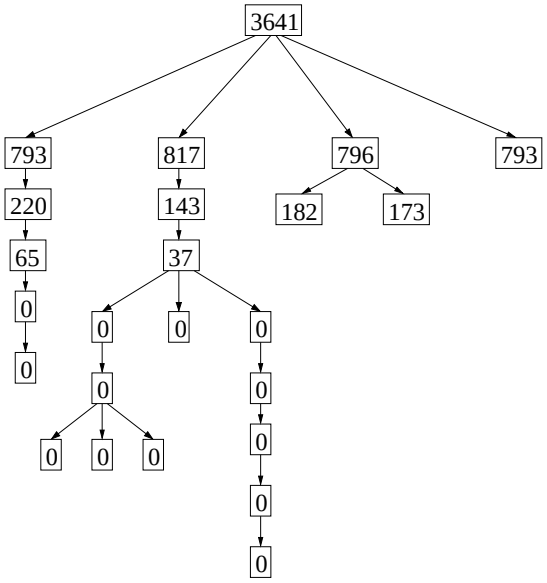

Figure 7: One of the trees in the HMM graph used in SPHINX. Each node is annotated with the number of times it was accessed while processing a sentence. The language model provides links from leaf nodes to roots of other trees.

6. Accelerating Speech Recognition

In this section we suggest strategies to enhance speech recognition performance. Since graph search dominates the total recognition time, we focus on SPHINX. The primary barriers to performance are the bandwidth in the memory hierarchy, and the absence of ILP. We address the bandwidth requirements by suggesting layout schemes to improve cache performance. Since the low ILP is an artifact of the sequential programming model used, we address it by exposing thread-level parallelism to the hardware.

6.1. Locality

When we eliminate all but compulsory cache misses by means of a large single-cycle cache, we find that IPC increases from 0.64 to 1.65, a speedup of over 2.5 compared to the baseline. This speedup presents an opportunity for improving performance by making the caches more effective. A study of the access patterns of the various nodes in the HMM graph shows that the distribution of accesses is anything but uniform. The HMM graph consists of a forest of trees. Each tree compactly represents all words that begin with a specific phone, and every descending path through a tree corresponds to a word, beginning with the phone at the root and ending at a leaf node. Transition probabilities between leaves and roots of the various trees complete the graph.

Figure 7 shows one of the 690 trees in SPHINX's HMM graph. Each node in this example has been annotated with the number of times it was accessed in the course of processing a single sentence. As can be seen, the number of accesses drops off steeply with increasing tree depth. This pattern holds true for all subtrees. Currently SPHINX uses a partial depth-first order in creating the graph — the words in the dictionary are inserted into the graph in alphabetical order. Thus each word on insertion forms a path from a root to a leaf. Nodes in the tree are laid out near their children rather than their siblings. However, the access patterns show that locality would be improved by allocating the nodes in memory in breadth-first order, thereby keeping siblings on common cache blocks. This strategy would also make hardware prefetching schemes more effective.

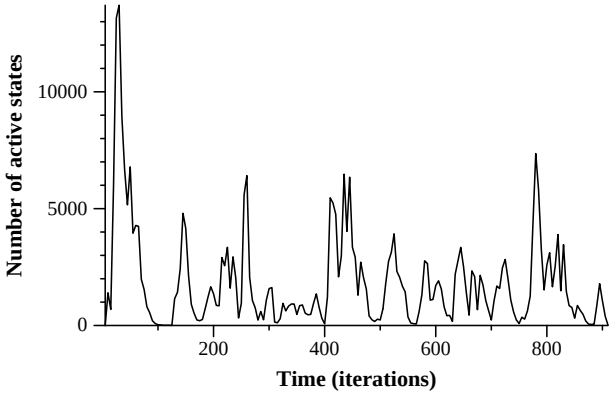

Figure 8: A profile of the number of active states in SPHINX.

6.2. Parallelization

To increase the IPC of SPHINX beyond sequential bounds we need to compensate for the absence of ILP by exposing more thread-level parallelism to the hardware. As described in Section 2, the graph search in SPHINX consists of alternating evaluation and prune phases. Both operate on the list of currently active states, and each of these states can be processed independently and in parallel. However, several dependencies need to be enforced by barrier synchronization. First, the next frame cannot be operated upon until the current frame has been entirely processed, as the active states for the next frame are only known at the end of the current one. Second, the evaluate and prune phases need to be executed in sequence. Third, several algorithmic dependencies exist within the prune phase. Finally, the answer-builder at the end of each sentence is entirely sequential.

Figure 8 shows the variation in the number of currently active states as SPHINX processes a single sentence. Since these states can be processed independently, they may be considered to be candidate threads, and are indicative of the potential thread-level parallelism in SPHINX. The number

of active states varies over a wide range, decreasing almost to zero several times before increasing again, indicating that the speech recognition process is relatively more certain of the correct path in those place. The number of active states increases or decreases gradually, making it possible for an adaptive platform to allocate and deallocate computational resources to speech recognition dynamically.

Most of the time in the beam search is spent within loops that iterate over the array of currently active states. Iterations of these loops can be executed in parallel. To determine a realistic upper-bound on the parallelism available in SPHINX, we instrument sim-alpha to time individual iterations in these loops, and overlap the timings for all iterations so that only the longest iteration contributes to the run time. Iterations of these loops take a mean of 204.5 cycles to execute. However, the distribution of iteration durations has a small number of outliers that take much longer to execute; the mode of the distribution is only 125 cycles. The serial parts, i.e. the parts of the beam search outside of loops, make up 21% of the total sequential execution time in steady state. Since the individual iterations are extremely lightweight, speedup is limited only by the serial parts. Thus, the upper bound on the speedup due to thread-level parallelism in this scheme is 4.71. In practice, by overlapping all iterations of each loop, while respecting the algorithmic dependencies between loops, we measured a mean speedup of 3.55, for a speedup of over 5.5 compared to the baseline. Overheads due to thread creation, synchronization and increased bandwidth requirements were not taken into account in this estimation.

7. Related Work

Several surveys of the state of the art in speech recognition may be found in the literature of recent years [2, 11, 14]. Speech recognition algorithms may be broadly broken up into three categories: statistical, knowledge-based and neural approaches. In recent years the statistical class of algorithms seems to give much better results than the other kinds [19]. The algorithms in this class have certain common features. They're all based on HMMs. HMMs are associated hierarchically with phones, words and word sequences. Recognition is reduced to the process of determining the probability that a given input corresponds to some word sequence. In theory this process could be repeated for all possible word sequences. The various algorithms proposed differ in the features they extract from the input and the way the language model is trained. Hybrid Neural/HMM approaches have been recently proposed [5], but these use neural networks to generate the transition probabilities for the network. The actual decoding process at recognition time remains the same.

Several authors have suggested methods to speed up speech recognition. Hon surveyed several approaches to implement speech recognition in hardware [10], including AT&T's Graph Search and ASPEN Tree Machines, SRI-Berkeley's Speech search machine, and CMU's PLUS architecture. These machines are all custom-designed parallel architectures specialized for graph or tree searching, albeit for older technologies and smaller vocabularies. Anantharaman and Bisiani [1] describe two custom pipelined architectures for the beam search in a speech recognition algorithm. Our study of the potential parallelization of SPHINX is orthogonal to these studies, in that we measure the available degree of parallelization in the beam search and possible speedups over sequential execution.

In the case of the parallelization of the beam search into threads, particularly relevant is a study conducted by Ravishankar [15]. This study implemented a multithreaded version of the beam search algorithm used in SPHINX, that statically partitions the data structures between a fixed number of threads, with no dynamic load-balancing. This implementation shows speedups of up to 3.85 for 5 threads. Each thread runs for the entire duration of the beam search, and the threads have 5 barriers per iteration. However, the algorithm of SPHINX it refers to differs substantially from the later release of SPHINX-II used in this study. They refer to an extra phase in the beam search (in addition to evaluate and prune), taking 50% of the runtime during sequential execution, that computes a set of acoustic scores for the entire HMM graph for each frame. The later version of SPHINX that we have used in this study moves this phase out of the graph search phase, doing the equivalent work during initialization and the answer-builder function.

8. Conclusion

The increasingly widespread use of speech recognition applications makes these important benchmarks to consider in designing future microprocessors and computer systems. In this paper we study the process of speech recognition from end to end by examining representative benchmarks for the front end (RASTA) and the back end (SPHINX).

We find that RASTA and SPHINX exhibit different behavior. RASTA is a compute-intensive, loop-oriented DSP kernel with a small memory footprint and regular accesses over small arrays. Only 6.7% of the recognition time is spent in the front end, with SPHINX accounting for the rest. SPHINX is a memory-bound data-dependent application with a graph search at its core. It has a miss rate of 2.5% over a 2 MB DL1 cache and a branch misprediction rate of 10%. The strain on the memory system and ILP result in an IPC of 0.64 on the baseline configuration. IPC improves to 1.0 when the L2 is increased from 512 KB to 4 MB.

To attain real-time speech recognition, SPHINX requires

an instruction throughput of 100 million instructions per second on the Alpha ISA. An 8 MB L2 cache causes a minimum of 16.7 MB/s of traffic to main memory. SPHINX runs in real-time on desktops and servers. Speech recognition performance may be improved by laying out the HMM graph in breadth-first order in memory to improve spatial locality, and parallelizing the graph search. Running SPHINX with a perfect memory system yields an IPC of 1.65, while a scheme for partitioning SPHINX into relatively fine-grained threads, without taking thread-related overhead and increased bandwidth requirements into account, yields an IPC of 3.55.

References

[1] T.S. Anantharaman and B. Bisiani. A hardware accelerator for speech recognition algorithms. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 216–223, 1986.

[2] Herve Bourlard, Hynek Hermansky, and Nelson Morgan. Towards increasing speech recognition error rates. *Speech Communication*, 18:205–231, 1996.

[3] Doug Burger and Todd M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[4] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual Symposium on Computer Architecture*, pages 266–277, 2001.

[5] Jurgen Fritsch. Modular neural networks for speech recognition. Master's thesis, Carnegie Mellon University and University of Karlsruhe, August 1996.

[6] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *Journal of Acoustic Society of America*, 87:1738–1752, 1990.

[7] H. Hermansky, N. Morgan, A. Bayya, and P. Kohn. RASTA-PLP speech analysis. Technical Report TR-91-069, The International Computer Science Institute, Berkeley, California, 1991.

[8] Hynek Hermansky and Pratibha Jain. Down-sampling speech representation in ASR. In *Proceedings of Eurospeech99*, September 1999.

[9] Stephen A. Herrod. *Using Complete System Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.

[10] Hsiao-Wuen Hon. A survey of hardware architectures designed for speech recognition. Technical Report CMU-CS-91-169, Carnegie Mellon University, School of Computer Science, August 1991.

[11] X. Huang, A. Acero, F. Alleva, D. Beeferman, M. Hwang, and M. Mahajan. From CMU Sphinx-II to Microsoft Whisper. Microsoft Research TR-94-20, 1994.

[12] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the International Symposium on Microarchitecture*, pages 330–335, 1997.

[13] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38:35–44, 1990.

[14] Louis C.W. Pols. Flexible human speech recognition. In *1997 IEEE Workshop on Automatic Speech Recognition and Understanding Proceedings*, pages 273–283, 1997.

[15] 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.

[16] D. Robert and E. Woodland. A hidden markov-model-based trainable speech synthesizer. *Computer Speech and Language*, (13:223-241), 1999.

[17] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Steve Herrod. Using the SimOS machine simulator to understand complex systems. *ACM Transactions on Modelling and Computer Simulation*, January 1997.

[18] Rick Simpson, Pat Bohrer, Tom Keller, and A.M. Maynard. SimOS-PPC PowerPC full system simulation, presentation.

[19] Steve Young. Large vocabulary continuous speech recognition: A review. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding, Snowbird, Utah*, pages 3–28, December 1995.