Sphinx Parallelization

James Tuck, Lee Baugh, Jose Renau, and Josep Torrellas
University of Illinois at Urbana-Champaign

May 2002

ABSTRACT

Speech recognition applications challenge traditional out-of-order processors because of low cache locality and poor branch behavior. We observe that these symptoms may be mitigated my exploiting the existing parallelism in these algorithms. In this project, we exploit many levels of parallelism in Sphinx, a leading speech recognition system, to improve architectural utilization by decreasing cache miss rates and improving branch prediction. The resulting parallel implementation will be evaluated in several single and multiprocessor systems. Additionally, we plan to evaluate Sphinx in the novel M3T architecture.

The contribution of this project can be classified as threefold: parallelizing Sphinx, evaluating it in several single processor and multiprocessor systems, and analyzing M3T's effectiveness in executing Sphinx. All the evaluations are performed with a new simulation environment developed especially for chip-multiprocessor environments.

1 Introduction

Simple voice recognition is already widely used on desktop computers, cell phones, and a few hand held devices. However, the quality of this recognition falls well below human capability. Speech recognition systems capable of speaker independent, continuous speech, and large vocabulary recognition are not as widely used do the large computational and memory requirements. To assist in the implementation and deployment of better performing speech recognition algorithms, many research groups have been characterizing speech recognition on current uniprocessor systems.

Recent studies have proposed various methods to improve the behavior of Sphinx, a speech recognition application from CMU, on uniprocessor systems. These studies considered the use of larger caches both at the L1 and L2 levels. In both cases, large amounts of cache were required to attain a reasonable IPC [1]. It was also suggested that prefetching of key data structures may improve the cache behavior without drastically changing the architecture. This approach will likely yield minimal benefits due to the data dependent nature of the algorithm. Predicting what structures will be used far enough in advance would prove challenging. For these reasons, it is necessary to

look beyond current uniprocessor systems for performance improvement and consider systems capable of directly exploiting parallelism.

Uniprocessor systems have long been the typical target for speech recognition applications. However, speech recognition algorithms offer a significant amount of parallelism that a traditional uniprocessor system cannot exploit. Moreover, due the fine-grained nature of the parallelism in Sphinx, many multi-processor work stations are incapable of utilizing the available parallelism as well. Therefore, we wish to analyze the performance of speech recognition on platforms capable of exploiting the available parallelism in the hope that it may lend additional insight into the behavior of these algorithms. However, we also hope to explore how the benefits of parallelism may afford cache locality and improve branch prediction rates, which are reported to be low for speech recognition algorithms like Sphinx [1]. Finally, exploiting the parallelism in these algorithms will assist in explaining why uniprocessor implementations behave as they do.

We have chosen Sphinx as our speech recognition platform, and we will evaluate the parallelism in Sphinx on several platforms, including IBM SP-2 workstations, SGI workstations, and simulated chip multiprocessors. In particular, we will focus on the M3T chip multiprocessor architecture.

This report will focus on proposing methods for parallelization, characterizing the different computational components of Sphinx by profiling them on a uniprocessor system, and analyzing their sensitivity to input sets. Finally, we will evaluate the effectiveness of our parallel implementation on the workstations listed above, and our simulated results for M3T.

2 Sphinx

Sphinx was designed to address the need for a more powerful model of speech processing, one based in the mechanics of speech. Sphinx models speech as expert linguists would – by breaking words down into their phonetic elements and using signal processing techniques to evaluate which phone was spoken. Such a system may eventually provide speaker independent and arbitrary language model recognition.

We choose Sphinx as our system because it is a mature project in the speech recognition community. Sphinx is a long-term CMU project with the objective to stimulate the creation of speech-using tools and applications while advancing high performing speech recognition systems.

To evaluate Sphinx, we consider only one of several configurations available. We will be working only with a baseline configuration that provides continuous, near-real-time speech recognition from a prerecorded speech sample. In the following sections, we will discuss how the Sphinx speech recognition algorithm works.

2.0.1 Signal Processing Front end

The audio input for Sphinx may be read in continuously from a sound card, or from a prerecorded audio file. The initial audio input is filtered every 10ms, using a 20ms, anti-aliasing, Hamming Window. The signal processing algorithms used in this stage help identify units of speech smaller than the phonetic units of words. The result of the filtering is a *cepstrum vector* which represents the recorded utterance. These very small utterances can be used to step through a Hidden Markov Model (HMM) of the phoneme, as described in the next section.

2.0.2 Feature Vector Distance Calculation and Quantization

In this stage of the computation, the uttered phoneme is determined by examining several cepstrum vectors in sequence. The cepstrum vector calculated in the signal processing front end is divided into several characteristic vectors that have been shown to be good keys for the speech recognition process. These features include the cepstrum, differenced cepstrum, doubly differenced cepstrum, and power coefficients.

The four feature vectors are then quantized. The quantization process is known as *scoring* since the feature vectors are compared to a codebook of prototype vectors and the prototype with the highest probability of matching (largest score) is chosen to represent the utterance. The selected prototype vector is used when searching the phonetic HMM [5]. The feature vector quantization contributes a large percentage of execution time in smaller input sets.

2.0.3 Searching

The searching phase accepts as input the quantized vector from the scoring algorithm. Using this value, each active HMM channel in the language model graph is evaluated.

The searching phase relies on a large graph of HMMs to find the proper utterance. HMMs are used to model not only the phones, but also the words and sentences that make up the language model. Consequently, when transitioning between two states of the HMM, Sphinx may stay within the same word, transition between words, or transition between sentences. As an utterance is fed into Sphinx, every possible matching sequence is kept until the utterance has concluded, at which point the sequence with the highest score is accepted. With many possible phones, many phones per word and many words per sentence, it is possible that thousands of phone sequences must be analyzed to attain good accuracy.

With the added constraint of real time operation, it is necessary to look for algorithms to speed the traversal of very large language models. Hence, Sphinx uses a dynamic programming algorithm based on the observation that any phone sequence closely matching a prefix of the utterance may be a prefix to the entire phone sequence that will most closely match the entire utterance. As each frame is evaluated, sequence prefixes whose probabilities are lower than some threshold are disposed of, and only the best potential prefixes seen so far are retained [6]. Language-specific information, such as what phones are likely to occur immediately before and immediately after

any given phone, and which words are likely to follow any given word, is employed to further prune improbable phone sequences. At the end, the sequence with highest probability is reconstructed by backtracking through the most probable phone sequence [1, 6].

2.0.4 Sphinx Inputs

Sphinx takes as input a language model, dictionary, feature vector codebooks, and some form of audio input. Each of these inputs can affect the quality of the recognition and the time required to perform recognition. Later we will discuss the impact of varying some of these factors, therefore it is important to understand their influence on Sphinx's behavior.

In the early phases of computation, the feature vector codebooks are used to quantize the input data into a single value that will drive the HMM searching phase. If the value is not quantized properly, the resulting HMM search could be widely skewed. To minimize the error in quantization, it is common to use multiple codebooks. When running Sphinx, this is illustrated through use of the "-top 4" option which specifies to use the best 4 codebooks. Improving the quality of vector quantization has a direct impact on the computation requirements in this stage of Sphinx.

The dictionary specifies how each word in the language model is pronounced phonetically. The phones have a direct mapping to an HMM available in the system. It is possible for the dictionary to have multiple pronunciations for each word. The total size of the dictionary is a factor in the measure of complexity in the searching phase.

Finally, the language model specifies what sentences can be recognized in the sample audio. The sentences, fragments, and word phrases specified here will be converted into a large graph of HMMs. Each valid trace through this graph is a called a beam. The larger and more complex the language model, the larger the number of beams that must be searched to determine the spoken utterance.

2.1 Architectural Characterization

Sphinx is similar to SPECINT applications in that it relies primarily on integer computation and is programmed to run sequentially. However, a comparison of Sphinx to the SPEC CPU2000 benchmarks [1] shows that it requires a larger memory footprint(109 MB), has a higher L1D miss rate (10%), and worse branch misprediction rates (10%) than all but one of the benchmarks it was compared against. Those benchmarks that did perform worse in one regard, usually outperformed Sphinx in other dimensions.

Sphinx shows this behavior due to frequent searching through HMMs. These graphs can be very large, and depending on the input, the same node in the graph may never be accessed twice. As a result, Sphinx exhibits poor cache locality and a large working set. This results in frequent displacement from the cache, and little benefit from fetching larger blocks of data [1].

To improve the performance of Sphinx, it will be necessary to alleviate the poor memory system behavior of Sphinx either through rewriting the software, or through novel architectures that can exploit larger working sets in memory, without requiring infeasible memory bandwidth. Simulations we have already performed suggest that eliminating the major memory bottlenecks can more than double the IPC of Sphinx when run on an aggressive superscalar processor.

2.2 Parallelization

Characterization of the Sphinx application by Agaram et al [1] and confirmed in our own simulations, shows that Sphinx spends a small percentage of its time in the signal processing front end. The remaining time is spent in vector quantization and searching. Which phase consumes more of the execution time is dependent on the the language model and audio input. For this reason, we focus on techniques that will improve the entirety of the speech recognition algorithm. However, we realize that larger language models will have a greater impact in the future. For this reason, it is necessary to ensure scalable parallelism for the searching phase of the algorithm.

Input parameters for which searching is expensive show that a large amount of time is spent searching through the HMMs to find the best representation of the spoken utterance. The memory system is placed under greater stress when searching is a large component of Sphinx's runtime behavior. Therefore, the best techniques for parallelization will target minimizing search time in the HMMs, while simultaneously improving the memory system utilization.

To improve the overall execution time of Sphinx we propose a pipeline of the main stages of execution. Each stage can in turn be parallelized using threads to whatever extent is beneficial.

2.2.1 Pipelining

In the discussion of the Sphinx algorithm, it was explained that there are several primary stages of operation. When Sphinx is run on a uniprocessor, these stages time share the processor and the cache. By examination of the behavior of each of these stages, we found that the cache behaved well for all stages except the searching stage. Also, the branch prediction was similarly well behaved for all but searching. This suggests the separate stages are well behaved on their own and may benefit from a pipelined implementation.

Further examination of the implementation confirmed that the stages were relatively independent with very little data passing from one stage to the next. Because each stage behaves well independently, and can only interfere destructively with the other stages, we decided to break signal processing, feature normalization, vector quantization, and searching into unique stages of operation. The resulting pipeline may be visualized as a dataflow architecture as in Figure 1.

This implementation also has the benefit of allowing parallelization and analysis of each phase independent of the others.

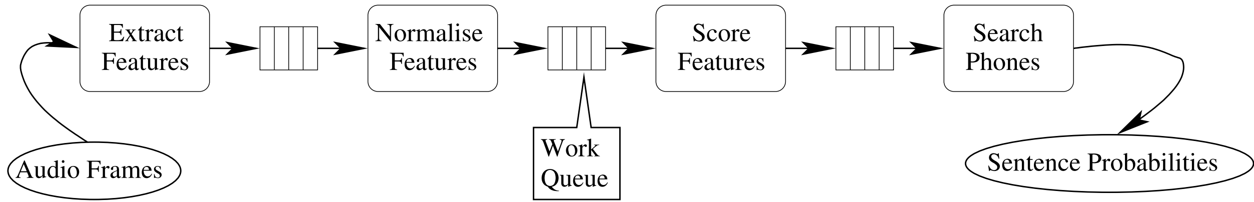


Figure 1: Pipeline Implementation

2.2.2 Multithreading

Multithreading may be applied to any stage that needs a speedup in order to balance the pipeline. In the case of the distance calculation for each feature vector, we can easily parallelize up to four threads by allocating a feature vector per processor. Due to imbalance in the computation of the vectors, the efficiency of the parallelization is diminished. However, this can be used if the distance calculation were a bottleneck in the pipeline implementation.

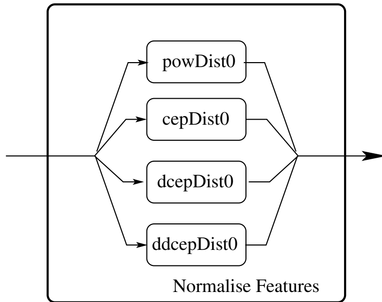


Figure 2: Parallelization of the distance computation.

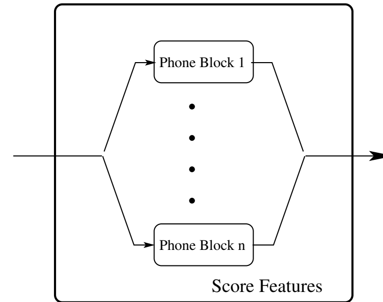


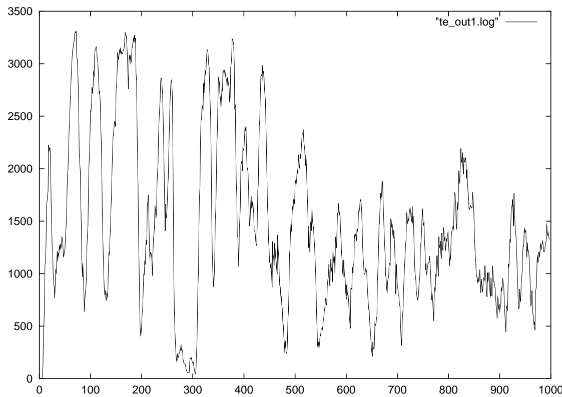
Figure 3: Parallelization of quantization.

The scoring, or vector quantization, phase is an important stage of parallelization for smaller language models, as will be shown later in the paper. The majority of the time in this stage is spent looping over a large number of codewords and calculating their probability. As a result, it is straightforward to partition the work over an arbitrary number of processors. Because the codewords live in contiguous memory and they are never modified during execution, we can expect good cache behavior regardless of the number of processors used. If enough processors are used, the working set may be kept in the cache, thereby mitigating some of the conflict misses experienced in the uniprocessor implementation.

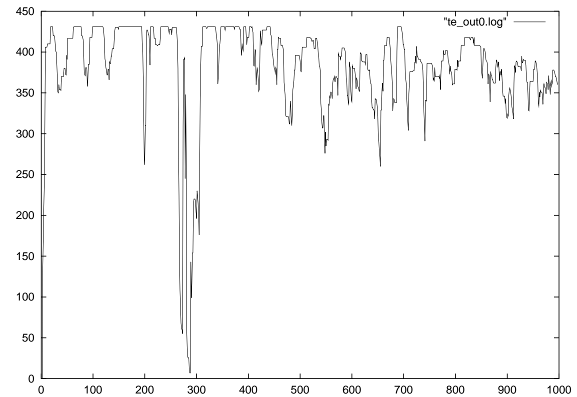
In general, the searching stage is the most important for parallelization, and there have been many proposals for parallelization of this stage. The overwhelming interest is due to the inherent parallelism of searching the HMM graph for the best trace.

In the course of execution, the search continuously evaluates thousands of channels. The evaluation and pruning of each channel is independent of the other active channels in a given frame. Therefore, there is a desire to parallelize

the search as much as possible. Below is a figure showing the number of active channels each frame of execution for one of our test cases.



(a)



(b)

Figure 4: (a) The number of active non-root channels (b) number of active root channels in a given frame of execution.

Figure 4 is a clear indicator that parallelism may be fruitful for the searching stage. We expect that the greatest performance benefit may lie in partitioning the set of phone models (HMMs) into groups, distributing these groups among threads, and letting each thread be responsible for computations on those models [2, 6]. Although reasonable results have been reported for this technique [6], several issues cause us to expect speedup less than linear to the number of processors [7]:

- Much of the latency is due to the search algorithm's considerable memory requirements. Though many read-only data structures can be made local to every processor, memory accesses will still be frequent and widespread [1].
- Since any initial partitioning will dictate which future paths can be explored on each thread, it will not be practical to migrate some paths from overburdened threads to underloaded threads. This suggests that the decisions made in the partitioning may have significant impact on parallel performance. Ravishankar [6] suggests a partitioning of models grouped together by right context; that is, given that some phones u, v, and w occur in the triphone sequence uvw, all triphone HMMs beginning with u will be gathered in the same partition, while triphones ending with vw will be distributed across partitions. We will defer to his judgment in this.
- The entire current frame must be completed before the next frame can begin computation. So, if one thread has a greater load than another, the latter thread will sit idle.

In the HMM graph, there are two kinds of channels: root and non-root. In every frame, there are are a large number of active root channels. The root channels with very good scores activate non-root channels. Because there are a large number of active root channels and these become active relatively independent of previous active channels, we statically partition these among a number of participating threads. However, it is unreasonable to statically partition non-root channels because the processor's memory in which they reside will not likely be the processor that activates the channel. Furthermore, because our target architecture is a CMP with only caches on chip, it is reasonable to allow the partitioning of non-root channels to occur at runtime.

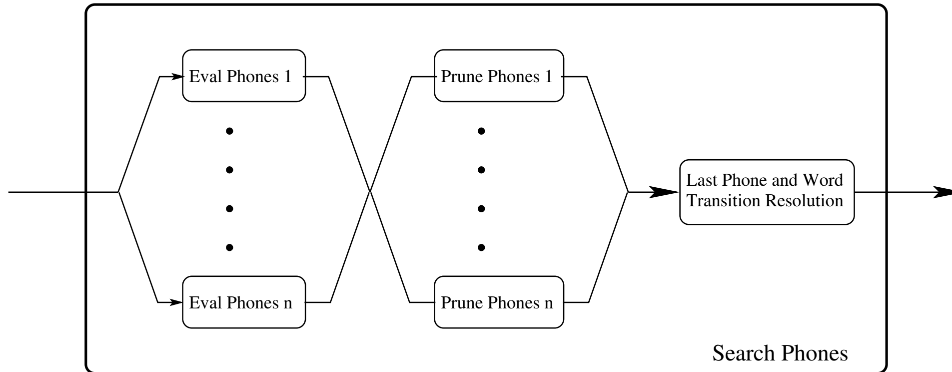


Figure 5: Search Algorithm

To perform dynamic partitioning, as root channels are evaluated, the activated non-root channels are assigned to the thread of their ancestor channel. In this way, the same processor will perform calculation on the same channel as long as it is active [6]. We tested this partitioning on a ten thread implementation and found that we attained reasonable load balance. There is some loss in efficiency, but for smaller numbers of threads the benefit of parallelism will far outweigh any effects of imbalance.

2.2.3 Benefits of Parallelization

It is possible that workstations running Sphinx with small language models run real-time already. However, parallelization on our hardware may obtain a significant performance advantage due to faster synchronization and more capability to support parallelism.

If better-than-real-time performance is obtained, it may be beneficial to widen the search beam to improve accuracy. Since the best acoustical match may not be the same phone sequence as the best sequential match, widening the scope of the search may trade excess processing power for improved accuracy. In addition to improving accuracy, we can consider improving the language models to represent more realistic dictionary sizes. Most of our tests will be run on a 5000 word dictionary, but the average speaker knows more than 50,000 words.

3 M3T Architecture

The M3T architecture is built on top of a chip multiprocessor. This subsection describes the main architectural parameters for M3T.

The main design constraints are the technology at $0.11\mu m$, the target frequency of 3GHz, and the maximum area at $350mm^2$. The component sizing has been done considering that the processor cycle is just 320 ps (pico seconds). The area and latency estimation is calculated by scaling down available processors in the market, by projections of XCACTI [4] and CACTI [9] for cache-like structures, or by personal contact with IBM employees.

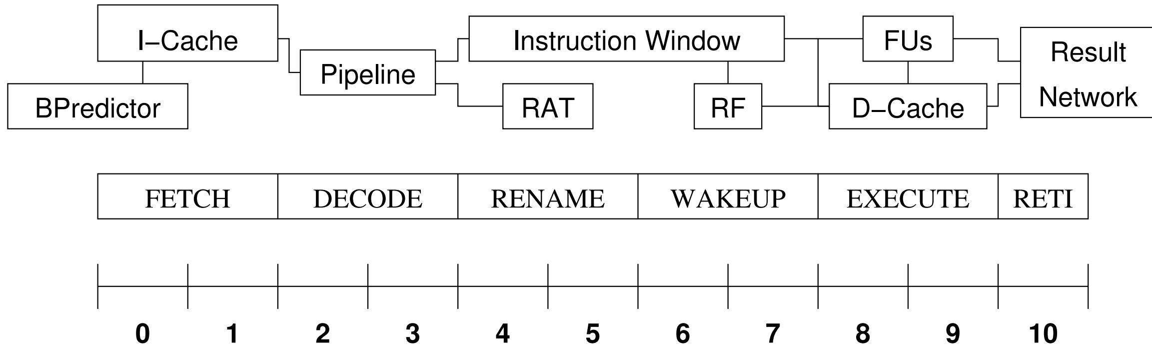


Figure 6: Single M3T core pipeline

A single M3T chip has 64 simple PPC440 like processors. The pipeline stages have been increased to 11 to meet the 3GHz requirement. The branch predictor is similar to the XScale from Intel (simple). Destination addresses are calculated with an 8 entry RAT together with a 128 entry BTB. Each processor has a small local instruction cache of 4KBytes-2way, and a data cache of 4KBytes-2way. The area required for the floating point unit is not negligible, therefore two cores share a floating point unit. Additionally a group of four processors share a small L2 cache. A group of four processors with an L2 cache and two Pending Tasks Queues is what we call a tile. The main parameters for a single processor and a tile are described in Table 1 and Table 2 respectively.

Processors	L1 DCache	L1 ICache
3GHz 2-issue BR Penalty: 9 clk FP Units: 1 (shared) Int, LD/ST Units: 2,1 Outstanding LD/ST: 2	Size: 4 KB Assoc: 2 Line: 16 B RTrip: 2 clk	Size: 4 KB Assoc: 2 Line: 32 B RTrip: 2 clk

Table 1: Parameters for a processor. *BR* and *RTrip* stand for branch and contention-free round-trip latency, respectively.

Tiles have a torus interconnect. Since a router is shared by four processors, the whole chip requires a total of 16 routers. Routers operate at the same frequency as the processor. Packets requires 1 cycle to cross a router. Additionally, for each interconnect link an additional cycle is charged. The packets sent through the network have a

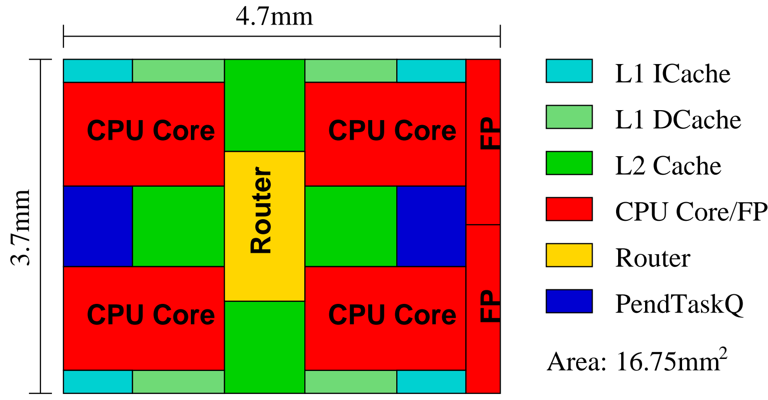


Figure 7: M3T Tile

limit of 32 bytes of payload plus the header size (13 bytes). Since the routers are connected with 120 bit width links, the total time to send a packet between two close routers would be 6 cycles (2 cycles for the routers, plus 1 cycle for the link, plus 3 cycles to transfer the packet $\frac{45*8}{120}$).

The memory subsystem is a 128 bit 200Mhz DDR. This has a peak bandwidth of 6.4GBytes/s.

Compute Power	Router	L2 Cache
Processors: 4 Shared FP Units: 2 PenTaskQ : 2 PenTaskQ size 16	Ports: 4 Torus network Cross Time: 1 clk Wire time: 1 clk Width: 120bits	Size: 64 KB Assoc: 4 Line: 32 B RTrip: 4 clk

Table 2: Parameters for a tile. A tile is composed of four processors sharing two FP units and a L2 cache. Tiles are interconnected with a torus network. PenTaskQ stands for Pending Task Queue.

We want to compare the M3T architecture with equivalent systems, but in order to be fair, it should be compared with architectures with approximately the same area constraints. Table 4 shows where the area goes in the M3T architecture. The processor size is based on projections for the IBM Power PC chips. A 2-issue out-of-order PowerPC (PPC440) has $2.2mm^2$ when configured without caches and FP unit. We estimate that the proposed M3T core should be around 10% bigger than the PPC440 because it has more stages in the pipeline, and includes control to handle an external FP unit. Table 3 explains the main differences between the M3T core and the PPC440. Besides the cache sizes, the main difference is the fastest clock in the M3T core. The M3T core cycles 4 times faster, we believe that this is possible because of three reasons. First, the caches and the M3T core SRAM structures (TLB, Register file) have been sized to meet the 3GHz requirement. Second, the M3T core has a pipeline depth of 11 instead of 7 (57% increase). Third, the PPC440 is design for the embedded market not for frequencies as high as general purpose processors.

Cache areas, and the pending tasks queue are calculated with CACTI [9]. The FP area is based on the area re-

	M3T Core	PPC440 Core
Branch Prediction	Dynamic	Dynamic
Stages Pipeline	11	7
Branch Penalty	9	5
I-Cache Size (KB)	4	32
I-Cache Assoc	2	64
D-Cache Size (KB)	4	32
I-Cache Assoc	2	64
TLB entries	32	64
issue width	2	2
Out-of-order	Yes	yes
Window Size	4	4
FP Unit	shared	none
Area w/o Caches (mm^2)	2.4	2.2
Area with Caches (mm^2)	2.9	5.3
Frequency	3GHz	750MHz

Table 3: Main differences between the M3T core and the PPC440 core. The area and frequency for the PPC are estimated by scaling down current $0.18\mu m$ implementations to $0.11\mu m$. The area estimated does not include the FP Unit.

quired for a FP unit in the Alpha21164PC ($4.62mm^2$ in $0.35/mu$m technology). After scaling down, this represents $0.46mm^2$. We also add a 10% area increase due to clock and power distribution, and I/O pads. Figure 7 shows the floor plan design for a single M3T tile.

The total chip area for a M3T chip is around $340mm^2$ for a $0.11\mu m$ technology. For approximately the same area is possible to have a chip multiprocessor with four Pentium 4 chips ($324mm^2$). The area for the CMP is calculated by scaling down the current Pentium area $217mm^2$ at $0.18\mu m$. Although it has a lot of design complexity, and it is not very clear how to cycle as fast as the previous two systems, for approximately the same area, it is possible to have an 8 issue processor. An aggressive 8 issue processor with 1.5MBytes L2 would require an area around $327mm^2$. A rough estimation is that a 8 issue processor would require around 4 times more area than an equivalent 3 three issue processor (Pentium 4 without caches). Once the processor core area is calculated ($260mm^2$), we added the area for the caches (L1 DCache 64KBytes, L1 ICache 32KBytes, L2 1.5MBytes). Alternatively, we also can have a CMP made of MIPS R10K processors. The chip can include 8 R10K-like processors ($29.3mm^2$ each), and 4 256KBytes L2 caches each one shared by two R10K-like processors ($13.7mm^2$ each) together with a 1MByte L3 cache ($50mm^2$). The total area would be around $340mm^2$. Nevertheless, it is not clear if the R10 CMP can reach the 3GHz target frequency. To simplify the comparisons between systems, for all the systems presented (8 issue, 4-Pentium 4, 8 R10K CMP, and M3T), we assume the same frequency (3GHz).

M3T	Number	Area (mm^2)	Total Area (mm^2)
CPU Core	64	2.4	153.6
FP Units	32	0.46	14.7
L1 DCache	64	0.25	16.0
L1 ICache	64	0.25	16.0
L2 Cache	16	3.50	56.0
PenTaskQ	32	0.90	28.8
Router	16	0.90	14.4
Clock & I/O	1	30	30.0
Total			**329.6**

Table 4: Area estimation for the M3T architecture.

4 Simulation Environment

We are using a brand new simulation tool called SESC based in MINT [8]. MINT is a MIPS-based execution engine. SESC extends MINT by simulating multiple architectural parameters. We evaluate two sets of architectures: M3T and several chip multiprocessors. Table 5 and Table 6 show the memory and the CPU core parameters evaluated. All the chip multiprocessors share the same parameters in the memory hierarchy to simplify the evaluation. While the latencies in the table correspond to an unloaded machine, we model contention in the entire system in detail except for the interconnection network.

	M3T			CMPs		
	L1 Cache	L2 Cache	Memory	L1 Cache	L2 Cache	Memory
Size	4KB	64KB	Inf	32KB	512KB	Inf
Assoc	2	4	N/A	4	8	N/A
Round Trip	2	5	224	3	10	227

Table 5: Baseline memory hierarchy configurations

	1-in-order	M3T Core	4-ooo	8-ooo
Issue width	2	2	4	8
Dyn issue	No	Yes	Yes	Yes
I-window size	N/A	8	32	256
BR penalty cycles	8+	9+	14+	16+
BR Predictor Type	Static	2bit	gshare	gshare

Table 6: Baseline CPU core configurations. BR and Dyn stand for Branch and Dynamic respectively.

4.1 Sphinx Setup

All our simulations are done in Sphinx, but we use three different input sets to better understand it. Sphinx is compiled with the IRIX MIPSPro compiler version 7.3 with -O2 optimization. Once the initialization has finished, Sphinx is simulated to completion. The initialization overhead in it is not negligible, and depends on the input set

provided. For the sets analyzed it goes from 1738 to 2181 millions of instructions skipped in the initialization phase.

5 Single Threaded Evaluation

To provide a baseline experiment by which to compare our parallel implementation, we will evaluate a single threaded implementation of Sphinx through function profiling, analyzing its sensitivity to input sets, and comparing the performance capability of different processor architectures.

5.1 Processor Sensitivity

When comparing an application across platforms, especially platforms that will be used for parallelization, it is useful to compare how the baseline application performs on these architectures. In this section, we analyze the single-threaded implementation of Sphinx using the BigDict input set on a single 1-issue in-order, an M3T quad, a 4-issue out-of-order, and an 8-issue out-of-order by looking at IPC, branch prediction rates, and cache miss rates.

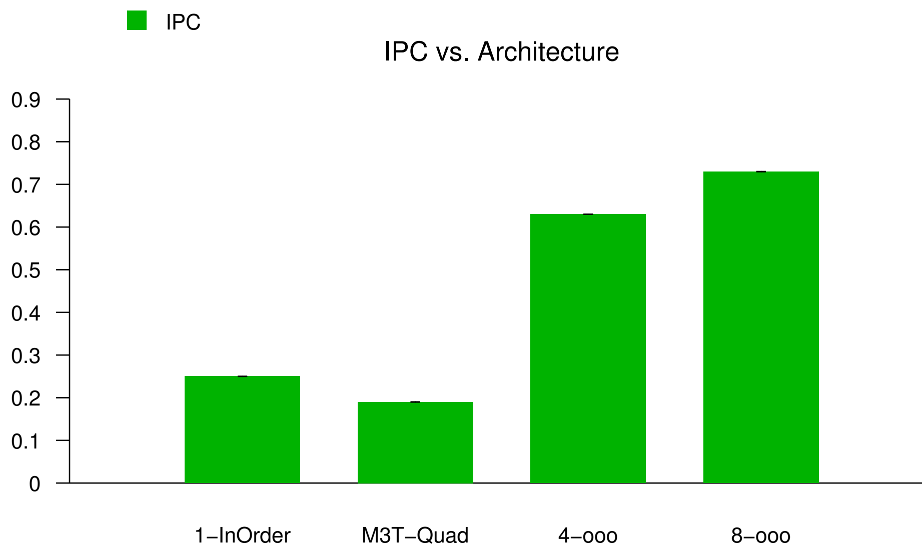


Figure 8: IPC for multiple processor configurations.

Figure 8 shows the IPC of Sphinx for each processor. As expected, the more aggressive superscalars do a better job at exploiting instruction level parallelism. Overall, the IPC for all processors is poor. The low IPC may be attributed to poor branch prediction rates and cache miss rates. As shown in Figure 9, the L2 cache miss rates are high across the board, even for larger caches of the 4-issue and 8-issue. Also, the branch misprediction rate for the out-of-order processors is also poorer than that of most SPECints.

While the more aggressive architectures show a significant performance improvement over the M3T quad and single-issue in-order, it is not proportional to the total area.

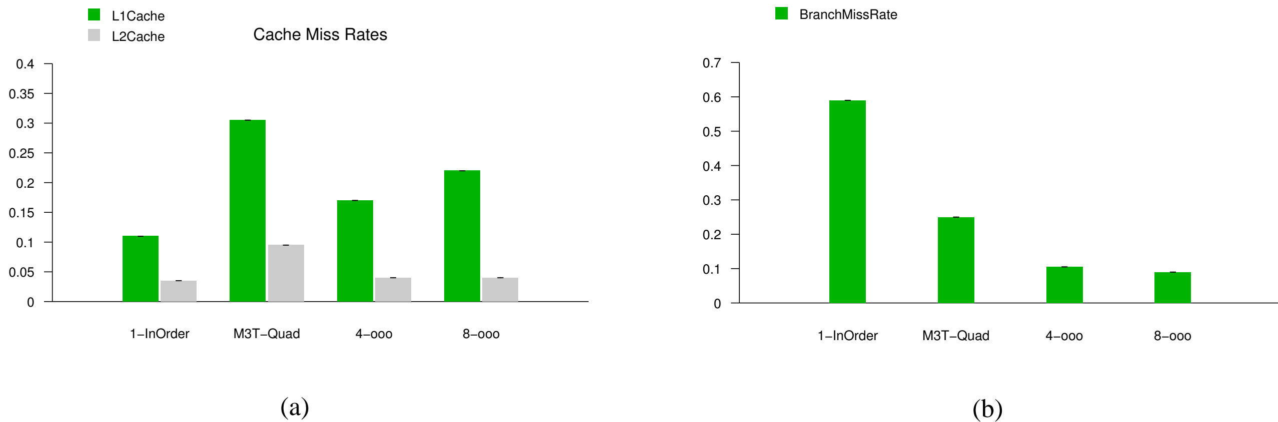


Figure 9: For each architecture: (a) cache miss rates (b) branch misprediction rates.

5.2 Input Set Sensitivity

It is well known that benchmarks are sensitive to different input sets and parameters; for Sphinx this is especially true. Although by changing some run-time configuration parameters the behavior is very different, this section only shows the impact of the input set.

Description	GoForward	MiBench	BigDict
Speech Length	5s	35s	10s
Execution time	1.4s	9.5s	9.3s
Instructions	136M	1141M	1420M
Words	117	117	3493
Phones	125716	125716	140886
Senomes	23355	23355	14006
Channels searched	124/fr	128/fr	1960/fr
Candidate words	7/fr	8/fr	205/fr

Table 7: GoForward, MiBench and BigDict input sets main characteristics. The number of instruction do not include the initialization that has been removed.

We consider three different input sets: GoForward, MiBench, and BigDict. All of them have been evaluated in SESC modeling an R10K processor. Those three input sets have similar parameters, the difference is the size of the voice stream to recognize, and the size of the dictionary.

GoForward is a test that comes with Sphinx. It is provided as a small example to verify that everything works fine. The dictionary has 114 words, and the input says “go forward ten meters”.

MiBench [3] is a suite of benchmarks for evaluating embedded systems. The suite also includes Sphinx as a benchmark. The input set consists of a small dictionary, the same as that of GoForward, with a bigger input set: a sentence that lasts 35 seconds.

BigDict uses a smaller input set than MiBench with a bigger dictionary. The input sentence lasts for 10 seconds, and the dictionary has 3493 words. Table 7 summarizes the main differences between the three input sets. The effect is summarized in Tables 8 and 9 which show the IPC, the execution time breakdown, and the cache miss rate.

Bench	IPC	% Busy	% Control	% Struct	% Mem
GoForward	1.83	45.86	21.49	0.64	15.04
MiBench	1.40	34.91	18.71	0.50	19.56
BigDict	0.74	18.59	34.27	1.35	16.89

Table 8: GoForward, MiBench and BigDict execution time breakdown.

Bench	L1 Miss Rate			L2 Miss Rate		
	Total	Load	Store	Total	Load	Store
GoForward	7.26%	8.33%	4.44%	0.28%	0.37%	0%
MiBench	22.17%	26.92%	4.49%	0.99%	1.07%	0%
BigDict	11.64%	13.72%	3.29%	8.14%	10.80%	0%

Table 9: GoForward, MiBench and BigDict L1 and L2 cache miss rates for loads and stores. Miss rates, for L1 and L2 cache, is calculated as the ratio of local accesses missing in the cache divided by the total number of accesses.

Cache miss rates are in the range of SPECint benchmarks (mcf has worse cache behavior). Nevertheless, when the size of the dictionary is increased (BigDict), the miss rate in the L2 cache increases considerable. The two main performance issues are the fetch engine and the memory subsystem. The small basic block size together with a high miss prediction rate (6%, 3%, and 12% miss rate for GoForward, MiBench, and BigDict respectively) severely restricts the average number of useful instructions fetched per cycle. The memory subsystem is also very important because L2 cache miss rates of 8% (BigDict) are very difficult to hide in modern processors.

Table 10 shows the breakdown of different types of instructions. For GoForward the average contention in the L1 cache ports is 2.96 cycles. This means that on average a memory operation needs 2.96 cycles to be issued to the cache because of contention in the cache port. For MiBench and BigDict the contention in the L1 cache is 2.11 and 1.41 respectively. This suggests that a processor with two memory ports is required for issues bigger than 3.

Bench	Branch	Load/Stores	Others
GoForward	9.05%	32.47%	58.46%
MiBench	14.45%	34.36%	51.29%
BigDict	11.07%	34.32%	54.61%

Table 10: GoForward, MiBench and BigDict instruction mix.

The impact of the different input sets also can be observed by profiling Sphinx to measure each functions weight. Our profile data was calculated skipping the initialization phase. In GoForward and MiBench most of the time was spent in the signal processing fronted, and in BigDict most of the time was spent in the search phase. For example, the

FFT function represented 24% in GoForward and a mere 1.2% in BigDict. The top three functions from GoForward represent more than 35% of the execution time, while in BigDict the same functions represent 10%. More extreme is the opposite case, where the top three functions in BigDict represent 62% of the execution time but are less than 1% in GoForward.

6 Multithreaded Evaluation

6.0.1 Experimental Setup

To analyze the effectiveness of our parallel implementation, we performed experiments on IBM and SGI workstations, and simulated the implementation on a CMP, which is the primitive form of M3T. In these experiments, the amount and type of parallelization is varied; in all but the single-threaded instance, the software is pipelined, but as many as four threads are dedicated to distance calculation, four threads dedicated to scoring, and eight threads to searching.

We will focus primarily on the results of the simple pipelined implementation of Sphinx without multithreading the individual stages. At the end of this section, we evaluate our preliminary results at multithreading each individual stage.

6.0.2 Workstation Results

On an IBM SP-2 workstation, we ran the baseline version of Sphinx with no parallelism, and a simple pipelined version of Sphinx. This parallel version uses no barriers, only locks. For this reason, we were able to speed up the native execution of Sphinx, as shown in Figure 10.

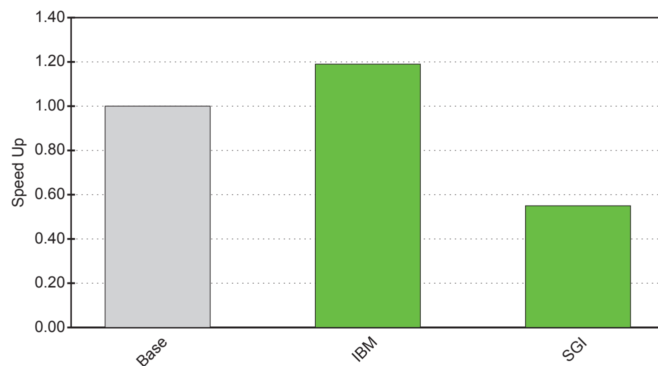


Figure 10: IBM SP-2 and SGI Workstation Results

We performed the same experiment on an SGI workstation of four R10000 MIPS processors. The results here were worse than the IBM due to worse memory latency between processors.

6.0.3 M3T Simulation Results

As the M3T architecture is our primary target for the parallelizations made, we ran a number of instances of parallelized Sphinx on the simulator. Due to Sphinx's input sensitivity, we probed several degrees and focuses of parallelization. In addition to the pipelined-only instance (1-1-1), we ran a multithreaded search-phase with two or four threads (1-1-2,1-1-4), a multithreaded scoring-phase using four threads (1-4-1), a multithreaded distance calculation phase with four threads (4-1-1), and combinations of all three (1-4-4,2-4-4,4-4-8). Some of these results are given in Figure 11.a.

Each instance featuring multithreaded scoring showed recognition problems, so we do not include these as the primary results, their accuracy being in question.

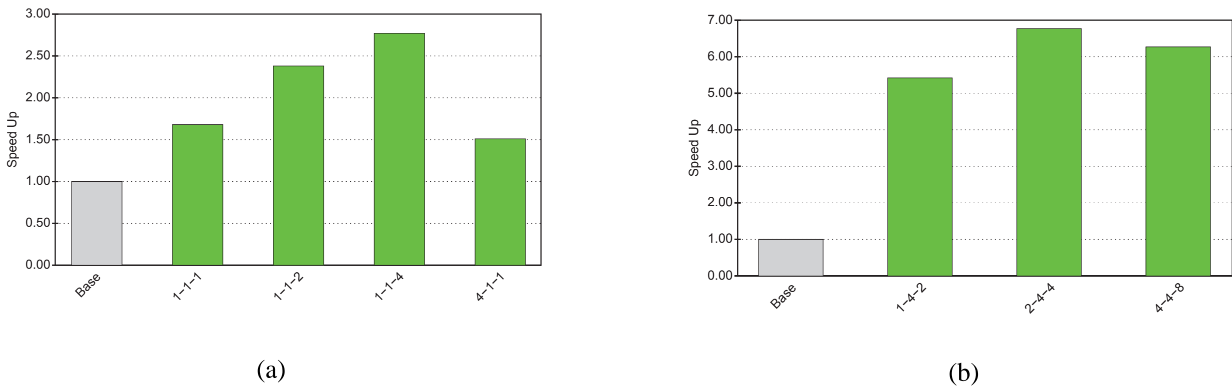


Figure 11: (a) M3T simulation results generating proper output. (b) Promising preliminary results for various multithreaded implementations.

The results obtained show steady improvement as Sphinx is pipelined, and as the number of threads in the search phase is increased. Much of the existing work on Sphinx parallelization focuses on improving the search phase [7, 2], yet our results indicate that, for this input set, while parallelizing search does yield a performance gain, the benefits diminish as more threads are introduced. Our profiles of Sphinx on small input sets show the scoring phase to be the most computationally intensive, so it seems likely that it is restricting performance.

The efficiency of the parallelization presented in Figure 11.a above is lacking. Results we have obtained from multithreading the first two stages have shown much better speedups and greater efficiency. Figure 11.b summarizes the experiments and the timing we achieved. While we find these results to be promising, we caution they are not completely trustworthy. Note that multithreading of the scoring stage results in a significant speedup over the strict pipeline. Furthermore, a combination of multithreading searching and scoring yields the greatest speedup.

6.0.4 Architectural Utilization

The cache and memory behavior of Sphinx benefited significantly from parallelization. This stems from the fact that each stage works largely independent of the other stages. Consequently, memory that is brought into the cache resides there longer. This is especially pronounced in the searching phase. For the same reason, the branch prediction logic is able to find more relationships between iterations.

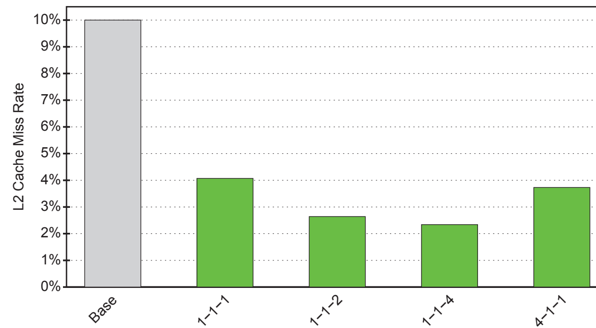


Figure 12: Cache miss rates for pipelined/multithreaded Sphinx

7 Conclusion

We have investigated methods for parallelizing Sphinx, characterized its single-threaded performance on four processor architectures, and characterized its multiple-threaded performance on several multithreaded architectures, including multiple-processor servers and a chip multiprocessor, and shown cache utilization improvement on the M3T.

We have determined that Sphinx's sensitivity to its input strongly affects the success of parallelization. However, dividing Sphinx's workload into pipelined stages can significantly improve data locality, and with it cache performance.

Also, due to the fine granularity of the parallelism we implemented, we were unable to achieve the same speedups on existing multiprocessors as we achieved in simulation. This makes a good case for chip multiprocessors capable of supporting fine grain parallelism.

The drastic improvement in cache performance due to parallelization indicates a potential direction uniprocessors may take to improve applications similar to Sphinx. M3T's ability to maintain each phase's local state almost entirely in cache leads to its superior cache performance; potentially, uniprocessor caches and branch prediction mechanisms might maintain context information per thread to produce similar performance benefits. For example, a cache may be partitioned to allow each phase or thread of a program a unique partition to help reduce conflict misses.

REFERENCES

[1] K. K. Agaram, S. W. Keckler, and D. Burger. Characterizing the SPHINX Speech Recognition System. Technical Report TR2000-33, Department of Computer Sciences, The University of Texas at Austin, 2000.

[2] S.-H. Chung, M.-U. Park, and H.-S. Kim. A Parallel Phoneme Recognition Algorithm Based on Continuous Hidden Markov Model. In *Proceedings of the 13th International Parallel Processing Symposium*. IEEE, 1998.

[3] M. R. Guthaus, S. R. J, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4nd Workshop on Wordload Characterization*, December 2001.

[4] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. L1 Data Cache Decomposition for Energy Efficiency. In *International Symposium on Low Power Electronics and Design*, August 2001.

[5] K.-F. Lee, H.-W. Hon, and R. Reddy. An Overview of the SPHINX Speech Recognition System. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 38, January 1990.

[6] M. Ravishankar. Parallel Implementation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition. Technical report, Indian Institute of Science, Bangalore, India, 1993.

[7] M. K. Ravishankar. Efficient Algorithms for Speech Recognition. Ph.D. Thesis, Carnegie Mellon University, 1996.

[8] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.

[9] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.