Characterizing the SPHINX Speech Recognition System

Kartik K. Agaram Stephen W. Keckler Doug Burger

Computer Architecture and Technology Laboratory
Department of Computer Sciences
Tech Report TR2000-33
The University of Texas at Austin
cart@cs.utexas.edu — www.cs.utexas.edu/users/cart

ABSTRACT

This paper examines SPHINX, a system for speaker independent, large vocabulary, continuous speech recognition. We find that SPHINX in particular, and speech recognition systems in general, display behavior that is substantially different from the compute-bound benchmarks that have traditionally driven popular computer system design. SPHINX applies considerable load on the memory hierarchy, with a large primary working set and poor locality. In this paper we quantify these results, and correlate them with the source code, showing that they are a consequence of the algorithms used, rather than specific implementation details of the processor, or the way the application is coded. The unprecedented growth of speech recognition applications makes it imperative that system designers lend them due consideration when designing the next generation of computer systems.

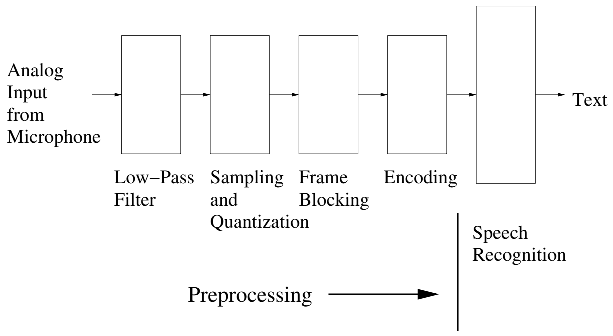
1 Introduction

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 [5], IBM's ViaVoice [8], Lernout & Hauspie's Voice Xpress [10], and Philips' FreeSpeech98 [12]. Recognition accuracies have been steadily improving as well, though current systems are still not accurate enough to take dictation. This, coupled with improvements in processor speeds and the looming reality of ubiquitous computing, promises to make speech a primary human/machine interface in the near future. In this context, there is a dearth of information on the performance of speech recognition applications on the common computing platforms of today. Special-purpose hardware architectures have been proposed for speech in the late 80's [2] [7], but these studies are outdated and inadequate in the context of trends of growing vocabularies and the use of general-purpose platforms in a multiprogramming environment.

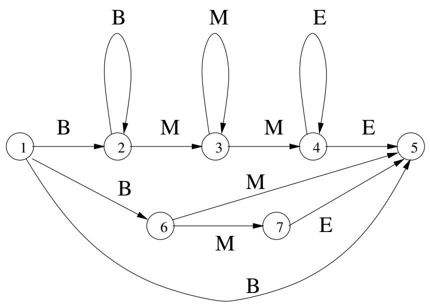
In this paper, we examine SPHINX [9], a system for speaker independent, large vocabulary, continuous speech recognition. On a vocabulary of over 21000 words, SPHINX achieves speaker-independent word recognition accuracies of 71-96%, depending on the complexity of the grammatical structure in the sentences. We study SPHINX on multiple platforms and using two simulators. Our results show that SPHINX in particular, and speech recognition applications in general, are fundamentally different from the benchmarks that have traditionally driven popular computer system design. SPHINX has a large working set with poor temporal locality and mediocre branch prediction. Cache performance is poor, and improves only slowly as cache sizes increase. We correlate these results back to the source, and show that they are not an artifact of this particular implementation, but a fundamental feature of current speech recognition algorithms based on Discrete Hidden Markov Models (HMMs).

In general, current desktops have sufficient resources to do dedicated real-time speech recognition at the levels of accuracy produced by programs like SPHINX. However, the load they place on the system is prohibitive in current multiuser/multiprogramming environments. For the immediate future, hand-held devices will be unable to hold sufficient memory to run speech recognition applications with any reasonably sized vocabulary, unless the memory hierarchy or recognition algorithms change drastically. Furthermore, if the accuracy of future speech recognitions systems are to increase, so too will the memory system and computational demands on the system.

The rest of this paper is organized as follows. First, we examine the performance of SPHINX on four contemporary platforms, and evaluate the extent to which they deliver on the promise of real-time speech recognition. Second, we use two simulators—SimpleScalar and SimOS-PPC—in concert to examine the dynamic characteristics of the program, and explain the dynamic features by correlating them with the source code. By doing so, we hope to provide a more general picture of the performance of speech recognition software on today's systems. Third, we compare the execution characteristics of SPHINX with several of the SPECINT2000 benchmarks, and show that it exercises the system considerably differently from them. Finally, we assess the impact of the instruction set architecture on performance by compiling SPHINX for 3 different ISAs—SimpleScalar's PISA (a MIPS-like instruction set), the Alpha, and the PowerPC—and running them on top of simulators with the same microarchitecture. With these experiments we hope to gain insight not only into the way speech recognition applications behave on different modern platforms, but also into the advantages and disadvantages of two of the simulators available to the research community in computer architecture today.



(a) The various stages in converting speech to text. This paper focuses on the final stage, that of software speech recognition.



(b) The phone HMM used in SPHINX. Arc labels denote probabilities.

Figure 1: An overview of speech recognition.

2 Speech Recognition Algorithms

Converting the spoken word into a textual representation automatically requires several stages, as shown in Figure 1 (a). First, a microphone converts the acoustical vibrations into an analog signal. This analog signal is then filtered to eliminate the high frequency components of the signal which lie outside the range of frequencies that the human ear can hear. The filtered signal is then digitized using a sampling and quantization phase. The digitized waveform is then blocked into frames, which are then compressed using one of several encoding schemes. At this point, preprocessing is complete, and recognition techniques can be applied to this representation of the audio input. Details of the various stages of preprocessing applied to the incoming signal in preparation for SPHINX can be found in Lee, et. al. [9]. In this paper we focus on SPHINX, which implements the final recognition stage.

Speech recognition is known as the process of converting frames to phones, phones to words, and words to sentences. A phone may be considered roughly equivalent to a single vowel or consonant sound. A typical 15-word sentence is composed of approximately 65 phones. Multiple frames form a phone. Much of the difficulty in speech recognition stems from the fact that the same frame may occur in multiple phones, and a single phone may contain different sequences of frames depending on the phones adjacent to it. SPHINX recognizes a sentence by performing a beam search through an a priori state network for the sequence of phones that best matches the input frame sequence. The SPHINX libraries employ discrete HMMs. HMMs are currently the predominant approach to speech recognition. They are characterized by a set of *states*, and an a priori set of *transition probabilities* between them. In SPHINX, each phone is modeled by a HMM.

Figure 1 (b) shows the structure of a HMM for a single phone. The vertices in the graph are the frames which compose the phones and the edges are the transition probabilities that connect the frames within the phone. This HMM is used as a framework to maintain lists of *current* states, and their associated probabilities. The incoming stream of preprocessed frames flows through each of the phone networks from left to right. As a frame is read in, each of the current states is propagated along each of the arcs leading from it. The list of current states is repopulated with the states at the other end of each of these arcs, while computing the probabilities for each.

In addition to the HMM phone models, SPHINX has an *acoustic model*, with a priori probabilities for transitions between phones, and a *language model*, with a priori probabilities for transitions between words. A pronunciation dictionary maps phone sequences to word spellings. The phone HMMs, acoustic model, language model, and pronunciation dictionary constitute the major data structures of SPHINX, comprising over 90MB altogether. At runtime, these data structures combine during initialization to yield a single huge network of states and transition probabilities. Recognition is now reduced to finding the path through this network that best fits the input signal. Thus, the speech recognition algorithm consists of a one-time initialization phase to load up the global a priori data structures, followed by a recognition phase where the incoming stream of sentences is processed atomically and in sequence using a beam search.

The beam search iteratively builds a tree of candidate paths through the HMM network. Starting from a single SILENCE node, it reads each incoming frame and adds new states, tagged by probabilities, to the tree of possible candidate solutions. Every node in the tree corresponds to a state in the a priori network. Periodic pruning eliminates candidate subtrees that deviate significantly from the best running solution. At the end of the sentence, the beam search is left with a set of candidate "last frames". A so-called *answer builder* selects the candidate with the highest probability, and retraces its path back to the original SILENCE node to recreate the constructed sentence.

Since the initialization phase 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 vanishingly small, and the recognition algorithm is almost entirely characterized by the beam search phase. Since the beam search ranges over the entire HMM network for each sentence, it exhibits poor locality. Locality is likely to suffer further in future, as the algorithms evolve to improve accuracy. The best systems of today do not yet approach the capabilities of human listeners. As shown above, today's language models incorporate contextual information as transition probabilities between words. This context allows them to decide between candidates like "to", "too", or "two". In spite of such contextual information, error rates for humans are still an order of magnitude less than machine error rates on many speech tasks [11]. Researchers have found that error-rates increase by 200-1000% [9] when an existing system is retrained on a larger vocabulary. Likely refinements to this algorithm, in seeking to improve accuracy, include incorporating information about language grammar, as well as contextual information at still higher levels, across whole sentences and beyond. Along with increasing vocabularies, these trends will contribute to still larger data set sizes and more computationally intensive searching.

3 Methodology

We have selected several distinct methods in this study, involving both native and simulated execution, to analyze the behavior of SPHINX. Each method has unique advantages and disadvantages. In this section we describe and compare the various methods used.

3.1 Native Execution

Native execution is the final test of an application's performance. However, running programs on real hardware limits the ability to probe the behavior of the application in a non-invasive fashion.

We can only measure the characteristics that are externally visible. In running SPHINX natively, we measure the run times and memory footprint for different numbers of sentences. The sentences input to SPHINX are shown in Appendix A at the end of the paper. We perform these experiments on 4 modern platforms, with the following configurations:

1. IBM RS/6000 H70 Server with 2 340MHz RS64-II processors, 64KB IL1, 64KB DL1, 512KB L2, 2GB RAM. Compiler - gcc 2.95.1 -O3.
2. 733 MHz Intel Pentium III, 16KB IL1, 16KB DL1, 256KB L2, 256MB RAM. Compiler - gcc 2.95.1 -O3.
3. 440 MHz Sun Ultra SparcIIi, 16KB IL1, 16KB DL1, 2MB L2, 384MB RAM. Compiler - gcc 2.95.1 -O3.
4. Compaq DS-10L, 466 MHz Alpha 21264 processor with 64KB IL1, 64KB DL1, 2MB L2, 256MB RAM. Compiler - gcc 2.9-gnupro-99r1 -O3.

Additional data can be collected using hardware performance counters and profiling tools, such as the Digital Continuous Profiling Infrastructure (DCPI) [3]. We are currently experimenting with these tools on the Alpha platform and with similar tools on the PowerPC platform to correlate the native results with those obtained from simulation.

3.2 Simulation

Simulation is useful for studying program characteristics that are normally not visible during native execution, such as cache behavior, dynamic instruction mixes, the performance of various microarchitectural structures like the fetch queue, register update unit, the branch prediction unit, etc. However, since the underlying machine is emulated in software, execution speeds are slowed by several orders of magnitude. Also, no single simulator can provide the means for asking all the questions one can think of with regard to the program's behavior. For these reasons, we study SPHINX under two simulation environments, SimOS-PPC and SimpleScalar, that are complementary in many ways.

Full-System Simulation: SimOS [6] [13] is a simulation environment capable of modeling complete computer systems, including a full operating system and all application programs that run on top of it. We use SIMOS-PPC [14], a port of SimOS for the PowerPC developed at IBM's Austin Research Laboratory.

SimOS-PPC has a number of properties that make it a useful simulation environment. It simulates an IBM PowerPC server in sufficient detail to run an unmodified version of AIX on it, thus capturing operating system as well as application behavior. SimOS-PPC provides multiple simulators at different levels of detail, and simulators can be switched in the course of a single run, using checkpoints. Finally, a well-designed interface is provided for collecting data that allows various collectors of varying levels of sophistication to be attached very easily.

Offsetting these advantages are several limitations. SimOS-PPC does yet include any detailed microarchitectural simulation. Instructions are assumed to execute in order, without any notion of pipelined or out-of-order execution. The memory hierarchy is imperfectly modeled. While SimOS-PPC has very detailed models for disks, cache and RAM are not modeled in equal detail. For example, bandwidth constraints are not maintained. Every access to cache or memory completes in a constant number of instructions.

We use SimOS-PPC to study the dynamic behavior of the various levels of the memory hierarchy, and the patterns of switches between kernel and user mode. SimOS-PPC also allows us to determine the source-level function in execution at a given cycle and thus correlate the dynamic behavior of the program with the source code.

The base machine configuration we use in our simulations corresponds to an IBM RS/6000 H70 Server with a 340MHz RS64-II processor, 64KB IL1, 64KB DL1, 2MB L2, and 512MB of RAM.

Microarchitectural Simulation: The SimpleScalar tool set [4] is a suite of simulation tools that perform detailed microarchitectural simulation. SimpleScalar's strengths and weaknesses are complementary to those of SimOS-PPC. SimpleScalar performs detailed microarchitectural simulation, without modeling the operating system. In particular, all system calls take zero cycles. It is much slower than SimOS, as simulation is being performed in much more detail. The memory hierarchy is modeled with a reasonably high accuracy. First order bandwidth constraints are modeled. Finally, SimpleScalar is quite portable across both instruction sets and host platforms.

We use SimpleScalar to examine the aggregate microarchitectural and memory system behavior of SPHINX, and to compare it with several SPEC CPU2000 benchmarks. Finally, the portability of SimpleScalar allows us to study the impact of ISA on performance, atop a constant microarchitecture.

The baseline SimpleScalar machine configuration has distinct 64KB IL1 and DL1 caches (block size 64), 512KB L2 (block size 128), 16KB entry 2-level branch predictor, an out of order core, 4 wide issue/decode, 8 wide commit, 4 integer ALUs, 1 integer multiplier, 1 floating-point ALU, 1 floating-point multiplier, 16-entry ITLB, and a 128-entry DTLB. Memory access latencies are 3 cycles for L1, 12 cycles for L2, and 70 cycles for main memory. When varying cache configurations, L1 capacities of 32KB or less have latencies of 1 cycle.

4 The Behavior of SPHINX

This section summarizes our findings on the characteristics of SPHINX. We run SPHINX for 12 sentences in these experiments. Execution time ranges from 16-20 billion instructions, depending on the platform, with an IPC of around 0.7. IPC halves when the out-of-order core is replaced by an inorder one. Figure 2 compares the performance of SPHINX on each of our platforms, in terms of the rate at which phones are processed. The bar on the left for each platform shows the number of phones processed per second, while the bar on the right shows the number of phones processed per 100 million cycles, thus normalizing the four platforms by clock rate. While the raw performance of platforms is comparable, and well over the required speed for real-time recognition at the level of accuracy provided by SPHINX, the various platforms operate over a wide range of clock speeds to achieve it. Thus, the PowerPC platform seems to be most efficient in terms of instructions per clock (IPC), while the Intel Pentium III is the least efficient. Desktops already seem to have the power to perform speaker independent speech recognition *in a dedicated environment*, though accuracy is

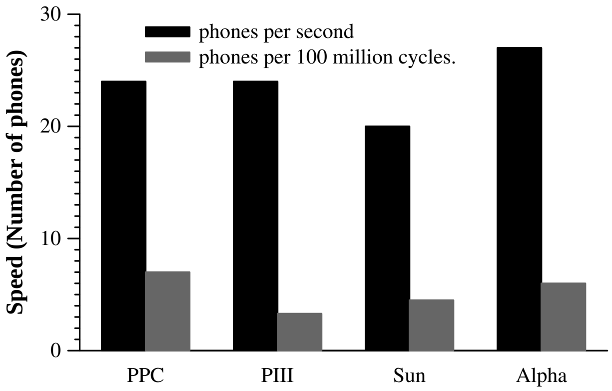


Figure 2: Relative processing speeds of SPHINX on various platforms. The bars on the left plot speed as number of phones per second, while the bars on the right for each platform plot the number of phones per 10^8 cycles.

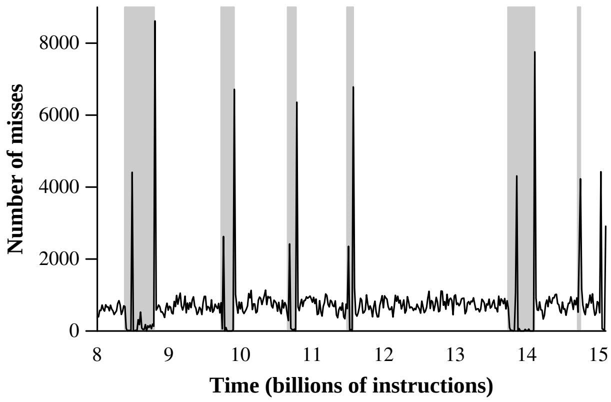
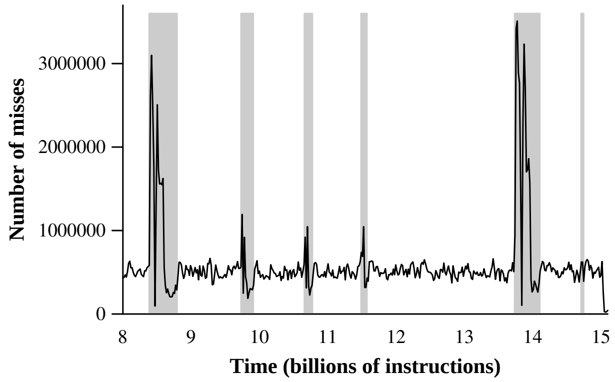


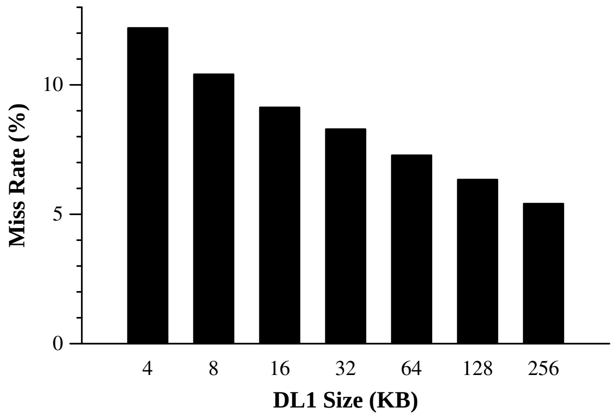
Figure 3: Dynamic IL1 misses. Shaded regions denote execution of the answer builder function. The end of this function coincides with sentence boundaries.

still problematic. The remainder of this section uses the simulators provide a more detailed view of the dynamic behavior of the application.

L1 Instruction Cache: SPHINX is quite well-behaved in the instruction cache. The baseline 64KB IL1 performs almost perfectly, with a miss-rate of essentially 0%. Figure 3 shows the time dependent instruction cache performance while processing 7 sentences, as generated by SimOS-PPC. SPHINX was primed into steady state by running five utterances before this data was collected. 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 primary spikes in the graph correspond to the exit of this function which is also the start of the next sentence. These are attributed to the initialization SPHINX performs before diving in to 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. We found that even reducing the IL1 cache capacity to 4KB only increases the miss-rate to 0.2%.



(a) Dynamic DL1 misses. Shaded regions denote execution of the answer builder function. The end of this function coincides with sentence boundaries.



(b) DL1 Miss Rate vs Capacity. We vary DL1 size with a constant 512KB L2.

Figure 4: Level 1 Data cache behavior.

DL1 Block Size	DL1 Miss Rate
16	0.1525
32	0.1028
64	0.0728

Table 1: DL1 Block Size vs Miss Rate.

L1 Data Cache: SPHINX exhibits poor data locality, including a miss-rate of 7% with a 64KB level-1 data cache. Figure 4 (a) displays the time-varying behavior of the baseline DL1 performance, overlaid with the execution of the recursive answer builder function at the end of each sentence-processing iteration, as generated by SimOS-PPC. This graph shows that the local maxima of DL1 misses correspond to the answer builder for each sentence. The primary spikes in the graph correspond to entry into this function, as a result of the backtracking required through the entire generated network to generate the final path. That is not to say that the DL1 is well-behaved the rest of the time. In between the answer-builders, mean miss-rates are at a substantial half million per 16.7 million instructions.

Figure 4 (b) shows the temporal locality in SPHINX, as measured by SimpleScalar. We fix the L2 cache capacity at 512KB, and plot the miss-rate of the DL1 while varying its size. Increasing DL1 size to 256KB causes miss-rate to drop only marginally to 5.5%. In a separate experiment, varying the size of the DL1 from 4KB to 16MB reduces miss rates proportionately, without dramatic improvements at any point. Over the same range, IPC variance is negligible, going from 0.506 to 0.513. Thus SPHINX exhibits poor temporal locality, in effect having a large single level working set that is never quite captured in the cache.

SPHINX does exhibit some spatial locality, but it is not able to improve performance. Table 1 shows the miss rate of the DL1 as measured by SimpleScalar, as we vary the DL1 block size from 16 to 64, while keeping the L1 capacity constant at 64KB, and L2 cache capacity constant at 512KB. Increasing the block size of the DL1 reduces miss rate. However, even a block size of 64 results in a miss rate of 7%.

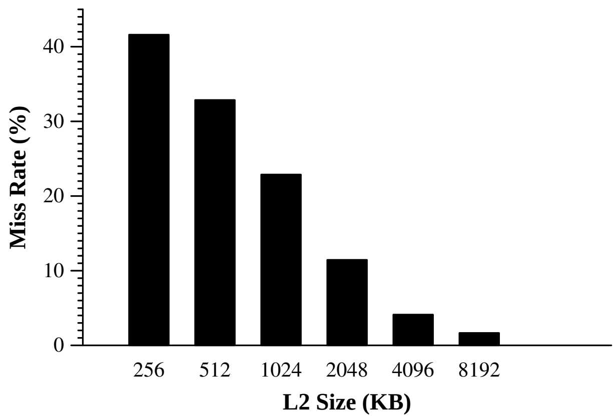


Figure 5: L2 Miss Rate vs Capacity

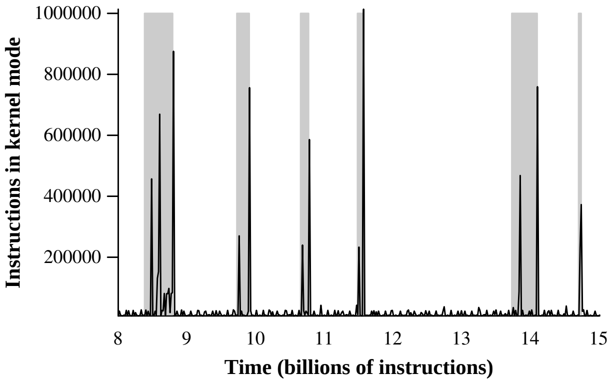


Figure 6: Kernel-mode instruction counts. Shaded regions denote execution of the answer builder function. The end of this function coincides with sentence boundaries.

L2 Cache: The theme of poor memory system performance carries over to the L2 as well. The baseline L2 cache capacity of 512KB shows a miss-rate of 33%. The dynamic trace generated by SimOS-PPC shows uniform behavior throughout execution. Figure 5 shows the miss rate in the L2, as measured using SimpleScalar, as we vary its capacity between 256KB and 8MB. It takes an L2 of 2MB to bring the miss rate to around 10%. Over the same range, IPC varies from 0.48 to 0.64.

Kernel Activity and Disk Usage: Overhead due to the operating system is negligible. SimOS-PPC tells us that kernel and user level instructions executed have an average ratio of 1:15. Figure 6 shows the time dependent behavior of kernel level instructions. Once again, activity peaks within 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. With a memory footprint of between 110 and 120MB on the various platforms, it takes 256MB of RAM to avoid swapping out to disk.

Summary: The preceding results serve to show that SPHINX heavily exercises memory, resulting in poor utilization of all levels of the memory hierarchy. The dynamic patterns of access and study of the source show that the lack of locality is not an artifact of this particular implementation; it is

	SPHINX	gcc	gzip	vpr	mesa	art	equake	ammp	bzip2
Execution aggregates									
Cycles (10^9)	31.0	55.0	15.0	20.0	82.0	27.0	23.0	65.0	13.0
IPC	0.51	0.29	1.05	0.78	0.19	0.58	0.7	0.25	1.25
Total memory allocated (MB)									
	109.0	24.0	186.0	37.5	11.0	4.3	41.6	22.0	184.0
Instruction mixes									
Loads	0.26	0.25	0.20	0.30	0.26	0.20	0.27	0.29	0.21
Stores	0.07	0.15	0.09	0.10	0.16	0.06	0.08	0.10	0.07
Branches	0.18	0.20	0.17	0.13	0.11	0.16	0.12	0.12	0.16
Branch misprediction rates									
	0.10	0.07	0.08	0.09	0.03	0.03	0.02	0.02	0.08
Cache miss rates									
DL1	0.07	0.02	0.02	0.03	0.00	0.43	0.03	0.07	0.02
L2	0.33	0.06	0.03	0.37	0.14	0.53	0.30	0.53	0.27
TLB miss rates									
ITLB	0.01	0.07	0.00	0.04	0.13	0.00	0.02	0.07	0.00
DTLB	0.33	0.12	0.10	0.10	0.07	0.50	0.10	0.14	0.10
RUU occupancy rates									
	42.70	8.70	49.60	45.70	2.55	61.50	28.80	26.20	44.80

Table 2: Comparison of SPHINX with selected SPEC CPU2000 benchmarks, with SPHINX executing 15.7 billion instructions and the SPEC benchmarks executing 16 billion instructions.

an inevitable consequence of the beam search algorithms themselves, that cause the entire HMM network to be traversed in a manner that is heavily influenced by the input.

5 Comparing SPHINX with the SPEC CPU2000 Benchmarks

In this section we compare the behavior of SPHINX with that of several SPEC 2000 benchmarks for the SimpleScalar instruction set architecture (PISA) and microarchitecture. SPHINX takes about 15.7 billion instructions to process 12 sentences. Correspondingly, all the SPEC benchmarks have been run for the first 16 billion instructions on the reference (ref) input set. Table 2 shows the highlights. From this table, we see the following results:

- None of the SPEC benchmarks has a footprint anywhere near as large as SPHINX. The number of pages allocated is much larger than any of the others, except for the compression benchmarks. In the case of gzip and bzip2, the large number of page allocations of about 180MB each is simply due to the size of the input being processed. Thus, while gzip and bzip2 use most of the data just once before moving on, SPHINX repeatedly accesses its 108MB.
- SPHINX has more branches than any of the SPEC benchmarks except for gcc. Further, branch prediction is significantly more inaccurate for SPHINX compared to any of the SPEC benchmarks. This is a result of the control-driven nature of the program, with the flow of

control dependent on the input data.

- L1 data cache and TLB performance is quite poor. The L1 data cache miss rate is poorer than that of all the benchmarks except for ammp. DTLB performance is worse than all the benchmarks except for art.
- As a consequence of the heavy strain on memory, RUU occupancy is up as well. The Register Update Unit or RUU [1] is a microarchitectural structure in SimpleScalar that automatically renames variables, keeps track of dependencies, and issues instructions when their dependencies are satisfied. In essence the RUU combines the instruction scheduling window and reorder buffer into a single microarchitectural structure. RUU occupancy is an indicator of the number of instructions in flight at any time. We deduce that the increased time spent by instructions in the RUU is due to dependencies on operands from memory, and therefore that the latency of the memory hierarchy is not being masked effectively.

In sum, SPHINX combines the largest memory footprint and the poorest branch prediction with among the poorest cache and TLB performances, compared to the SPEC benchmarks.

6 ISA and Compiler Impact on Performance

In this section we focus on the instruction set architecture and compiler, and examine their impact on performance. We run SPHINX on ports of SimpleScalar with 3 different front-ends, for the PISA, Alpha, and PowerPC ISAs. All ports have identical microarchitecture; the only differences are in the ISA and compiler used. The PISA binary was compiled by a port of gcc 2.6.2, the PPC binary by gcc 2.95.1, and the Alpha binary by gcc 2.9-gnupro-99r1. The optimization level was set to -O3 for all compilers.

Table 3 highlights the important differences in these runs. PISA is the most efficient instruction set with the fewest instructions. We surmise that this is because of the difference in the quality of compiler used. Both the Alpha and PowerPC binaries were compiled using later versions of production quality compilers, while the PISA compiler is a port of an earlier version. The newer compilers are perhaps able to better schedule the code to tolerate memory stalls. This hypothesis is supported by the lower occupancy rates in the RUU and the Load/Store Queue (LSQ).

Comparing Table 3 with Figure 2, we see that microarchitecture has a substantial impact on performance. With a common microarchitecture, the Alpha ISA has a higher IPC than that of the PowerPC. However, Figure 2 shows that the PowerPC platform has a more efficient microarchitecture, achieving better throughput per cycle. Of course microarchitecture complexity and clock rate are correlated, and of the machines we considered, the Alpha achieved the highest overall performance. As expected, cache and memory performance remain nearly identical across platforms.

7 Conclusion

Since simulation has become an integral component of computer system design methodology, it is critical to choose representative benchmarks. With speech recognition looming as the next killer application, its behavior must be understood if future computer systems are to sustain sufficient

	SS-PISA	SS-PPC	SS-Alpha
Execution aggregates			
insts (10^9)	15.7	19.8	19.1
cycles (10^9)	31.0	36.6	31.0
IPC	0.51	0.54	0.61
Instruction mixes			
Loads	0.26	0.24	0.24
Stores	0.07	0.08	0.06
Branches	0.18	0.14	0.14
Total memory allocated (MB)			
	109.0	110.0	117.0
Structure occupancy rates			
RUU	42.7	34.7	33.6
LSQ	16.8	12.9	11.9

Table 3: Comparing SPHINX on multiple ISAs

performance for its execution. Furthermore, simulation studies of benchmarks and architectures have their own limitations. Existing simulators can provide data on only a limited set of aspects of the benchmark, due to the conflicting demands of simulation speed and precision.

In this study, we address both the benchmark and simulator. We study SPHINX, a speech recognition application, and compare it with several SPEC benchmarks to determine how representative existing benchmarks are of a future application. Cognizant of the limitations of our tools, we use multiple simulators to perform this study, in an effort to broaden the scope of our investigations. SimpleScalar is useful as a detailed microarchitectural simulator yielding aggregate statistics for many microarchitectural features. SimOS-PPC is a full-system simulator that yields transient statistics that show us how various system-wide features evolve during the course of a program's execution. Using these two simulators in concert proved to be valuable in understanding both the dynamic and aggregate behavior of SPHINX.

This study shows that SPHINX is different from the SPEC benchmarks in the extent to which it exercises the memory system. It has a working set of 256MB and a mediocre IPC of 0.5 on a high-end desktop of today. While only a 4KB IL1 is required, a 64KB DL1 has a miss-rate of 7%, while a 512KB L2 has a 33% miss-rate. Branch prediction is also poor. In all these respects, SPHINX differs from the SPECINT2000 benchmarks. From our analysis of the algorithms used, we see why. The fundamental nature of the beam search algorithm causes the entire HMM network to be traversed in a manner that is heavily influenced by the input.

With poor data cache performance and large working set sizes, speech recognition algorithms present substantial challenges to existing architectures. The computational requirements are relatively small, thus requiring relatively simple processors to perform the search. The critical component is the memory system. Especially in the context of the inadequate performance of these applications today, efforts to improve transcription accuracy are likely to exacerbate these trends. Even though the word recognition accuracies of these programs exceed 90% in some cases, we observe that SPHINX is unable to recognize any but the simplest of sentences completely correctly.

Improving transcription accuracy is a challenge. We do not yet know how speech recognition algorithms will evolve to address this, or how the changes will affect performance. In their current form, speech recognition applications are more suited to offline transcription, rather than online transcription or interactive applications. We anticipate that achieving high fidelity speech recognition, even without the constraints on realtime execution, will require significant storage to hold the static dictionary data structures and the a priori HMM probabilities. The beam search may be accelerated with a more efficient organization of these critical data structures and hardware assisted prefetching or streaming the components required during the search. These memory system design tradeoffs must be examined and understood in order to provide sufficient accuracy and speed of speech recognition and other search-based applications in future low and high end computer systems.

8 Acknowledgments

We thank Tom Keller and the rest of the SimOS-PPC group at IBM's Austin Research Laboratories for making SimOS-PPC available to us for this study. Thanks also to the members of the CART group for hours of discussion and comments on several drafts of this paper. This work was funded by grants from IBM and the Intel Research Council.

References

[1] Santosh G. Abraham, Rabin A. Sugumar, B. R. Rau, and Rajiv Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 139–152, December 1993.

[2] 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.

[3] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? Technical Note 1997-016. Digital Equipment Corporation Systems Research Center, Palo Alto, Calif., July 1997.

[4] 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.

[5] Dragon Naturally Speaking, http://www.dragonsys.com.

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

[7] 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.

[8] IBM ViaVoice software, http://www-4.ibm.com/software/speech/.

[9] 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.

[10] L&H VoiceXpress, http://www.lhs.com/voicexpress/.

[11] Richard P. Lippman. Recognition by humans and machines: miles to go before we sleep. *Speech Communication*, 18:247–258, 1996.

[12] Philips FreeSpeech98, http://www.speech.be.philips.com/.

[13] 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.

[14] Rick Simpson, Pat Bohrer, Tom Keller, and A.M. Maynard. SimOS-PPC PowerPC full system simulation, presentation, http://www.cs.utexas.edu/users/cart/simOS/reports/NonConfidentialOct20-1999/-index.htm.

A Input data

The input data set for the experiments in this paper consists of the following sentences:

1. AS COMPETITION IN THE MUTUAL FUND BUSINESS GROWS INCREASINGLY INTENSE MORE PLAYERS IN THE INDUSTRY APPEAR WILLING TO SACRIFICE INTEGRITY IN THE NAME OF PERFORMANCE
2. FOR A TWO TRILLION DOLLAR BUSINESS BUILT ON PUBLIC CONFIDENCE THIS TREND IS DISHEARTENING AT BEST AND DOWNRIGHT DANGEROUS AT WORST
3. SO FAR THIS YEAR THE INDUSTRY'S SLIDE HAS BEEN APPARENT IN ALLEGATIONS OF INSIDER TRADING BY FUND MANAGERS UNSAVORY FUND SALES PRACTICES AT BANKS AND THE USE OF HIGH RISK DERIVATIVE SECURITIES IN NORMALLY CONSERVATIVE MONEY MARKET FUNDS
4. AND IN THE LATEST BLOW TO PUBLIC CONFIDENCE IN THE FUNDS SOMEONE AT INDUSTRY TITAN FIDELITY INVESTMENTS DECIDED LAST FRIDAY TO SEND NEWSPAPERS THURSDAY'S PRICES FOR MOST FIDELITY FUNDS BECAUSE THE FRIDAY DATA WASN'T READY
5. IN THE ENSUING PUBLIC HUMILIATION SUFFERED BY FIDELITY THIS WEEK AS IT WAS FORCED TO FESS UP TO ITS INTENTIONAL MISTAKE THE COMPANY HAS SOUGHT TO MAKE CLEAR THAT NOBODY BOUGHT OR SOLD A FUND DIRECT FROM FIDELITY AT THE WRONG PRICE
6. THOUGH NEWSPAPERS LAST SATURDAY UNKNOWINGLY LISTED THURSDAY PRICES FOR ABOUT ONE HUNDRED AND FIFTY OF FIDELITY'S TWO HUNDRED AND EIGHT FUNDS THE COMPANY SAYS TRANSACTIONS EFFECTED FRIDAY WERE MADE AT THE CORRECT PRICES WHICH FIDELITY RECEIVED AFTER IT MET NEWSPAPER DEADLINES WITH THE WRONG DATA EARLY FRIDAY EVENING
7. BUT THAT DOESN'T CHANGE THE FACT THAT MOST FIDELITY'S SHAREHOLDERS WERE LIED TO IN VARYING DEGREES IF THEY LOOKED AT THEIR FUND PRICES IN NEWSPAPERS LAST WEEKEND
8. THE UNIDENTIFIED FIDELITY EMPLOYEE WHO MADE THE DECISION TO GO WITH INCORRECT PRICES SHOULD SIMPLY HAVE LISTED THEM AS NOT AVAILABLE
9. THAT WOULD BE STANDARD FIDELITY PROCEDURE SAID JANE JAMIESON SENIOR VICE PRESIDENT AT THE BOSTON BASED COMPANY
10. FIDELITY'S MISTAKE AND MOST OF THE SELF ADMINISTERED BLACK EYES SUFFERED BY THE FUND INDUSTRY THIS YEAR HAVE ONE THING IN COMMON THEY COULD HAVE BEEN PREVENTED IF THE EMPLOYEES INVOLVED WERE PLAYING BY THE RULES EITHER THE FUND COMPANIES' INTERNAL RULES OR THE PRUDENT MAN RULE THAT IS SUPPOSED TO GOVERN THE INVESTMENT OF MONEY ENTRUSTED BY THE PUBLIC TO ALLEGEDLY PROFESSIONAL MONEY MANAGERS
11. THE PROBLEM IS THAT IT CAN BECOME VERY TEMPTING TO BEND OR BREAK THE RULES IN THE NAME OF STAYING COMPETITIVE
12. TAKE FIDELITY'S CASE AS AN EXAMPLE