Appears in 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)

Architectural Optimizations for Low-Power, Real-Time Speech Recognition

Rajeev Krishna, Scott Mahlke, and Todd Austin
Advanced Computer Architecture Lab
University of Michigan (Ann Arbor)
{rkrishna, mahlke, austin}@eecs.umich.edu

ABSTRACT

The proliferation of computing technology to low power domains such as hand–held devices has lead to increased interest in portable interface technologies, with particular interest in speech recognition. The computational demands of robust, large vocabulary speech recognition systems, however, are currently prohibitive for such low power devices. This work begins an exploration of domain specific characteristics of speech recognition that might be exploited to achieve the requisite performance within the power constraints of such devices. We focus primarily on architectural techniques to exploit the massive amounts of potential thread level parallelism apparent in this application domain, and consider the performance / power trade-offs of such architectures. Our results show that a simple, multi-threaded, multi-pipelined processor architecture can significantly improve the performance of the time-consuming search phase of modern speech recognition algorithms, and may reduce overall energy consumption by drastically reducing dissipation of static power. We also show that the primary hurdle to achieving these performance benefits is the data request rate into the memory system, and consider some initial solutions to this problem.

1. INTRODUCTION

Recent years have seen a dramatic proliferation of computing technology. This trend is particularly evident in the domain of low power, hand held computing systems, where the growth of an increasingly computer savvy, interconnected society has created huge demands. This interest in portable computing, in turn, has revitalized interest in interface technologies that match the portability of the computing platform itself. On-screen keyboards and special script forms (such as Palm Computing's "graffiti"), while effective for simple jobs, leave a good deal to be desired.

The obvious next step in such interface technologies is speech recognition. Interest in such an interface is demonstrated by the current use of basic utterance (one or two words) recognition systems on many hand-held devices. True large vocabulary speech recognition under real-time constraints, however, remains beyond the computational capabilities of such systems for the foreseeable future.

The current state of the art technology in large vocabulary speech recognition is built off of stochastic search techniques over linguistic models of phonetic and syntactic language constructs. The nature of these search algorithms present a number of interesting characteristics such as the availability of extensive thread level concurrency, high memory demand, and relatively poor memory request locality. As such, these applications do not fit under the assumptions made in the design of modern microprocessor systems. This paper will explore domain specific architectures that exploit these characteristics to improve performance on speech recognition while minimizing potential negative effects on device power consumption.

We begin this analysis in Section 2 with a brief introduction to the relevant speech recognition algorithms as well as an introduction to the CMU-SPHINX speech recognition library used in this work. This introduction will serve as motivation for our multi-pipeline architecture concept, and present the source of the massive thread level concurrency found in this application space. Section 3 will introduce and describe the architecture and programming model used to expose and manage available concurrency at the hardware level. Section 4 presents a number of experiments exploring the capabilities and constraints of this architectural model, leading, after a review of related work in Section 5, into conclusions and planned future directions in Section 6.

2. SPEECH RECOGNITION

Speech recognition is the task of translating an acoustic waveform representing human speech into a textual representation of that speech. This is achieved through a series of steps, as shown in Figure 1. First, DSP style operations such as A/D conversion and extraction of key feature vectors are performed. The resulting feature vectors are then passed into a search phase, which involves two internal steps of Gaussian probabilistic scoring and linguistic model search tree traversal.

The complexity of the speech recognition task derives from natural variations in human speaking patterns (e.g. accents) and the natural ambiguity of spoken words without semantic contexts (e.g. "there" vs. "their"). The probabilistic scoring and model evaluation are intended to help account for these variations. The "recognition result" of such efforts is represented by the hypothesis that

CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.

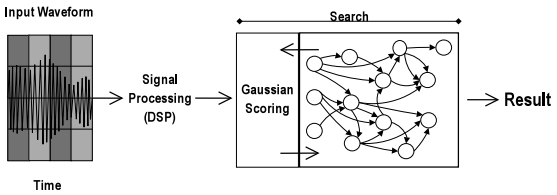


Figure 1: Overview of steps in speech recognition. Acoustic input data undergoes a number of DSP–style operations to produce discrete feature values. These values are then probabilistically scored against training data, and used to explore a stochastic model of language.

receives the highest probabilistic match to the input sequence upon conclusion of the search process. The stochastic modeling framework utilized for such operations has converged on the "hidden Markov model" (HMM), which is essentially a technique for describing complex probabilistic systems in a mathematically tractable way [20, 21].

Our speech recognition software is based on the SPHINX speech recognition library developed at Carnegie Mellon University [13, 18, 19]. SPHINX is widely regarded as a state of the art research level speech recognition infrastructure, and thus provides a good framework from which to develop our architectures. We utilize the SPHINX–2 library in this study [1]. As with many modern speech recognition systems, SPHINX operates on a knowledge base developed from models of human speaking patterns represented as a HMM. In order to provide speaker independent speech recognition, SPHINX–2 utilizes standardized models to represent each phoneme (a fundamental unit of spoken sound, e.g. the /k/ sound in 'car') in the English language. Combinations of two sequential phonemes (a "senone") are trained on acoustic data from a large sample of speakers. The "senone" unit is used to capture co-articulations, or transitional sounds made by the vocal tract between distinct phonemes. Words in the knowledge base are thus constructed as interconnected networks of trained senones, using probabilistic transitions to describe the likelihood of two sound units being heard consecutively.

This same approach is used at a higher level to describe probabilistic transitions between words. Thus, the system might distinguish the phrase "Their car" from the phrase "There car" because the first combination has a higher overall probability of occurring in normal English. It is apparent, however, that the "There car" hypothesis cannot be discarded until recognition has proceeded sufficiently far forward to determine that it is the less likely path. It is this same uncertainty, applied at the senone level, or at the level of individual sample frames, that leads to an active, concurrent exploration of a potentially large number of paths through the knowledge base. The exploration

[1]The current version of this library is SPHINX–3. The primary difference between versions is a more sophisticated continuous Gaussian scoring model. Our analysis indicates that the optimizations discussed in this paper are directly applicable to comparable portions of SPHINX–3, and that the new computation in SPHINX–3 is predominantly a small, often repeated, and well understood mathematical computation, making it an ideal candidate for ASIC style optimization.

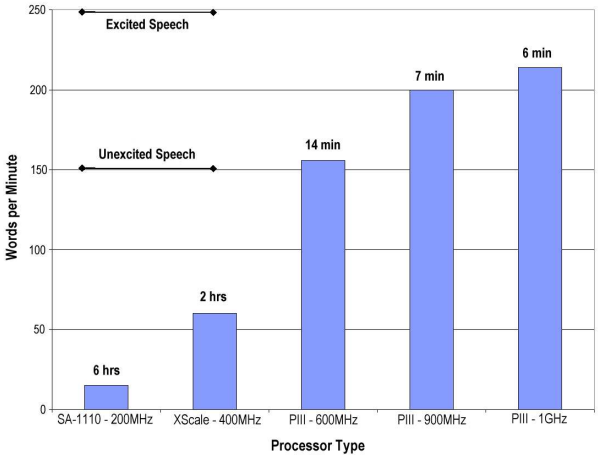


Figure 2: Performance and power considerations for speech recognition on modern architectures.

of each of these paths during any given iteration of the search phase, however, is conceptually independent from the exploration of any other path.

This characteristic concurrency potential is previously described [5, 17, 26, 9] and implies the availability of hundreds to thousands of independent threads during a given iteration of the search algorithm. Other commonly seen characteristics of speech recognition include high memory demand and relatively poor memory performance, owing to the overall footprint of the knowledge base and the highly input-dependent nature of the search process. For example, for similar cache configurations, SPHINX sees data cache miss rates on the order of one per 40 instructions as opposed to one per 200 on SPEC "Crafty", a benchmark known for poor cache behavior. Complete analysis of SPHINX is available in our earlier study [17].

These characteristics make modern architectures relatively unsuitable for running such applications, particularly on low-end computing platforms. Figure 2 provides a rough estimate of the speech recognition rate achievable on various modern computing systems. Annotated above each bar is the time each processor class would operate on a single "AA" rechargeable battery (1600 mA·Hr). It is clear that, while high end systems are within the performance range necessary for real-time speech recognition, they far exceed the power budget of portable devices. As counterpoint, processors for low-power devices simply do not have the computational power to meet the demands of speech recognition. We thereby arrive at the goal of this work: to produce architectural designs that achieve high-end desktop processor performance on speech recognition without exceeding the power budget of portable systems.

When considering these numbers, it is important to note that the recognition accuracy in these results was approximately 70–80 percent. While we do not address the accuracy problem in our work, a classic trade off in this domain is that of performance for accuracy [17]. Thus, by designing an architecture that provides for better performance, we are indirectly shifting the tradeoff in a favorable direction. Furthermore, the search phase addressed throughout this work comprises only the last steps in speech recognition. The previously mentioned DSP-style "front end"

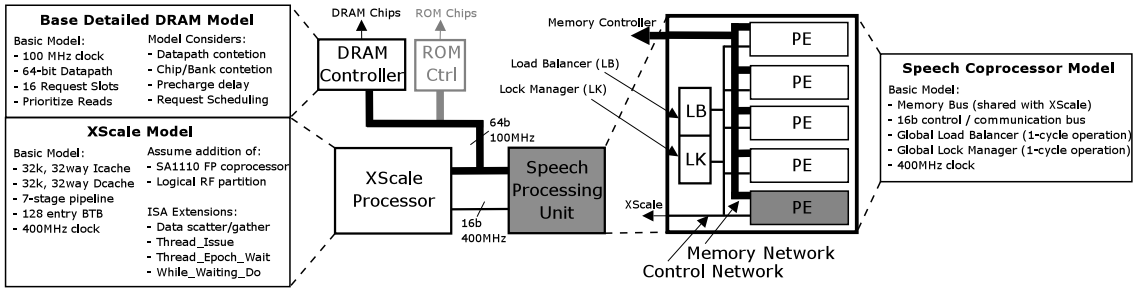


Figure 3: Overview of proposed architecture.

processing, however, represents a very small fraction of runtime computation relative to the search phase, and a number of DSP solutions exist to perform this task in a power efficient manner.

3. SYSTEM ARCHITECTURE

The concurrency available in the search phase of speech recognition immediately suggests the use of Simultaneous Multithreading (SMT) or Symmetric Multiprocessing (SMP) architectures. While the principles behind these architectures are very important, traditional SMP and SMT systems are ill-suited to this specific application space because the key to achieving desired performance is not just to exploit parallelism, but to exploit that parallelism in an efficient manner. For example, one of the primary characteristics of this domain is high memory demand, resulting in relatively long latencies. While a standard arrangement of SMP processors would improve performance (when combined with data partitioning and other similar techniques utilized in this work to expose parallelism efficiently), each individual processor would still suffer low utilization due to the underlying workload, particularly when coupled with the unsophisticated memory systems generally found on portable devices. An SMT processor could tolerate system delays by switching to other execution contexts, particularly with the availability of a large number of threads. A single such processor, while benefiting from better resource utilization, would require a massive number of resources (and correspondingly sophisticated scheduling logic) to truly exploit the amount of concurrency in this domain. Thus, following the same insight found in other works [23, 16], we arrive at the need for a combined SMP/SMT system, employing simple, low-power processing elements and distributing work through an initial static partitioning to maximize load balancing, minimize the complexity of scheduling logic, and maximize processor utilization. On top of this basic infrastructure, we incorporate architectural features to help expose parallelism easily and manage the level of exposed parallelism efficiently.

3.1 Architectural Model

Figure 3 presents an overview of our proposed SMP / SMT architecture. At this level of detail, the major features are the main system processor (modeled in our simulations as a Intel XScale 400MHz embedded processor [1] with a FPA style floating point coprocessor [14] and a number of specific ISA extensions) and the attached speech coprocessor. These divisions are logical only, and do not necessarily represent chip boundaries. Two communication infrastructures connect these processors and system peripherals. The first is a relatively small, low speed control bus connecting the XScale and the speech coprocessor. This bus is primarily used to start speech processing, collect results, and perform a small amount of inter-PE communication within the coprocessor. The second is a larger, more sophisticated memory bus, interfacing to a standard DRAM memory system.

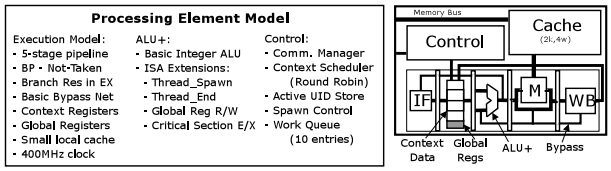


Figure 4: Details of a single processing element.

The speech coprocessor consists of a set of execution pipelines (Processing Elements or PEs), and a few resources to aid in parallelism management. These extra resources consist of a dynamic load balancer and a small number of global locks (discussed in more detail in the evaluation section), neither of which requires sophisticated logic. The dynamic load balancer requires extra interconnects with the PEs in order to identify which pipelines are overworked and which are idle. Job movement commands, however, are send over the shared command bus. All communication with the global lock manager occurs over the command bus.

The execution resources of each PE (depicted in Figure 4) are very simplistic. Due to the availability of extensive thread level concurrency, our model is able to eliminate many of the power consuming resources utilized to improve performance on modern microprocessors. For example, the processing elements have no ability to expose ILP, the scheduling and issue logic of which accounts for more than 15% of chip power on modern processors [7]. Furthermore, the inherent complexity of stochastic search programs provides strong impetus to use integer representations of data and avoid long latency computations. For example, the search tree evaluation phase of CMU-SPHINX issues only integer arithmetic and logical operations (along with loads and stores) excluding even multiplication and division operations. As a result, these pipelines consist of little more than an ALU and sufficient logic to handle control, memory, and basic mathematical operations. The basic execution model is of a standard 5-stage, in-order pipeline with

branch-not-taken prediction and branch resolution in the execute stage.

The hardware thread contexts consist of a set of integer and control registers (for the ARM instruction set employed by the XScale processor, approximately seventeen 32-bit registers), and the selection logic required to identify a ready context from which to execute. Each pipeline also contains a bank of "locally global" registers (accessed using ISA extensions) used to hold parameters and data that are global to the PE and must be accessible by any context, and a small "work queue" used to buffer a few work elements for future assignment to a context. A small cache (2k, 4way in the baseline case) is included to capture basic spatial locality and improve integration with the memory system. No cache coherence protocols are employed between PE caches as data consistency is maintained at the software level through partitioning and locking. Finally, the control logic of each PE maintains certain properties between running contexts to simplify programming and help expose more concurrency. These are discussed in the next section as part of the programming model.

3.2 Programming Model

The basic premise of our programming model is to expose potential concurrency, leaving the job of deciding how much concurrency to exploit to the architecture. As the speech processing elements are drastically scaled down versions of the main processor, the coprocessor component and the main processor component have slightly different programming constructs.

The main processor utilizes a fork/join style of constructs to expose parallelism at the program level. The primary construct on the main processor is the "THREAD–ISSUE" instruction. This instruction signals the beginning of a parallel execution phase, and has the effect of issuing an "start executing at this instruction" command to each speech PE. The main processor may then perform other work or execute a "EPOCH" instruction, forcing it to await the completion of the previously issued parallel section. As an added optimization, the main processor may schedule work and switch to a second context while awaiting an epoch. This secondary context returns control to the main context when an epoch is reached, and resumes execution when the next "wait" instruction is issued. This allows the main processor to operate in parallel with the speech coprocessor without holding up future parallel sections while it completes non-critical work. In our environment, this secondary context is used to generate inputs for Gaussian scoring for subsequent iterations of the search phase while the speech coprocessor is evaluating the current iteration. Beyond these modifications and the addition of a floating point coprocessor, the main processor in our model is identical to a standard XScale in both architecture and programming model.

In order to tolerate the latency of exposing parallelism, it is useful to allow individual speech PEs to generate new threads of execution. This model distributes the work of creating threads to the processors that are going to execute them. Threads spawned in this way allow the programmer to quickly expose segments of potential concurrency within a main processor epoch. The primary parallelizing construct on individual speech PEs is the "THREAD–SPAWN" instruction. This instruction models a function call, and may be dynamically converted to a function call if the necessary execution resources for a new thread context are unavailable.

While we are not at the point of making such considerations in this work, the design of this programming model is intended to allow for a substantial amount of dynamic parallelism control at the hardware level. While our current system only throttles thread spawning when hardware contexts are unavailable, it is possible to introduce other considerations (such as available memory bandwidth), dynamically matching the level of exposed parallelism to the immediate overall resource availability of the system, or to desired performance and power goals.

Data Partitioning and Data Locking

An important requirement to truly exploit concurrency on stochastic search applications like speech recognition is the ability to perform low overhead fine-grain data locking. As search phase algorithms traverse and evaluate search tree nodes, sequential access to node data must be maintained to avoid race conditions. Clearly, a centralized locking manager would quickly become a bottleneck in a massively parallel system evaluating tens to hundreds of search nodes simultaneously.

Our solution to this problem is the use of data partitioning and the inclusion of a UniqueID field in the THREAD–SPAWN instruction. Partitioning of search tree nodes gives each node a "home" PE which is responsible for ensuring sequential access to the node. This has the added benefit of providing an initial distribution of workload. A number of simple techniques can be used to identify the home partition of a particular data element (for example, distributing node data in such a way that the partition number may be easily acquired by a bit mask of the data address). Partitioning may be performed statically offline, and thus has no negative impact on runtime performance.

Within an individual speech PE, the UID field of the SPAWN instruction is used to identify threads that may not be executed concurrently. When a hardware context begins execution of a work unit, it inherits the UID of that work unit. Newly spawned threads may not execute if their UID matches the UID of a currently executing thread. The resulting architecture is very similar to the destination tag broadcast bus in a CAM–based out–of–order scheduler, though in this instance the broadcast is well outside of any critical execution paths, operates over a very small window, and need only be utilized on cycles involving thread spawns. In practice, we find that conflict over unique IDs is very low relative to the number of spawned jobs.

Beyond this internal locking mechanism, certain operations (e.g.: dynamic memory allocation) require the ability to maintain global lock state, necessitating the small global lock store provided by the architecture. These locks are accessed through explicit ISA extensions. A final set of extensions allows a given context to enter a "critical section", preventing the given PE's context scheduler from switching to another context during that time. This allows exclusive access to "locally global" data within the scope of a single PE.

4. PERFORMANCE EVALUATION

This section explores the effectiveness of our architectural model in improving the performance of speech recog-

nition on a embedded system level hardware platform. Simulations were performed on a modified version of the SimpleScalar/ARM toolset [4] running a hand parallelized version of the CMU-SPHINX speech recognition engine (v2.0.4) compiled with GNU-GCC for the ARM platform. A search tree traversal profile was generated using a training input stream and partitioned using the hMetis graph partitioning toolset [15]. Due to the nature of the search tree, it was possible to generate efficient partitions with no intra–word edge cuts. The architecture, however, inherently supports (through the unique ID on thread spawn) the communication necessary to accommodate less well behaved partitions. Our parallelization efforts have moved approximately 92% of the search phase code over to parallel execution on the speech coprocessor. About half of the eight percent remaining on the main processor is executed concurrently with coprocessor code through use of the secondary "epoch" context. All experiments were performed using a 11400 word vocabulary and corresponding language model generated by use of the Cambridge Statistical Modeling Toolkit [10] and based on transcripts of famous speeches and selections of text available online through the Gutenberg project [3].

Power Estimation

As an actual hardware implementation of our architecture is not yet available, we construct power estimates by combining data from a number of sources, and attempt to account for most of the power consumption in the system.

Power estimates for the XScale processor are taken from the Intel PXA250 technical specification [1]. Power estimates for the speech PE execution cores are based on a conservative area scaling of the relevant die area of the XScale processor. In each case, we compute active power for the number of seconds of execution time each processor spent actively executing, and add idle power dissipation (again, based on the technical specification) for the remaining time.

Power consumption estimates for speech PE thread contexts and caches are taken from Cacti 3 [24]. Idle power dissipation from inactive thread contexts is accounted for throughout execution. Cache active power is accumulated on cache access cycles (for each cache in the system), and idle power is accumulated on all other cycles. Cache idle power consumption is assumed to be 25% of active power. This matches the relative power dissipation levels of the XScale processor model (a reasonable assumption for on-chip caches) and also corresponds with relative idle power dissipation levels seen in other works [11].

Power estimates for the memory system are based on the SDRAM system power calculator available from Micron Technologies [2]. This system power calculator account for active, precharge, Read/Write, and background power dissipation based on activity rate for a single DRAM chip. The result of this power estimate was multiplied across the number of DRAM chips required to hold our datasets.

While this attempts to account for most of the major power consuming components of the system, there are clearly certain components that *do* vary across our architectural models, but are not accounted for here. The most obvious example is the power consumed by the interconnect network, which will vary by use and by number of processing elements. Unfortunately, a reasonable model for power consumption of this component was unavailable. As our later analysis will demonstrate, however, the interconnect network sees very low utilization in our infrastructure, and the system is *very* tolerant of interconnect latency. As such, the steps can be taken to trade off performance for lower power on the interconnect, mitigating it's effect or our power estimates.

4.1 Idealized Performance

We begin our analysis with an idealized processor model in order to explore the latency induced by various specific components. Our baseline idealized system grants each speech PE a ten element work queue, and a 2k, 4way cache with a fixed 100 cycle latency on cache misses. At 400MHz, this represents a very slow, high bandwidth DRAM. Our idealized model also assumes a single cycle delay to spawn a new thread or to transmit data along the communication network, and further assumes a contention free communication network. We explore the effect of constraining many of these resources in later sections.

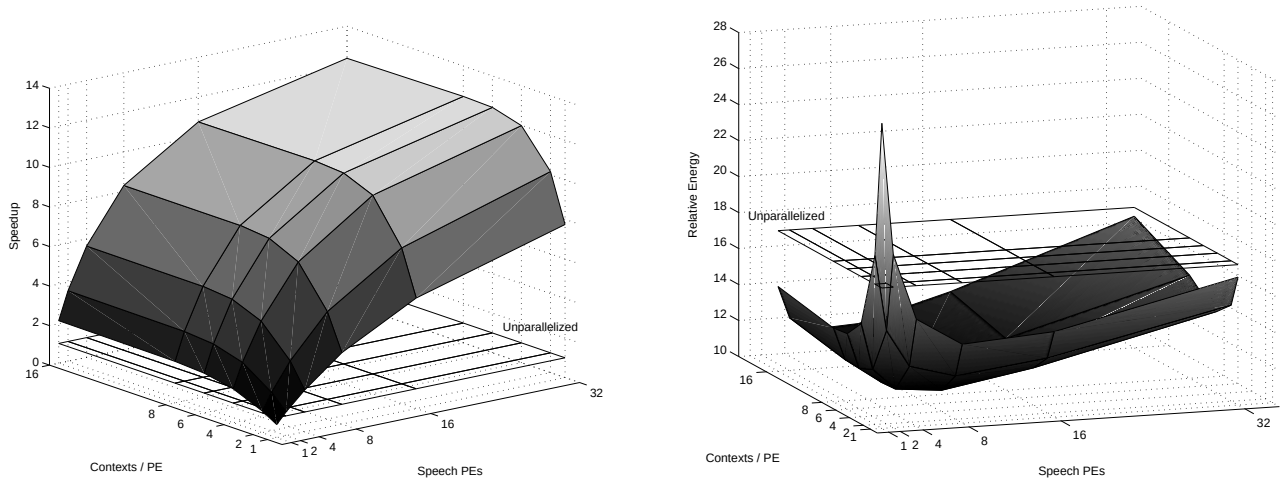
The performance speedup of various speech PE / context combinations relative to unparallelized speech recognition code on the same workload is depicted in Figure 5a. This figure depicts a number of interesting points. First, it should be noted that the overhead of parallelization is fairly substantial in our model. This is best represented by the speedup for the single speech PE, single context experiment (around 60% of unparallelized). This configuration incurs nearly the entire parallelization overhead (main processor "epoch" thread code still runs concurrently) with no extra parallel resources. Much of this overhead comes from the replacement of loops in the original code with thread spawn instructions, and the need to repeatedly perform certain data gathering operations at the beginning of each thread of execution. This data gathering is performed once, outside of the loop, in the original code, and primarily represents access to global variables or data that we do not assume the speech PEs have direct access to.

The second important point depicted in the figure is the dramatic performance improvements achievable through the addition of parallel resources. The speedups seen along the "Speech Processors" axis nearly matches the theoretical maximum linear speedup given the percent of overall code that is parallelized. This is a clear demonstration of the massive potential for concurrency available in the application space.

A final important point from this figure is the effect of adding thread contexts to each PE. The data shows that, for this configuration, the performance benefits of adding contexts appears to peak out at around eight contexts. This peak is due to a combination of factors ranging from accommodation of all available delays to insufficiently aggressive parallelization within a PE. We will consider this effect in greater detail below.

Power Considerations

A representation of relative energy consumption for our baseline model (assuming a sufficiently distributed high-bandwidth memory system) is depicted in Figure 5b. The most interesting result in this figure is that increased performance through the addition of concurrent processing resources has the effect of decreasing overall *energy* consumption for the experimental workload. It is important to note that *power* consumption increases monotonically with increased computational resources. The observed reduc-



(a) (b)

Figure 5: Speedup (a) and energy consumption (b) relative to unparallelized code on idealized model.

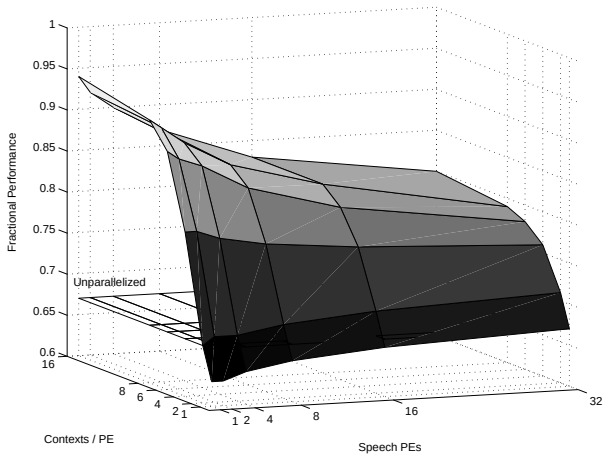


Figure 6: Fractional relative performance of 100 cycle memory latency relative to 50 cycle memory latency.

tion in energy is the result of running at a slightly higher power output for a dramatically shorter period of time. A more detailed analysis of the underlying power numbers contributing to these results shows that active energy consumption increases monotonically with increased computational resources, but overall varies little. This is expected, as the amount of "work" performed, as measured by computation required to process the input speech, is the same for all configurations. The source of the overall energy savings is a large reduction in static (idle) energy consumption. In effect, the extra power consumption of added computational resources is mitigated by the more efficient use of those computational resources (less time dissipating idle power due to latency in the system). In later sections, we will consider actual energy consumption under a more realistic set of constraints.

Tolerance of Memory Delays

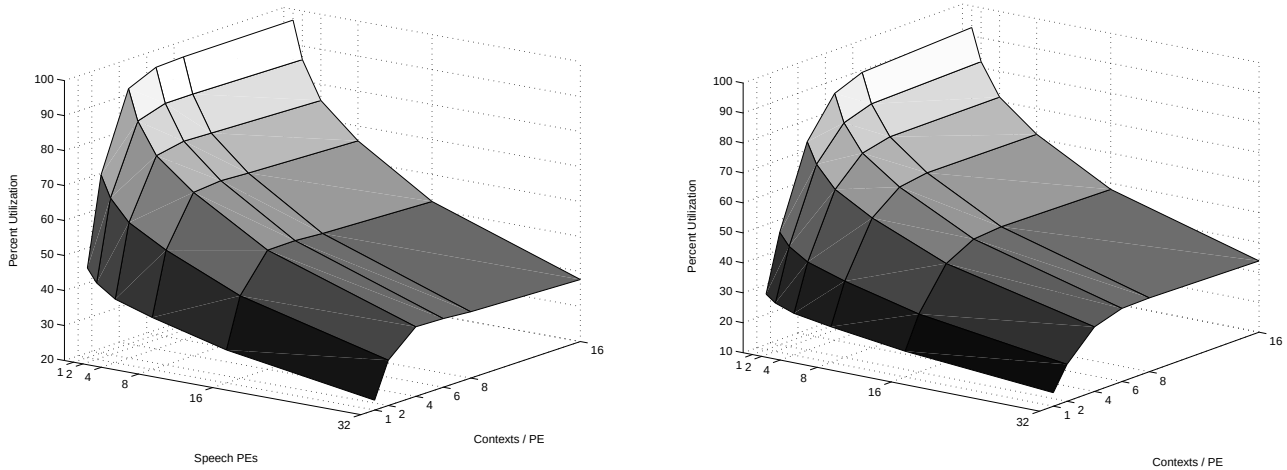
In order to explore the effectiveness of our architecture at tolerating delays in the system, we turn to the most obvious source of latency, memory requests. Figure 6 shows the fractional relative performance of moving from a fixed 50 cycle memory latency to our baseline of 100 cycles. The relative reduction in performance loss for greater numbers of thread contexts (as depicted by the higher fractional performance) is a clear indication that multiple thread contexts serve to help tolerate latencies. Indeed, the added thread contexts are able to reclaim much of the performance loss from doubling memory latency.

The beneficial effects of multiple hardware contexts is further seen in Figure 7, which shows average percent utilization of speech PEs in various configurations. We see in Figure 7b that for a 100 cycle latency, a greater number of contexts are required before peaking out (relative to 50 cycles shown in (a)), as would be expected. While some reduction is unavoidable, the addition of a couple of extra thread contexts is often enough to reclaim much of the lost utilization due to increased memory latency.

Cache Configuration

In order to handle spatial locality in data (related to accesses to multiple data sub-elements of a single search tree node), each speech PE includes a small local cache. We investigate the effect of cache size and configuration by evaluating a number of size and associativity values. The cache line size was fixed at 32 bytes to match the cache line size of the main XScale processor.

The baseline 2K, 4way cache saw a 22-30% miss rate for a 4x4 (PE/ctx) configuration. Increasing the cache size to 4K while maintaining the same associativity level achieved a 9% performance improvement and reduced the average miss rate to the 17-23% range. In contrast, reducing the cache size to 1K decreases performance by 15-17% from the baseline and increases the average miss rate to the 40% range. While this data matches expected results for a varied cache size, it is important to recognize that the actual ideal cache size for this domain will depend on the specific architecture configuration, as added thread con-



(a) (b)

Figure 7: Speech processor utilization at 50 cycle memory latency (a) and 100 cycle memory latency (b).

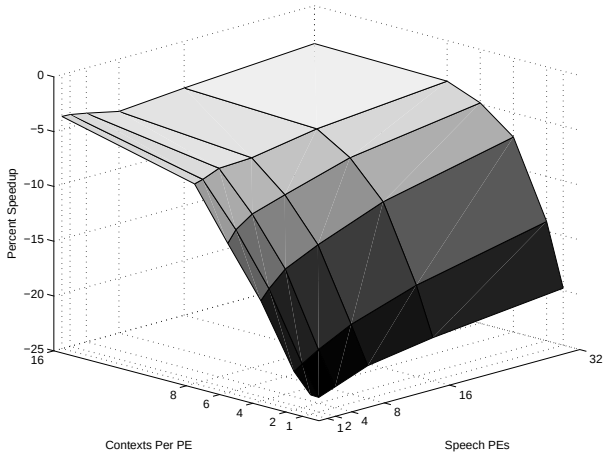


Figure 8: Relative performance loss of moving from a 2k cache to a 1k cache. Both configurations were 4-way set associative with 32byte blocks.

texts do much to tolerate the delays introduced by smaller cache sizes (shown in Figure 8). With respect to cache associativity, moving from a 4way cache to a direct mapped cache (again, for the 4x4 datapoint) had the effect of reducing performance by around 30%, while increasing to a fully associative cache had a trivial positive impact on performance.

Register Pressure

In order to explore the potential for reducing the amount of hardware resources required for each thread context, we perform a simple register pressure test. Utilizing the -ffixed flag in GCC, we performed a series of experiments successively eliminating registers from the compiler's allocation pool for our targeted search phase code. This experiment produced two interesting results. First, register pressure is relatively high on the speech PEs. It was not possible to successfully compile the program with more than five registers eliminated. Experiments with five registers eliminated (11 remaining) demonstrated a 5% slowdown as compared to the unrestricted code. While this is a trivial slowdown relative to the potential performance improvements we have discussed thus far, it does suggest that one cannot eliminate enough registers to significantly impact the architectural design. It is important to note, however, that GCC's register allocator is rather poor, and a better register allocator may relax the register pressure on the speech PEs.

The far more interesting and useful result was revealed in a register pressure experiment on the main XScale processor (again, during the search phase code). Restricting the compiler to two general purpose registers and two floating point registers (out of fifteen and seven respectively) resulted in a slowdown of less that 0.5%. This suggests that the extra "epoch" context used to perform background processing during parallel code sections can be entirely virtualized through a logical partitioning of the existing register file. With compiler support, this extra context could be added with a minimal change to the existing architecture. In this instance, a better register allocator would simply improve the already sufficient result.

Work Queue Size

The "work queue" constitutes a small bank of register space on each speech PE which is used to buffer work elements for future assignment to thread contexts. As our parallelization algorithm converts a thread spawn into a function call when resources are unavailable, it becomes possible for a previously spawned thread context to complete it's work unit but remain empty because the context running the section of code that issues spawn instructions is busy operating on a work unit of it's own. As only one context may run at a time, a considerable number of cycles may pass before the PE returns to the thread spawning code. The presence of a small work queue allows the newly free context to accept a work unit from this small buffer, improving the opportunity for overall processor utilization.

We evaluate the usefulness of this work queue by varying its size. Eliminating the work queue entirely demonstrated a 10% slowdown on average for smaller numbers

of contexts. As the number of contexts passed the utilization point (approximately 8 contexts in the ideal model), this slowdown disappeared as potential work queue usage points were eliminated. A small work queue (less than five entries), however, was sufficient to to reclaim any performance loss.

It should be noted that our evaluation did not consider true elimination of the work queue. Work queue entries were still used to hold thread spawn requests that resulted in a UID conflict, and to accept migrated jobs. True elimination of the work queue would simply exacerbate the slowdown as contexts were forced to stall on conflicting UIDs during a thread spawn, and dynamic load balancing was scaled down.

Thread Spawn Delay

Our idealized model assumes no delay incurred in the process of spawning threads. We evaluate the effect of adding a realistic constraint by considering situations in which the act of spawning a thread stalls both the spawning and receiving context to account for register copies and context setup. We consider delays in number of cycles of (2 + number of registers to copy) and (10 + number of registers to copy) and find no appreciable affects on performance, strongly suggesting that this is not a architectural constraint on performance.

Communication Infrastructure

Our idealized model assumes no delay or contention on the inter-PE communication network. This network is primarily used at the beginning and end of a parallelized section of code (in communication between the main processor and speech coprocessor), and during job migration and global lock acquisition. As such, contention for this network should be fairly low. We test this hypothesis by implementing a basic 8-bit bus as a communication network and assuming that a communication allocates the bus for the number of bytes of transfer plus two extra cycles of protocol overhead. We find no appreciable affect on performance (less than 0.2% slowdown on a 32 PE x 32 context configuration), suggesting that this too is not an architectural constraint on performance. It should be noted that the lack of contention is due to the lack of communication between processors. As such, variations in partitioning that lead to greater job migration (either due to dynamic load balancing or cut edges) could lead this to become a more significant contributor to performance degradation.

Global Locking

The global locking mechanism provides the ability to establish mutual exclusion across all running threads. This is a very slow, easily congested means of performing locking, and as such every effort is made in the software to utilize fine–grain locking mechanisms available on individual processing elements. The facility is provided, however, because certain operations require global exclusion capabilities.

As access to the global lock manager is inherently constrained by access to the communication infrastructure, we would expect no greater impact on performance due to the speed of this very simple hardware as due to the bandwidth through the communication network. Our baseline results show that both the delay due to cycles of lock manager operation, and indeed delays due to contexts waiting for currently held global locks is trivial relative to other performance factors.

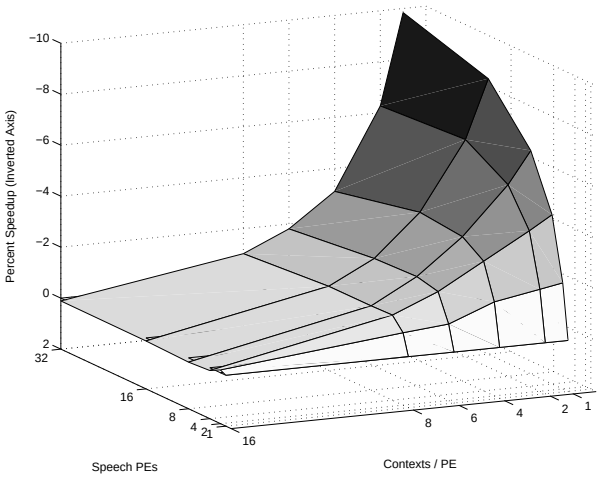


Figure 9: Percentage slowdown relative to baseline by eliminating dynamic load balancing.

Load Balancing

Our baseline architecture includes a global job scheduler to perform dynamic load balancing between speech PEs. While the profile based partitioning scheme used to perform initial work distributions proves very effective in balancing speech PE usage over the duration of the program, it is unable to account for imbalances occurring during individual "SPAWN to EPOCH" segments. The result is that each PE spends approximately the same time actively computing, but the accumulation of small workload imbalances leads to a correspondingly large number of idle cycles waiting for other processors to finish and reach the next epoch. In order to mitigate this effect, the global job scheduler identifies jobs awaiting assignment to a context (waiting on the work queue) of a PE with no free context, and issues commands to migrate such job to the work queue of a PE with free contexts. Migratable jobs are identified by the program, and may not spawn new jobs themselves, or require access to data local to the home processor. This set of constraints minimizes the impact of job migration on the communication network, and allows the job to execute with highest possible throughput on the receiving PE. As these constraints basically match threads representing the inner-most loops of nested search algorithm traversals, they are quite plentiful.

The relative slowdown caused by removal of the load balancing engine is depicted in Figure 9. This chart clearly shows that, for a small number of contexts, the effect of load balancing can be quite substantial, particularly for a larger number of partitions. As an increased number of partitions increases the potential for imbalance, this result is quite reasonable. An interesting characteristic, however, is how quickly this effect is dissipated by the addition of thread contexts. As the load balancing engine will only migrate a job from the work queue of a busy processor to a free context on another processor, the addition of thread contexts directly reduces the availability of jobs eligible for migration. In the extreme case, free contexts are always

available, and no useful job migration occurs.

To extend this analysis, we consider the aggressiveness of load balancing by placing a constraint on the number of free contexts that must be available on the receiving PE before a job may be migrated to it. Our results show that, such reduced aggression (requiring a larger number of free contexts) only serves to degrade performance.

Partition Quality

As discussed previously, knowledge base data partitioning serves the dual purposes of aiding in fine grain locking support and providing an initial distribution of workload. Early simulations during the development of this research showed that parallelization without partitioning seriously constrains the achievable throughput (approximately 3x for 16 processors in an idealized framework). The previous subsection demonstrated that, despite profiling, this initial distribution can only balance overall workload, allowing for substantial imbalance in smaller computational segments. We therefore explore whether profile based partitioning is in fact necessary or helpful, particularly given the need for dynamic load balancing regardless.

We consider the effect of two partitioning options. The naive approach distributes partitions based purely on number of search tree nodes, with no consideration of use. The more sophisticated partitioning scheme is the baseline used throughout this paper, weighing each search tree node by access frequency during a profiling run, and partitioning to equalize weight. The results of this evaluation are shown in Figure 10 and depict a fairly substantial slowdown for the naive partition scheme for smaller numbers of speech PEs (i.e. smaller numbers of partitions). This slowdown is substantially greater than the effect of eliminating load balancing, and demonstrates simply that the relatively slow job migration implemented by the load balancing infrastructure is not as effective a method of exploiting parallelism and tolerating delays as the fast job assignment within a single speech PE. A secondary effect of the naive partition is increased pressure on the communication network as a substantially (often over 3x) greater number of jobs are migrated in an attempt to balance workload. The graph also appears to show a peak in slowdown at 8 processors, and a relative improvement in performance for a greater number of partitions. Our analysis indicates that this relative improvement derives from the fact that, given enough partitions, frequently traversed nodes will inherently end up in different partitions simply through an effort to equalize the number of nodes. As profile based partitioning can be performed offline in a "do once, use forever" fashion, we see no benefit to avoiding a detailed, profile based partitioning step.

4.2 Detailed Memory System Analysis

The discussion thus far has demonstrated the potential performance improvements achievable through efficient use of available parallelism. The second primary characteristic of speech recognition applications is high memory demand. Thus, in order to fully consider true system performance, we incorporate a highly detailed full memory system SDRAM simulator from the University of Maryland [25] to our modeling infrastructure. For this and all subsequent evaluation, we move to a less idealized model, not only incorporating the detailed memory system, but also incorporating a realistic thread spawn delay (1 cycle

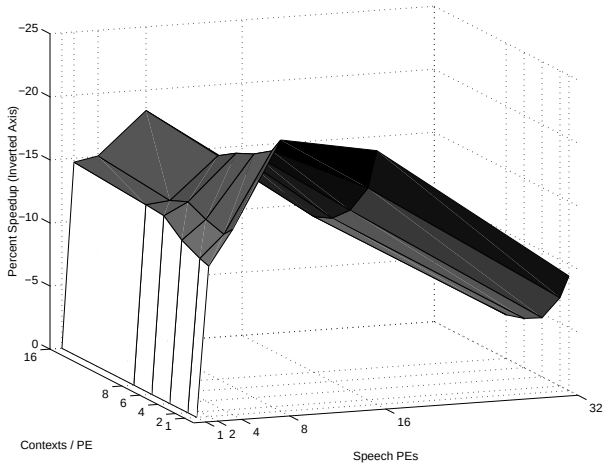


Figure 10: Percentage slowdown of naive partition relative to baseline.

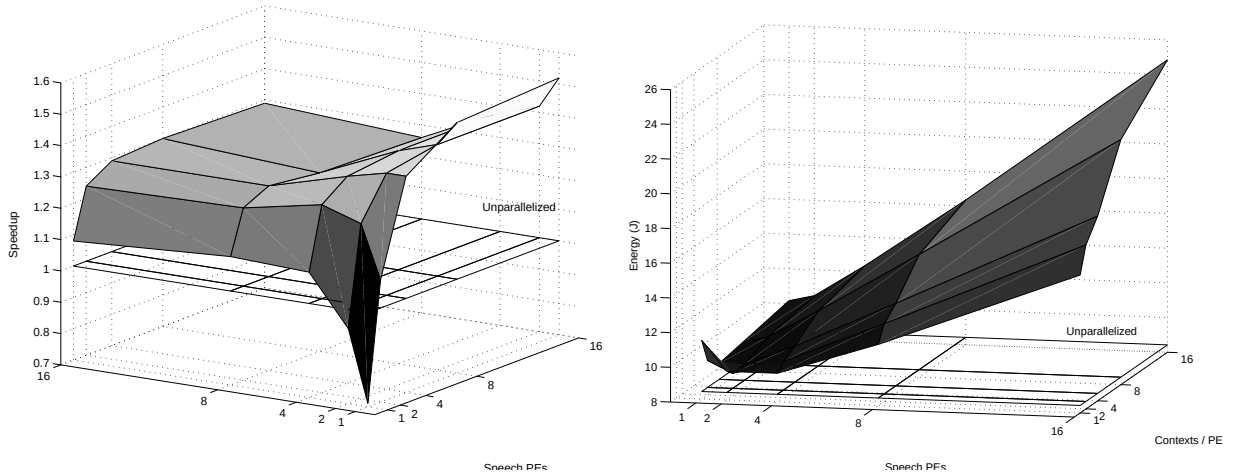
for every two registers copied plus two control cycles) and communication interconnect (16-bit bus). As discussed previously, these factors do not have a substantial impact on performance, but are included for completeness. All subsequent experiments also incorporate the load balancing job scheduler (the activity of which is constrained by thread spawn delay, it's own internal delay model, and the communication delay) and utilize the profiled initial partition. Each speech PE is given a 2k, 4-way cache, and a 5 entry work queue. The main processor is constrained to half of it's total registers during parallelized sections to account for the logical "background" context.

Figure 11 shows the effects on speedup and energy of detailed memory system modeling using a 100MHz SDRAM system with one channel and a 64–bit bus. Clearly, memory access constitutes a key bottleneck in our system. It is important to note, however, that even the relatively small performance improvements seen here can mitigate some of the energy consumption of added computational resources. Based on full system power estimates, the unparallelized version of this code running on our 400MHz XScale model would achieve around 60 words per minute and run for around 4.5 hours. By comparison, a single PE with 4 contexts would be able to achieve nearly 70 words per minute for around four hours. Due to the clear memory bottleneck, added processors were able to do little to improve performance, and thus did not produce the static power dissipation seen in the idealized results..

In order to explore memory system effects more thoroughly, we categorize memory characteristics as those related to DRAM request rate, or the ability to issue requests into the memory system at a high rate, and those related to data transfer, or the constraints due to bus, channel, chip, and bank contention within the memory system.

DRAM Request Rate

We consider the impact of DRAM request rate by manipulating the frequency of the DRAM system and the number of memory channels (thereby the number of simultaneous outstanding requests). Figure 12 considers the relative performance between a 100MHz memory and a 200MHz memory. As shown in the figure, the extra bandwidth to



(a) (b)

Figure 11: Speedup (a) and relative energy (b) charts for full memory system simulation (100 MHz DRAM, 1 channel, 64 bit interconnect).

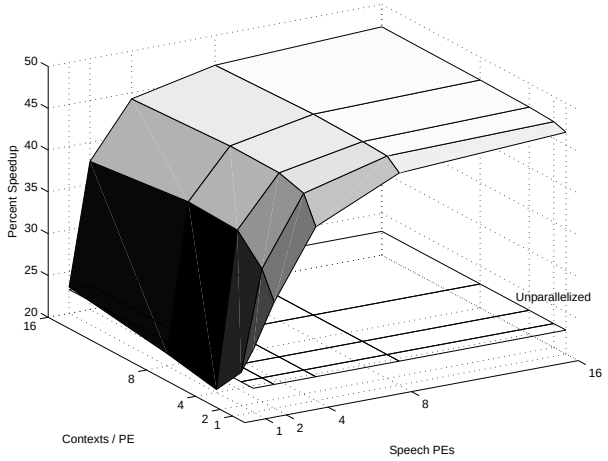


Figure 12: Percentage speedup of 200MHz DRAM system over 100MHz DRAM.

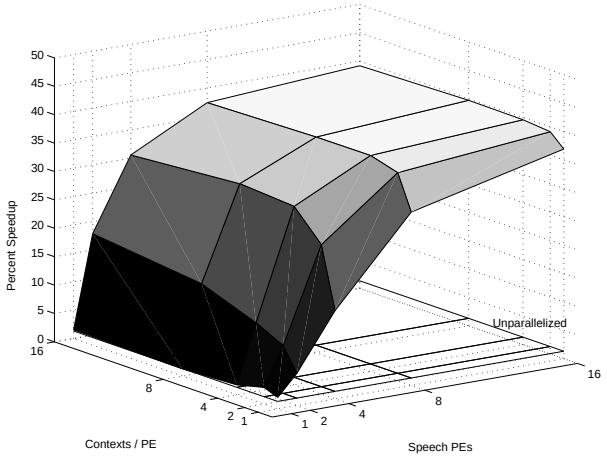


Figure 13: Percentage speedup of two independent memory channels over single channel.

the DRAM system is best exploited by an increasing the number of speech PEs. As an individual PE can only issue one memory request per processor cycle, and the DRAM is only relevant on cache misses, increasing the number of contexts does not utilize the DRAM as aggressively. Figure 13 considers the relative performance improvement by doubling the number of simultaneous outstanding requests. This too has a fairly substantial impact on performance, and shows benefits for both an increased number contexts and processors.

DRAM Data Transfer Rate

We consider the impact of DRAM data transfer contention and internal contention by manipulating the width of the DRAM data channel and by attempting to distribute data across banks by partition to reduce bank contention. These evaluations are shown in Figures 14 and 15 respectively, and show little relative performance gain as compared to the DRAM request issue modifications performed in the previous section. This strongly suggests that the key bottleneck in the memory system is the frequency and number of requests that can be handled within the requisite timeframe, not contention for internal DRAM resources.

Memory Partitioning

Given the previous analysis regarding bottlenecks in the memory system, we consider one possible optimization: moving immutable knowledge base data to a high bandwidth ROM, and eliminating those requests from the memory system entirely. This optimization is possible because large portions of the knowledge base data (for example, the Gaussian probability tables, logarithmic computation tables, and language model information) are read only. The dynamic search tree, containing current scoring and active node information, must be writable, but constitutes a relatively small portion of the overall memory footprint of the program. In order to gain insight into the potential of such an optimization, we isolate memory requests to im-

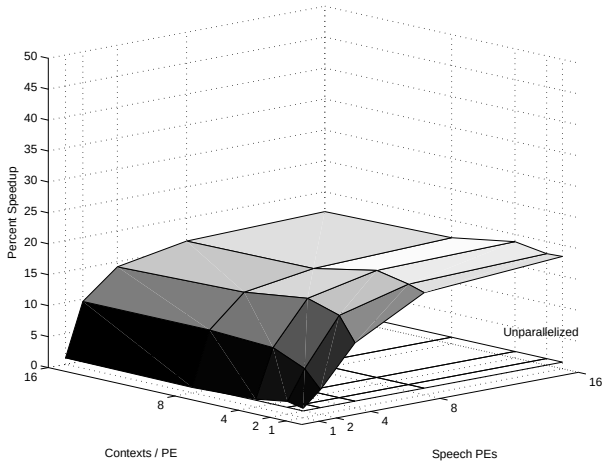


Figure 14: Percentage speedup of 16 Byte DRAM channel width over 8 Byte.

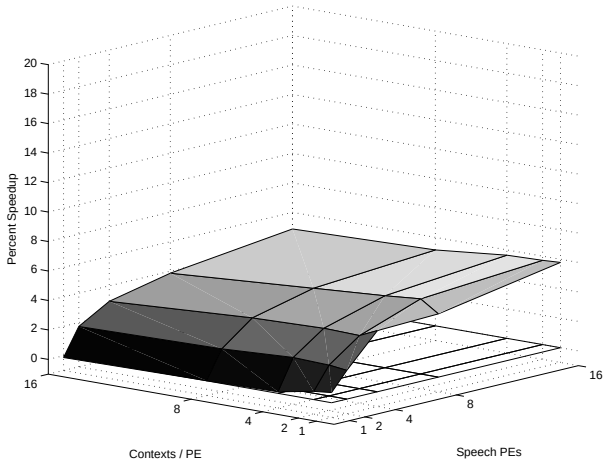


Figure 15: Percentage speedup of 100MHz, 1 channel, 8 Byte wide memory with data placement optimizations over baseline.

mutable knowledge base data and assign a fixed memory access latency of 60 cycles to such requests (again, representing a slow flash memory or ROM system). The result of this experiment is shown in Figure 16. As 60 cycles is far greater than the average latency to the DRAM system when request rate is low (around 40 processor cycles), we see a marked slowdown in configurations with few processors and contexts. As counterpoint, however, we see a notable increase in relative performance as demand for the memory system increases. This demonstrates how important access to the immutable knowledge base data is to overall program performance, and suggests that a memory stream partitioning technique may be quite effective in reducing the memory bottleneck.

5. RELATED WORK

The idea of exploiting available parallelism in the speech recognition domain to improve performance is not new, and a number of parallel architectures and algorithms have

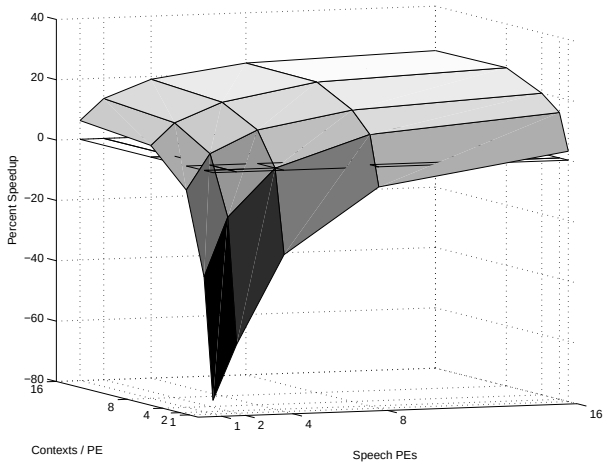


Figure 16: Percentage speedup of 100MHz, 1 channel, 8 Byte wide memory with data placement optimizations and static data set to fixed 60 cycle memory latency over identical memory system without static data partitioning.

been investigated in other works. The primary difference between these works and this paper is that our focus is not only on performance, but also on constraining power to allow speech recognition in a very confined domain, dramatically changing the characteristics of the problem. Hon explored several approaches to hardware speech recognition, including AT&T's Graph Search and ASPEN Tree Machines and CMU's PLUS architecture [12]. These machines all consider the same basic problem we are looking at here, with a focus on pure recognition performance on large, multiprocessor systems. These are also on older architectures and thus consider far smaller vocabularies.

Anantharaman and Bisiani develop a custom hardware accelerator for speech recognition [6]. Their work is primarily focused on transformations from the algorithm leading to a custom hardware implementation and thus acquires a number of algorithm specific characteristics. Our work, in contrast, exploits only the most general aspects of the speech recognition domain, achieving substantial performance improvements through a general architectural model that is applicable to a number of problems within the stochastic search domain. Chatterjee and Agrawal consider connected speech recognition on the MARS pipelined processor [8]. Their primary focus in this work is implementation and performance on their existing architecture and do not consider architectural arrangements to maximize recognition performance or power considerations.

Ravishankar [22] presents a parallel implementation of the SPHINX beam search heuristic. Many of the techniques described in his work are utilized in our parallel implementation, with modification for our programming model. He is able to demonstrate 3x performance improvements on a 4-processor SMP system, tracking well with our experimental results in low power, embedded environment. Our work also demonstrates the effectiveness of added SMT support it improving performance and mitigating delays.

With regards to SMT and SMP, recent work has also shown potential benefits of hybrid system similar to ours.

Sasanka et al consider chip multiprocessors and SMT facilities on out of order processors for multimedia benchmarks and find that CMP processors provide the best energy efficiency, and hybrid systems show the ability to accommodate both high performance and high energy efficiency [23]. Kaxiras et al observe similar trends on mobile phone DSP workloads [16]. While the search phase of speech recognition that we address is not DSP-like in nature, the basic principles of power saving through exploitation of concurrency remains the same.

6. CONCLUSIONS AND FUTURE WORK

This work considers the potential use of thread level parallelism inherent in the domain of continuous speech recognition applications to make such capabilities available within the computational and power constraints of hand-held and portable systems. We demonstrate that a high-concurrency execution environment with SMT-like facilities for tolerating latency can dramatically improve performance of speech recognition, and that a reduction in the effect of static power dissipation can actually reduce overall energy consumption for a given workload. We demonstrate that the primary bottleneck to implementation of such an infrastructure is access to the memory system, and further identify the bottleneck as request rate into the memory system as opposed to contention for internal memory system resources. Overall, we find that from an original system performance of 60–70 words–per–minute for around 4 hours of continuous speech totaling around 16000 words (on a single "AA" battery), we are realistically able to achieve around 95–100 words–per–minute for 3 hours, totally around 18000 words.

Our analysis clearly places the focus of future work on memory system optimizations. One promising avenue is partitioning of the data request stream by type, reducing the demand on individual memory resources. Future work will also need to consider power consumption related factors such as heat dissipation, as our architecture must operate at a higher power level (for a shorter duration) than the original. As counterpoint, given a memory model capable of handing the demand of this application, it may be possible to employ voltage and frequency scaling techniques, trading off excess performance for lower energy and instantaneous power consumption.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work is supported under the DARPA/MARCO Gigascale Silicon Research Center and C2S2 Research Center. Additional support was provided by the National Science Foundation, grant number CSA-0310511. Equipment support was provided by Intel.

8. REFERENCES

[1] Intel PXA250 processor. http://developer.intel.com/.

[2] Micron Technologies. http://www.micron.com/.

[3] Project Gutenberg. http://promo.net/pg/.

[4] Simplescalar toolset. http://www.simplescalar.com.

[5] K. Agaram, S. Keckler, and D. Burger. A characterization of speech recognition on modern computer systems. In *Proceedings of 4th Annual Workshop on Workload Characterization*, December 2001.

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

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[8] S. Chatterjee and P. Agrawal. Connected speech recognition on a multiple processor pipeline. volume 2, pages 774–777, May 1989.

[9] C.Lai, S.Su, and Q.Zhao. Performance analysis of speech recognition software. In *Proceedings of 5th Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[10] P. Clarkson and R. Rosenfeld. Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings of EUROSPEECH'97*, pages 2707–2710, 1997.

[11] C.Zhang, F. Vahid, and W. Najjar. A Highly–Configurable Cache Architecture for Embedded Systems. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[12] H. Hon. A survey of hardware architectures designed for speech recognition. Technical Report CMU-CS-91-169, August 1991.

[13] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2):137–148, 1993.

[14] D. Jagger and D. Seal. *ARM Architecture Reference Manual (2nd edition)*. Addison–Wesley, 2000.

[15] G. Karypis. Metis family of multilevel partitioning algorithms. http://www-users.cs.umn.edu/ karypis/metis/metis/index.html.

[16] S. Kaxiras, G. Narlikar, A. Berenbaum, and Z. Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, November 2001.

[17] R. Krishna, S. Mahlke, and T. Austin. Insights into the memory demands of speech recognition algorithms. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues*, May 2002.

[18] K. Lee, H. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34:35–44, 1990.

[19] K.-F. Lee. *Automatic Speech Recognition: The Development of the SPHINX System*. Klewer Academic Publishers, 1989.

[20] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2):257–286, February 1989.

[21] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall, 1993.

[22] M. Ravishankar. Parallel implementation of fast beam search for speaker-independent continuous speech recognition. *Computer Science & Automation*, 1993.

[23] R. Sasanka, S. Adve, Y. Chen, and E.Debes. Comparing the Energy Efficiency of CMP and SMT Architectures for Multimedia Workloads. Technical Report UIUCDCS-R-2003-2325, 2003.

[24] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, August 2000.

[25] D. Wang and B. Jacobs. MASE DRAM memory simulator manual. http://www.ece.umd.edu/courses/enee759h.S2003 /references/mase_dram.pdf.

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