A Gaussian Probability Accelerator for SPHINX 3

Binu K. Mathew, Al Davis, Zhen Fang

UUCS-03-02

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

November 18, 2002
Revised on July 22, 2003

Abstract

Accurate real-time speech recognition is not currently possible in the mobile embedded space where the need for natural voice interfaces is clearly important. The continuous nature of speech recognition coupled with an inherently large working set creates significant cache interference with other processes. Hence real-time recognition is problematic even on high-performance general-purpose platforms. This paper provides a detailed analysis of CMU's latest speech recognizer (Sphinx 3.2), identifies three distinct processing phases, and quantifies the architectural requirements for each phase. Several optimizations are then described which expose parallelism and drastically reduce the bandwidth and power requirements for real-time recognition. A special-purpose accelerator for the dominant Gaussian probability phase is developed for a 0.25μ CMOS process which is then analyzed and compared with Sphinx's measured energy and performance on a 0.13μ 2.4 GHz Pentium4 system. The results show an improvement in power consumption by a factor of 29 at equivalent processing throughput. However after normalizing for process, the special-purpose approach has twice the throughput, and consumes 104 times less energy than the general-purpose accelerator. The energy-delay product is a better comparison metric due to the inherent design trade-offs between energy consumption and performance. The energy-delay product of the special-purpose approach is 196 times better than the Pentium4. These results provide strong evidence that real-time large vocabulary speech recognition can be done within a power budget commensurate with embedded processing using today's technology.

1 Introduction

For ubiquitous computing to become both useful and real, the computing embedded in all aspects of our environment must be accessible via natural human interfaces. Future embedded environments need to at least support interfaces such as speech (this paper's focus), feature, and gesture recognition. A viable speech recognizer needs to be speaker independent, accurate, cover a large vocabulary, handle continuous speech, and have implementations amenable to mobile as well as tethered computing platforms. Current systems fall short of these goals primarily in the accuracy, real time, and power requirements. This work addresses the latter two problems. Modern approaches to large vocabulary continuous speech recognition are surprisingly similar in terms of their high-level structure[16]. Our work is based on CMU's Sphinx3[7, 10] system. Sphinx3 uses a continuous model that is much more accurate than the previous semi-continuous Sphinx 2 system but requires significantly more compute power.

Sphinx3 runs at 1.8x slower than real time on a 1.7 GHz AMD Athlon. Performance is hardly the problem since Moore's Law improvement rates means all that is needed is a little time. A much more important problem is that the real time main memory bandwidth requirement of Sphinx3 is 800 MB/sec. Our 400 MHz StrongARM development system has a peak bandwidth capability of only 64 MB/sec and this bandwidth costs 0.47 watts of power. A reasonable approximation is that power varies with main memory bandwidth for Sphinx3 indicating that Sphinx3 is at least an order of magnitude too slow and consumes an order of magnitude too much power for embedded applications. This provides significant motivation to investigate an alternate approach.

A simplistic view of Sphinx3's high-level structure consists of 3 phases: front-end signal processing which transforms raw signal data into feature vectors; acoustic modeling which converts feature vectors into a series of phonemes; and a language model based *search* that transforms phoneme sequences into sequences of words. The process inherently considers multiple probable candidate phoneme and word sequences simultaneously. The final choice is made based on both phoneme and word context. We focus on the dominant processing component of the acoustic and *search* phases in this paper which we call GAU and HMM respectively.

The organization of Sphinx3 is shown in Figure1. Rectangles represent algorithmic phases and rounded boxes represent databases. The numbers in parenthesis are the approximate on-disk size of the databases before they are loaded into memory and possibly expanded. The 3 phases are front end signal processing, Gaussian probability estimation, and Hidden Markov Model evaluation, subsequently referred to as FE, GAU, and HMM.

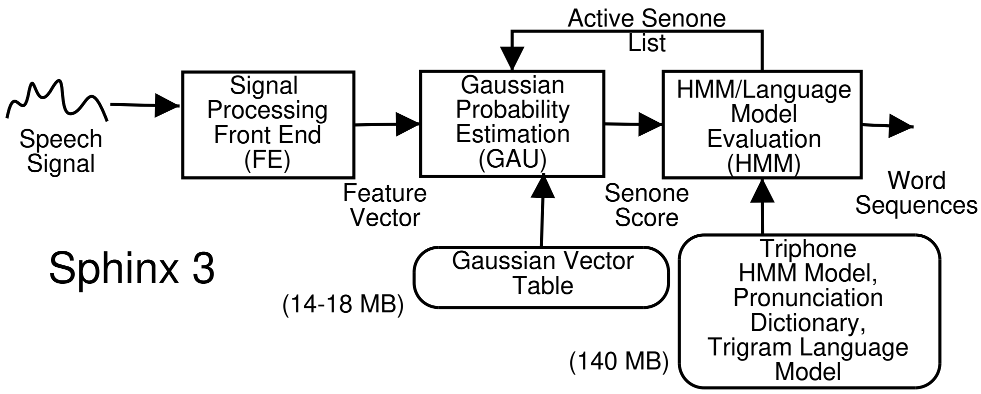

Figure 1: Anatomy of a Speech Recognizer

A more accurate and detailed view is that GAU precomputes Gaussian probabilities for sub-phonetic HMM states (senones). The output of the GAU phase is used during acoustic model evaluation and represents the probability of observing a feature vector in an HMM state. The set of feature vectors that can be observed from a state are assumed to follow a Gaussian distribution. The Gaussian probability is computed as the weighted sum of the Mahanalobis distance of the feature from a set of references used while training the recognizer. The Mahanalobis distance is a statistically significant distance squared metric between two vectors. Given a feature vector $Feat$ and the pair of vectors (M, V) (hereafter called a component) which represent the mean and variance from a reference, GAU spends most of its time computing the quantity:

$$d = \Sigma_{c=1}^{8} FinalWeight_c + FinalScale_c * \sum_{i=1}^{39} (Feat[i] - M[i])^2 * V[i]$$

GAU is a component of several recognizers including Sphinx3, Cambridge University's HTK, ISIP and Sirocco to name a few [6, 7, 14, 13, 4]. For Sphinx3, the length of all three vectors is 39 and each element is an IEEE 754 32-bit floating point number. The Gaussian table contains 49,152 components.

Sphinx uses feedback from the HMM phase to minimize the number of components GAU needs to evaluate. In the worst case, every single component needs to be evaluated for every single frame. A real time recognizer should have the ability to perform 4.9 million component evaluations per second. In practice, the feedback heuristic manages to reduce this number to well under 50%. The Viterbi search algorithm for HMMs is multiplication intensive, but Sphinx as well as many other speech recognizers convert it to an integer addition problem by using fixed point arithmetic in a logarithmic domain. FE and GAU are the only floating point intensive components of Sphinx.

The Sphinx3 code spends less than 1% of its time on front end processing, 57.5% of the time on the Gaussian phase and 41.5% on the HMM phase. While our work has addressed the entire application, the work reported here addresses the optimization and implementation of the dominant Gaussian phase. The contributions include an analysis of the Sphinx3 system, an algorithmic modification which exposes additional parallelism at the cost of increased work, an optimization which drastically reduces bandwidth requirements, and a special-purpose co-processor architecture which improves Sphinx3's performance while simultaneously reducing the energy requirements to the point where real-time, speaker-independent speech recognition is viable on embedded systems in today's technology.

2 Characterization and Optimization of Sphinx 3

To fully characterize the complex behavior of Sphinx, we developed several variants of the original application. In addition to the FE, GAU and HMM phases, Sphinx has a lengthy startup phase and extremely large data structures which could cause high TLB miss rates on embedded platforms with limited TLB reach. To avoid performance characteristics being aliased by startup cost and TLB miss rate, Sphinx 3.2 was modified to support check-pointing and fast restart. For embedded platforms, the check-pointed data structures may be moved to ROM in a physically mapped segment similar to kseg0 in MIPS processors. Results in this paper are based on this low startup cost version of Sphinx referred to as *original*.

Previous studies have not characterized the 3 phases separately [2, 8]. To capture the phase characteristics and to separate optimizations for embedded architectures, we developed a "*phased*" version of Sphinx 3. In *phased*, each of the FE, GAU and HMM phases can be run independently with input and output data redirected to intermediate files. In the rest of this paper *FE, GAU, HMM* refers to the corresponding phase run in isolation while *phased* refers to all three chained sequentially with no feedback. In *Phased*, FE and HMM are identical to *original*, while GAU work is increased by the lack of dynamic feedback from HMM. Breaking this feedback path exposes parallelism in each phase and allows the phases to be pipelined. *GAU OPT* refers to a cache optimized version of the GAU phase alone. *PAR* runs each of the FE, GAU OPT and HMM phases on separate processors. It also uses the same cache optimizations as *GAU OPT*.

We used both simulation and native profiling tools to analyze Sphinx 3. Simulations provide flexibility and a high degree of observability, while profiled execution on a real platform provides realistic performance measures and serves as a way to validate the accuracy of the simulator. The configurations used to analyze Sphinx 3 are:

Native execution: SGI Onyx3, 32 R12K processors at 400 MHz, 32KB 2 way IL1, 32KB 2 way DL1, 8 MB L2 . Software: IRIX 64, MIPS Pro compiler, Perfex, Speedshop.
Simulator (default configuration): SimpleScalar 3.0, out of order CPU model, PISA ISA, 8 KB 2 way IL1, 2 cycle latency, 32 KB 2 way DL1, 4 cycle latency, 2 MB 2 way L2, 20 cycle latency, 228 cycle DRAM latency, L1 line size 64 bytes, L2 line size 128 bytes .

It appeared likely that a multi-GHz processor might be required to operate Sphinx in real time. Parameters like L1 cache hit time, memory access time, floating point latencies etc were measured on a 1.7GHz AMD Athlon processor using the lmbench hardware performance analysis benchmark [9]. Numbers that could not be directly measured were obtained from vendor micro architecture references. Simplescalar was configured to reflect these parameters. Unless mentioned otherwise, the remainder of this paper uses the default configuration.

Native profiling indicates that the *original* Sphinx 3 spends approximately 0.89%, 49.8% and 49.3% of its compute cycles in the FE, GAU and HMM phases respectively. Another recent study found that as high as 70% of another speech recognizers execution time was spent in Gaussian probability computation [8]. In the *phased* version we found that approximately 0.74%, 55.5% and 41.3% of time was spent in FE, GAU and HMM respectively. Since FE is such a small component of the execution time, we ignore it in the rest of this study and concentrate on GAU and HMM.

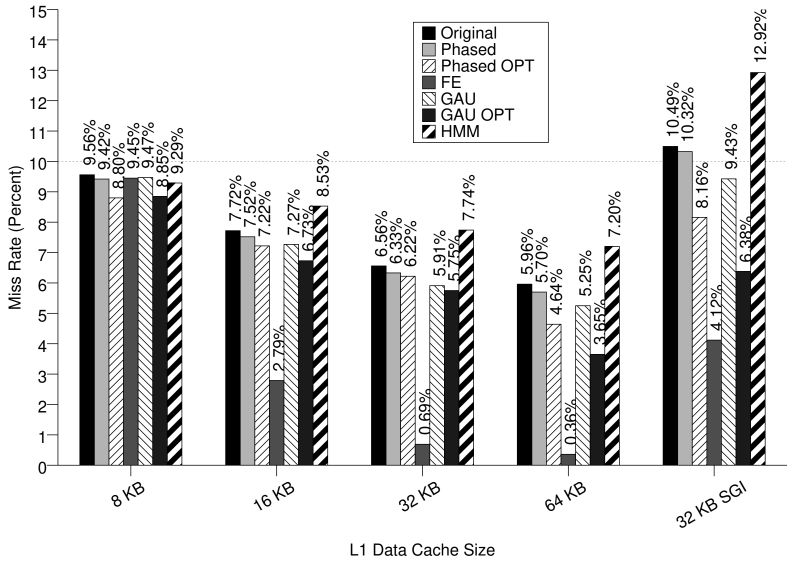

Figure 2: L1 DCache Miss rate

Figures 2 and 3 show the L1 Dcache and L2 cache miss rates for *original*, *phased*, FE,

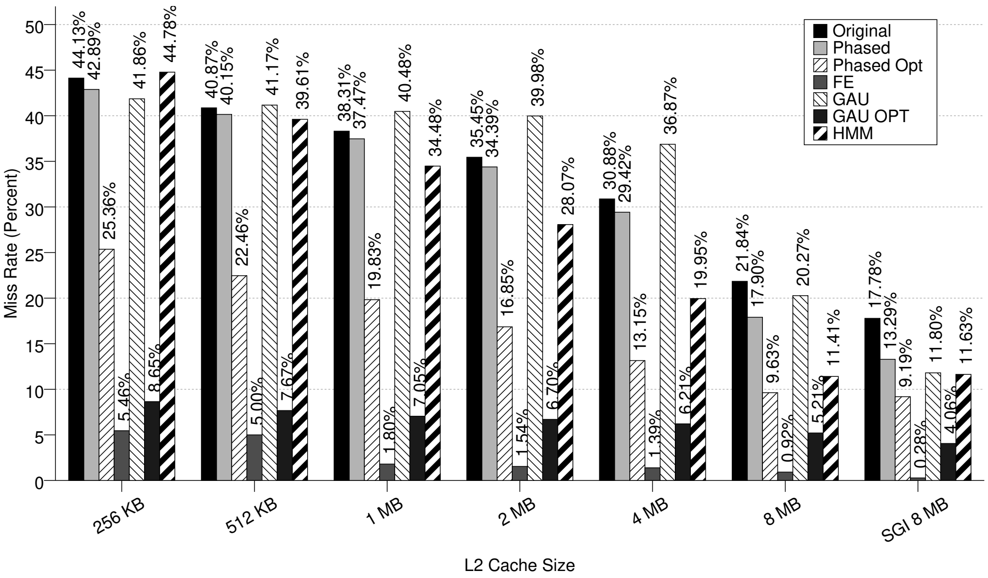

Figure 3: L2 Cache Miss rate

HMM and GAU for a variety of configurations. Since earlier studies showed that larger line sizes benefited Sphinx II, 64 byte L1 and 128 byte L2 cache line sizes were chosen [2]. In addition, the L2 cache experiments assume a 32K L1 Dcache. Both figures assume an 8 KB ICache. Since Sphinx has an extremely low instruction cache miss rate of 0.08% for an 8KB ICache, no other ICache experiments were done. The SGI data provides a reality check since they represent results obtained using hardware performance counters. Though the SGI memory system latency is much lower than that of simulated processors on account of low processor to memory clock ratio, the L2 results are very similar in character to the 8MB simulation results in spite of the effects of out of order execution affected by memory system latency and differences in cache replacement policy. The L1 results are not directly comparable since the R12000 uses a 32 byte L1 line size and suffers from cache pollution caused by abundant DTLB misses.

Figure 4 shows the average bandwidth required to process the workload in real time. This is obtained by dividing the total L2 to memory traffic while Sphinx operates on a speech file by the duration in seconds of the speech signal. The evidence suggests that bandwidth starvation leading to stalls on L2 misses is the reason this application is not able to meet real time requirements. The memory bandwidth required for this application is several times higher than what is available in practice (not theoretical peak) on most architectures. For example, a 16 fold improvement in L2 size from 256 KB (the L2 size of a 1.7 GHz Athlon) to 8 MB (SGI Onyx) produces only a very small decrease in the bandwidth requirement of *GAU*. This phase essentially works in stream mode making 100 sequential passes per

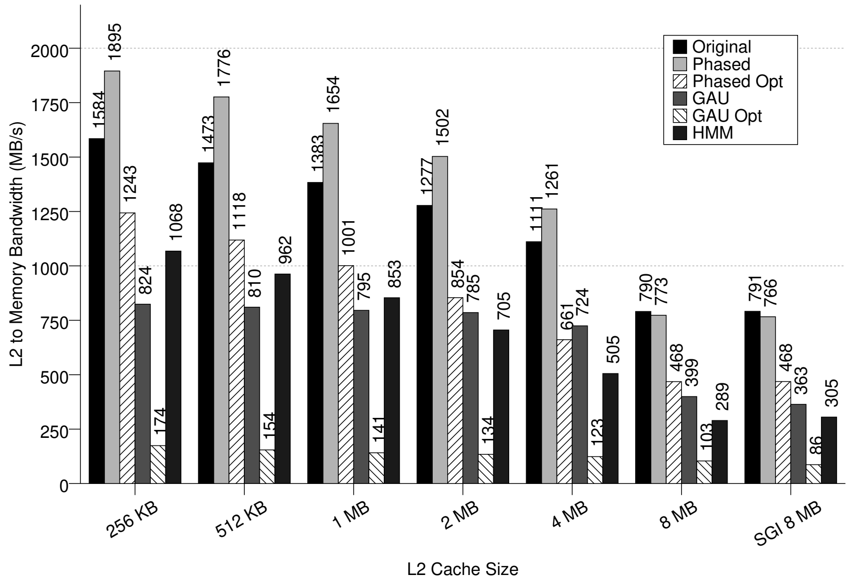

Figure 4: L2 to Memory Bandwidth

second over a 14 MB Gaussian table. The speech signal itself contributes only 16KB/s to the total bandwidth requirements. Some computation saving heuristics in Sphinx also have the beneficial side effect of helping to save bandwidth by not touching blocks that are deemed improbable. Until the L2 size reaches 8 MB, long term reuse of Gaussian table entries in the L2 is infrequent. It should be noted that the bandwidth requirement of *GAU* in isolation is more severe than if it were operating inside *original*, since feedback driven heuristics cannot be applied.

2.1 ILP in Sphinx

Before exploring special-purpose architecture extensions for speech, it is worthwhile to investigate the limits of modern architectures. GAU is a floating point dominant code while HMM is dominated by integer computations. GAU also appears to be easily vectorizable. We performed two simulation studies to explore possibilities for extracting ILP. For GAU, a surplus of integer ALUs were provided and the number of floating point units were varied. Since this algorithm uses an equal number of multiplies and adds, the number of floating point adders and multipliers were increased in equal numbers from 1 to 4, which corresponds to the X axis varying from 2 to 8 FPUs in Figure 5. Two different memory system hierarchies were considered: a reasonable one for a multi GHZ processor () and an aggressive memory system with lower latencies. **Reasonable configuration:** 32KB DL1, 4 cycle latency, 2MB L2, 20 cycle latency, 2 memory ports. **Aggressive configuration:** 32KB DL1, 2 cycle latency, 8MB L2, 20 cycle latency, 4 memory ports.

The *SGI-2+2f* entry describes the measured total IPC on the R12000 which has 2 integer and 2 floating point units. The SGI-2 entry is the measured floating point IPC alone. In the case of GAU, IPC remains low because of the inability of the algorithm to have sufficient memory bandwidth to keep the FPUs active. In the case of the R12000 which can issue two floating point operations per cycle, the IPC for this loop is an underwhelming 0.37. GAU OPT, uncovers opportunities for ILP by virtue of its cache optimizations thereby improving IPC greatly. However, IPC saturates at 1.2 in spite of available function units. A recently published study also indicated IPC in the range of 0.4 to 1.2 for another speech recognizer [8]. Clearly, the architecture and compiler are unable to automatically extract the available ILP which again argues for custom acceleration strategies.

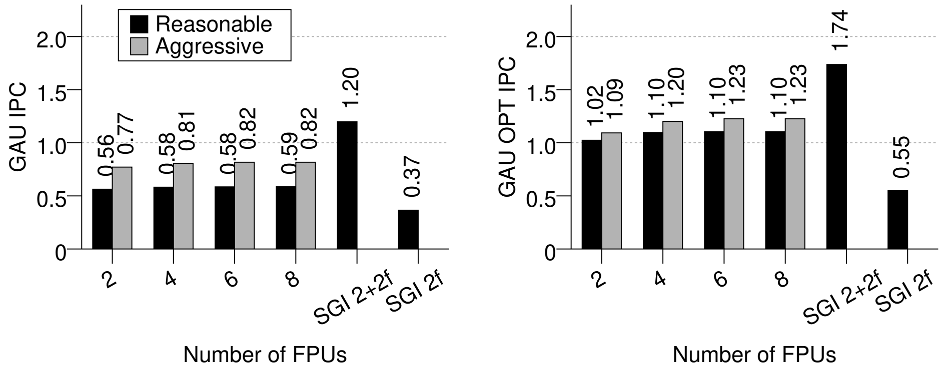

Figure 5: GAU and GAU OPT IPC

Figure 6 shows the corresponding experiment for the HMM phase. In this experiment, the number of integer adders and multipliers in equal numbers varies from 1 to 4. In spite of available execution resources, IPC remains low. It should be noted that in both experiments, the SGI results are indicative of cases where the CPU to memory clock ratio is low. This ratio will undoubtedly increase in the future.

The observations from sections 2, 2.1 have several implications: If speech is an “always on” feature, it could cause significant L2 cache pollution and memory bandwidth degradation to the foreground application. To guarantee real time processing, it might be better to stream data around the L2 rather than pollute it. Since the L2 cache is one of the largest sources of capacitance on the chip, accessing it for stream data also incurs a large power overhead. Low power embedded platforms may not need any L2 cache at all since dramatic increases

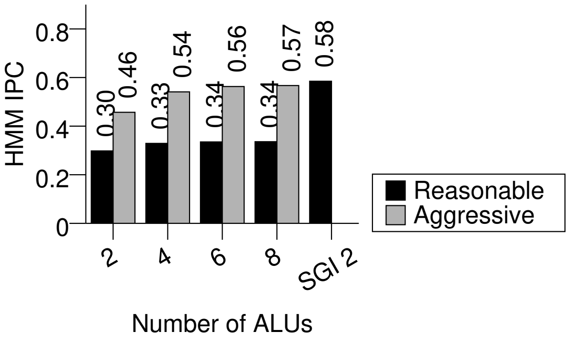

Figure 6: HMM IPC

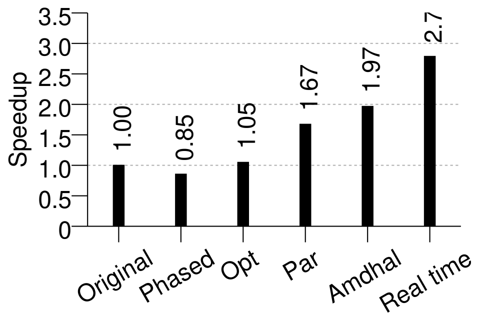

Figure 7: Measured Speedup on R12K

in L2 size are not accompanied by corresponding improvements in DRAM bandwidth or performance. Bandwidth reduction is important for its own sake as well as to reduce power consumption. Bandwidth partitioning so that each phase has independent access to its data set is important.

2.2 Results of Software Optimizations for Sphinx

Cache Optimizations: In the Section 2, GAU was shown to be bandwidth starved. The GAU code in *phased* was instrumented and found to require approximately twice the amount of computation as in *original*. However, Figure 7 shows that *phased* suffers only 0.85 times slow down over *original* on an R12000. Clearly, a large fraction of the excess computation is hidden by memory latency. With processor to memory speed ratios increasing in the future, an out of order processor can hide an even larger amount of compute overhead. The key is to improve the memory system behavior without an unreasonable increase in compute requirements.

To achieve this goal, two transformations were performed on *phased*. First, a blocking optimization similar in spirit to loop tiling is performed which delays the initial speech signal by 100ms or 10 frames. The Gaussian probabilities for all 10 frames are then computed by making a single pass over the Gaussian tables. This effectively reduces the number of passes to 10 per second where *original* would have done 100. The blocking factor is limited to 10 to avoid a perceptible real-time lag at the decoder output. Sphinx allocates the mean and variance vectors used for Gaussian computation described in section 1 separately. Second, every component evaluation consumes one mean and one variance vector. We interleaved corresponding vectors to reduce cache conflicts. Since Sphinx originally allocated each table of vectors separately and each is more than 7 MB, they potentially

```
for(senone = 0; senone < N; senone++)     // Loop 0
for(block=0; block < 10; block++)         // Loop 1
  for(c=0; c < 8; c++)                    // Loop 2
    {
     for(i=0, sum=0.0; i < 39; i++)       // Loop 3
        {
         t = X[block][i] -
              Gautable[senone][c].vector[i].Mean;
         sum = t * t *
                 Gautable[senone][c].vector[i].Var;
         sum = max(sum, MINIMUM_VALUE);
        }
     score[senone][block] += sum *
                  Gautable[senone][c].FinalScale  +
                  Gautable[senone][c].FinalWeight;
    }
```

Figure 8: Cache Optimized Gaussian Algorithm

conflict with each other in the cache. To avoid this, corresponding mean and variance vectors are interleaved and padded with an additional 64 bytes to be exactly 3 L2 cache lines long. This padding strategy consumes bandwidth but simplifies DMA transfers for the co-processor architecture described later. The optimized version appears in Figure 8. Note the interleaving of vectors and a *blocking* loop that is not present in the equation shown in Section 1. More details of how this affects a hardware implementation will be presented in the next section. The optimized version appears in Figures 2, 3, 4 and 7 as the data point GAU OPT.

GAU OPT demonstrates the true streaming nature of GAU. Figure 4 shows that GAU OPT uses a factor of 4.7 to 3.9 less bandwidth than GAU in simulation with a factor of 4.2 improvement obtained on a real machine. This supports our claim that GAU processing can be done without an L2 cache. With a 256 KB L2 cache, the GAU OPT bandwidth is 174 MB/s. We have calculated that with no heuristic, and no L2 cache, GAU OPT can meet its real time requirements with 180 MB/s of main memory bandwidth. This has important implications for the scalability of servers that process speech.

Figures 2 and 3 show dramatic reduction in the cache miss rates in both simulation and native execution. The L2 native execution results are better than simulation results. The large variation in the L1 results is due to the 32 byte L1 line size on the R12000 and also possibly because of an extremely large number of TLB misses. The software miss handler could easily pollute the L1 cache. The important point is that Figure 7 shows that OPT, a version of *phased* with our GAU OPT blocking optimizations achieves a slight speedup over *original* in spite of performing a larger number of computations. In summary, to be able to extract parallelism, the feedback loop was broken which approximately doubled the GAU workload. With cache optimizations (which are not possible with feedback), the loss due to the extra GAU workload is recovered and the exposed parallelism is now open for further optimization.

Parallelization: Based on the percentage of execution time, Amdahl's law predicts a factor of 1.97 speedup if GAU and HMM processing could be entirely overlapped. It is clear that a special-purpose architecture for GAU can have significant speedup, as well as power and scaling benefits. We parallelized Sphinx in order to see if there were any practical impediments to achieving good speedup. We developed, a parallel version of Sphinx, called *PAR*, which runs each of the FE, GAU OPT and HMM phases on separate processors. In effect, this models an SMP version of Sphinx 3 as well as the case where each processor could be replaced by a special-purpose accelerator. As shown in Figure 7, the parallel version achieves a speedup of 1.67 over the original sequential version. A custom accelerator will likely be even better. The HMM phase was further multi-threaded to use 4 processors instead of 1, but the resulting 5 processor version was slower than the 2 processor version due to the high synchronization overhead. Our research shows that HMM processing also benefits from special-purpose acceleration but that work is not reported here.

3 A GAU Acceleration Architecture

The tight structure of the GAU computation lends itself to a high-throughput custom implementation. The key questions are how to achieve area, power and bandwidth efficiency as well as scalability. This section describes how we achieved these goals by a) reducing the floating point precision, b) restructuring the computation, and c) sharing memory bandwidth.

Sphinx designers try hard to eliminate floating point computation wherever possible. GAU and FE are the only floating point dominant computations in Sphinx. An attempt was made to convert GAU to use fixed point integer arithmetic. This was a total failure. The computations require a very high dynamic range which cannot be provided with 32 bit scaled integer arithmetic. Fortunately, the scores of the highly probable states are typically several orders of magnitude higher than those of the less likely ones indicating that a wide range is more important than precision.

Earlier work explored the use of special-purpose floating point formats in Gaussian estimation to save memory bandwidth [11]. Special floating point formats should be almost invisible to the application. This reduces complexity and enables the development of speech models without access to any special hardware. We conducted an empirical search for the precision requirements by creating a custom software floating point emulation library for GAU. The library supports multiplication, addition, MAC, and $(a-b)^2$ operations on IEEE 754 format floating point numbers. The approach was to experimentally reduce mantissa and exponent sizes without changing the output results of the Sphinx3 recognizer. The result is an IEEE 754 compatible 12-bit mantissa and 8-bit exponent format similar to an IEEE 754 number in that, it has a sign-bit, an 8-bit excess 127 exponent and a hidden one-bit in its normalized mantissa. Unlike IEEE 754 which has 23 explicit-bits in the mantissa, we only need 12-bits. Conversion between the reduced precision representation and IEEE 754 is trivial. Though our study was done independently, we subsequently found a previous study which arrived at similar conclusions based on an earlier version of Sphinx [15]. However this study used digit serial multipliers which cannot provide the kind of throughput required for GAU computation. Hence we chose to use fully pipelined reduced precision multipliers instead.

Another key insight is that current high performance microprocessors provide a fused multiply add operation which would benefit GAU. However, GAU also needs an add multiply (subtract-square) operation. There is scope for floating point circuit improvements relying on the nature of $(a-b)^2$ always returning a positive number. Further gains can be obtained both in area, latency, power and magnitude of the numerical error by fusing the operations $(a-b)^2 * c$. This is the approach we have taken.

Figure 9 illustrates the system context for our GAU accelerator. Figure 10 shows the details of the accelerator itself.

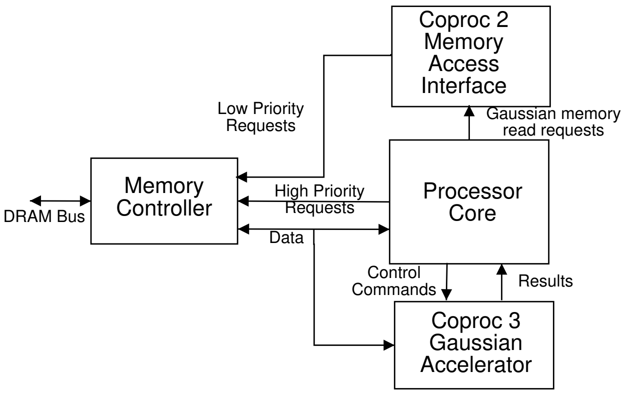

Figure 9: Top Level Organization of Gaussian Estimator

We implement loops 1,2,3 (from the optimized GAU algorithm in Figure 8) in hardware while the outer loop is implemented in software. The max operation can be folded into the denormal floating point number handling section of our floating point adder without additional latency, but empirically it can be discarded without sacrificing recognition accuracy. The organization in Figure 9 is essentially a decoupled access/execute architecture where the outer loop runs on a host processor and instructs a DMA engine to transfer X, Mean and Var vectors into the accelerators input memory. A set of 10 input blocks are transferred into the accelerator memory and retained for the duration of a pass over the entire interleaved Mean/Var table. The Mean/Var memory is double buffered for simultaneous access by the DMA engine and the accelerator. The accelerator sends results to an output queue from where they are read by the host processor using its coprocessor access interface.

The datapath consists of an $(a - b)^2 * c$ floating point unit, followed by an adder that accumulates the sum as well as a fused multiply add $(a * b + c)$ unit that performs the final scaling and accumulates the score. Given that X, Mean, and Var are 39 element vectors, a vector style architecture is suggested. The problem comes in the accumulation step since this operation depends on the sum from the previous cycle, and floating point adders have multi-cycle latencies. For a vector length of N and an addition latency of M, a straight forward implementation takes (N-1) * M cycles. Binary tree reduction (similar to an optimal merge algorithm) is possible, but even then the whole loop cannot be pipelined with unit initiation interval.

This problem is solved using by reordering Loops 1,2,3 to a 2,3,1 order. This calculates an $(X - M)^2 * V$ term for each input block while reading out the mean and variance values just once from the SRAM. Effectively this is an interleaved execution of 10 separate vectors on a single function unit which leaves enough time to do a floating point addition of a partial sum term before the next term arrives for that vector. The cost is an additional 10 internal registers to maintain partial sums. Loops 2,3,1 can now be pipelined with unit initiation interval. In the original algorithm the Mean/Var SRAM is accessed every cycle whereas with the loop interchanged version this 64-bit wide SRAM is accessed only once every 10 cycles. Since SRAM read current is comparable to function unit current in the CMOS technology we use, the loop interchange also contributes significant savings in power consumption.

The *Final Sigma* unit in Figure 10 works in a similar manner except that instead of a floating point adder, it uses a fused multiply add unit. It scales the sum, adds the final weight and accumulates the final score. Due to the interleaved execution this unit also requires 10 intermediate sum registers. This unit has a fairly low utilization since it receives only 8 * 10 inputs every 39 * 10 * 8 cycles. It is useful since it makes it possible for the host processor to read one combined value per block instead of having to do 8 coprocessor reads. To save power this unit is disabled when it is idle. In a multi-channel configuration it is possible to share this unit between multiple channels.

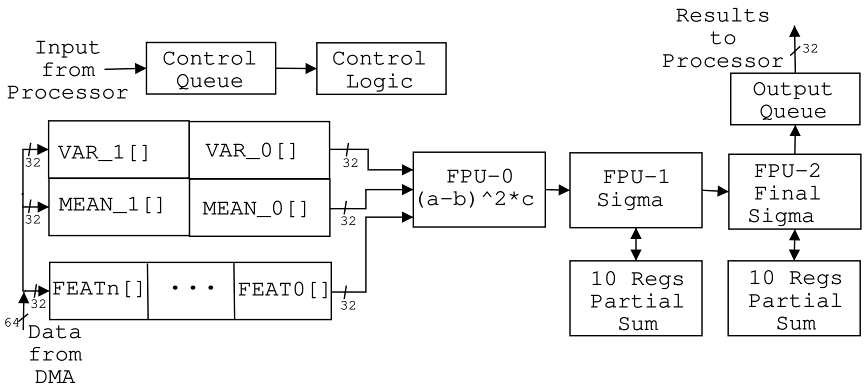

Figure 10: Gaussian Coprocessor

The datapath shown in Figure 10 has been implemented using a datapath description language (Synopsys Module Compiler Language) and is subsequently synthesized for a 0.25μ CMOS process. The control sections were written in Verilog and synthesized using Synopsys Design Compiler. The gate level netlist is then annotated with worst case wire loads calculated using the same wire load model used for synthesis. The netlist is then simulated at the Spice level using Synopsys Nanosim and transistor parameters extracted for the same 0.25μ process by MOSIS. Energy consumption is estimated from the RMS supply current computed by Spice. The unoptimized fully pipelined design can operate above 300 MHz at the nominal voltage of 2.5 volts with unit initiation interval. At this frequency the performance exceeds the real time requirements for GAU, indicating an opportunity to further reduce power. A lower frequency and voltage can be used to further reduce power.

The accelerator was designed and simulated along with a low-power embedded MIPS-like processor that we could modify as needed to support special purpose co-processor accelerators. This control processor is a simple in-order design that uses a blocking L1 DCache and has no L2 cache. To support the equivalent of multiple outstanding loads, it uses the MIPS coprocessor interface to directly submit DMA requests to a low priority queue in the on-chip memory controller. The queue supports 16 outstanding low priority block read requests with block sizes being multiples of 128 bytes. A load request specifies a ROM address and a destination – one of the Feat, Mean or Var SRAMs. The memory controller initiates a queued memory read and transfers the data directly to the requested SRAM index. A more capable out of order processor could initiate the loads directly. Software running on the processor core does the equivalent of the GAU OPT phase. It accumulates 100ms or 10 frames of speech feature vectors (1560 bytes) into the Feat SRAM periodically. This transfer uses the memory controller queue interface. Next, it loads two interleaved Mean/Var vectors from ROM into the corresponding SRAM using the queue interface . A single transfer in this case is 640 bytes. The Mean/Var SRAM is double buffered to hide the memory latency. Initially, the software fills both the buffers. It then queues up a series of vector execute commands to the control logic of the Gaussian accelerator. A single command corresponds to executing the interchanged loops 2,3,1. The processor then proceeds to read results from the output queue of the Gaussian accelerator. When 10 results have been read, it is time to switch to the next Mean/Var vector and refill the used up half of the Mean/Var SRAM. This process continues until the end of the Gaussian ROM is reached. When one cache line of results has been accumulated, they are written to memory where another phase or an i/o interface can read it.

The processor frequency was chosen to provide a capability similar to the well known StrongArm. We also have a cycle accurate simulator which is validated by running it in lock step with the processor's HDL model. The simulator is detailed enough to boot the SGI Linux 2.5 operating system and run user applications in multitasking mode. CAD tool latency estimates are used to time the simulated version of the accelerator. The resulting system accurately models the architecture depicted in Figures 10 and 9. The GAU OPT application for this system is a simple 250 line C program with less than 10 lines of assembly language for the coprocessor interface. Loop unrolling and double buffering were done by hand in C. The application was compiled using MIPS GCC 3.1 and run as a user application under Linux inside the simulator. It was able to process 100ms samples of a single channel in 67.3ms and scale up to 10 channels in real time. The actual data may be seen in the next section. Energy consumption was estimated using Spice simulation.

4 Accelerator Results

Though the Gaussian estimator was designed for Sphinx3 and the MIPS-like embedded processor, the results are widely applicable to other architectures and recognizers. There are several levels at which this system may be integrated into a speech recognition task pipeline similar to *Phased*. For example, an intelligent microphone may be created by using a simple low-power DSP to handle the A/D conversion and FE phase and then use the GAU co-processor for probability estimation. The probability estimates can then be sent to a high-end processor or custom accelerator that does language model computation thereby hiding more than 50% of the compute effort required for speech recognition. On desktop systems, the Gaussian accelerator may be part of a sound card or the Gaussian accelerator may be directly attached to the main processor. On commercial voice servers, the Gaussian estimator may be directly built into the line cards that interface to the telephone network thereby freeing up server resources for language model and application processing. This also has important implications for server scalability described in the next section.

The HUB4 speech model used in this study has 49,152 interleaved and padded Mean/Var components each occupying 3 L2 cache lines of 128 bytes or a total of 384 bytes/component. Thus the total size of the Gaussian table is 18MB. Sphinx processes this table 100 times every second, but uses some heuristics to cut down the bandwidth requirement. To guarantee real time processing, we can do brute force evaluation using the Gaussian accelerator at low-power. Because of our blocking optimization (GAU OPT), we need to process the data only 10 times per second with a peak bandwidth of 180 MB/s which can be further reduced by applying the sub-vector quantization (non-feedback) heuristics in Sphinx. Not only does our design bring the bandwidth requirements to limits possible on embedded systems, it also drastically improves the power consumption. On a 400 MHz Intel XScale (StrongARM) development system where the processor itself consumes less than 1 W, we measured a peak memory bandwidth of 64MB/s which consumes an additional 0.47 W. The factor of 4 or more bandwidth savings is significant for the embedded space since it indicates that a 52-watt server can be replaced by a 1-watt embedded processor.

In addition to power advantages, our design is also scalable. Figure 11 shows that our system can be scaled to process up to ten independent speech channels in real-time. The main limitation is our in-order processor with its simple blocking cache mode. The *Final Sigma* stage enables the design to scale even with blocking caches due to the removal of destructive interference between the cache and the DMA engine. For embedded designs the power required to support out of order execution may be excessive but such an organization is likely in a server. One channel of speech feature vectors contributes about 16 KB/s to the memory bandwidth. The outgoing probabilities consume 2.3 MB/s.

By setting a threshold on acceptable Gaussian scores and selectively sending out the scores, this can be significantly reduced. The dominant bandwidth component is still the Gaussian table. We can add additional *Feat* SRAMs and Gaussian accelerator data paths that share the same Var/Mean SRAMs since the Gaussian tables are common for all channels, thereby reusing the same 180 MB/s vector stream for a large number of channels. With a higher frequency implementation of the Gaussian datapath, multiple channels can also be multiplexed on the same datapath. In a server, the Gaussian estimation of several channels can be delegated to a line card which operates out of its own 18 MB Gaussian ROM. The partitioning of bandwidth, a 50% reduction in server workload per channel as well as reduced cache pollution leads to improved server scalability.

Finally, we compared the Spice results from our fully synthesized coprocessor architecture with an actual 2.4 GHz Pentium4 system that was modified to allow accurate measurement of processor power. Without considering the power consumed by main memory, the GAU accelerator consumed 1.8 watts while the Pentium4 consumed 52.3 watts during Mahanalobis distance calculation representing an improvement of 29 fold. The performance of the Pentium4 system exceeded real-time demands by a factor of 1.6 while the coprocessor

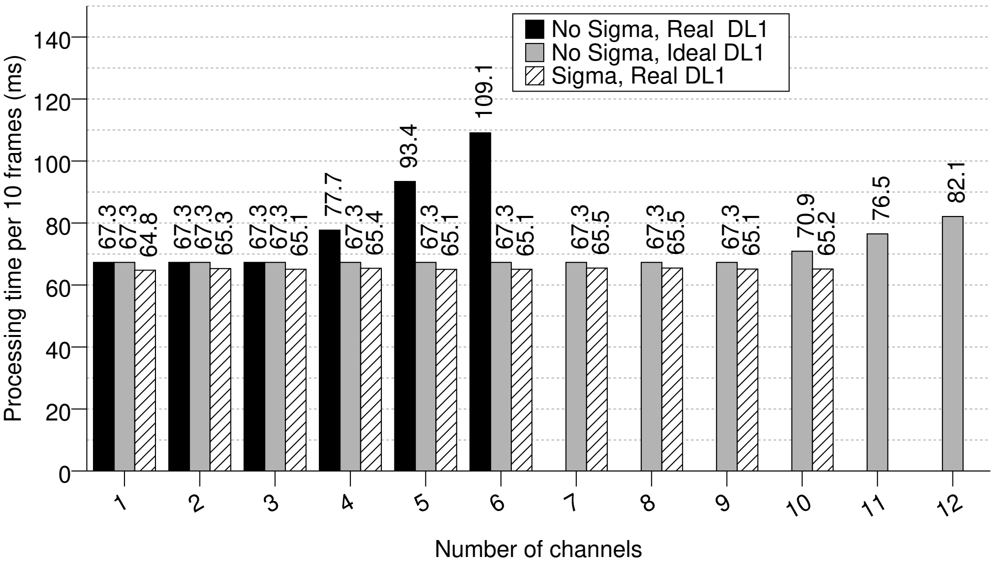

Figure 11: Channel Scaling

approach exceeded real-time by 1.55. However the Pentium4 is implemented in a highly tuned 0.13μ process whereas the GAU accelerator was designed for a generally available TSMC 0.25μ process. When normalizing for process differences, the advantage of the GAU coprocessor approach increases significantly. After normalizing for the process, the co-processor's throughput is 187% higher than the Pentium4, while consuming a whopping 271 times less energy. However it is important to note that energy consumption vs. performance are common design trade-offs. A more valid comparison is the energy-delay product. The GAU co-processor improves upon the energy-delay product of the Pentium4 processor by a factor of 507.

However the processor is only part of any system. Main memory is an important consideration as well. When the memory is included the GAU co-processor approach improves upon the Pentium4's energy delay product by a factor of 196, has an energy advantage of a factor of 104, and the throughput performance stays the same as the processor-only results.

5 Related Work

Most speech recognition research has targeted recognition accuracy [5, 4]. Performance issues have been secondary and power efficiency has largely been ignored. Ravishankar improved Sphinx performance by reducing accuracy and subsequently recovering it in a less computationally active phase and developed a multi-processor version of an older version of Sphinx [10]. However, details of this work are currently unavailable. Agram provided a detailed analysis of Sphinx 2 and compared this analysis with SPEC benchmarks [2]. Pihl designed a 0.8μ custom coprocessor to accelerate Gaussian probability generation for an HMM based recognizer. However Pihl's work proposed a specialized arithmetic format rather than the IEEE 754 compatible version described here. Furthermore the number of Gaussian components need to be processed per second has escalated from 40,000 in the case of Pihl's coprocessor to 4.9 million for our accelerator during the last 7 years and this trend is likely to continue as the search for increased accuracy proceeds. Pihl's work did not address scalability which is a central theme for this research. Tong showed an example of reduced precision digit serial multiplication for Sphinx [15]. Anatharaman showed a custom multiprocessor architecture for improving the Viterbi beam search component of a predecessor of Sphinx [3]. Application acceleration using custom coprocessors has been in use for decades, however current researchers are exploiting this theme for reducing power consumption. Piperench is one such approach which exploits virtualized hardware, and run-time reconfiguration [12]. Pleiades is a reconfigurable DSP architecture that uses half the power of an Intel StrongARM for FFT calculations [1].

6 Conclusions

Sphinx3 has been analyzed to show that real-time processing is problematic due to cache pollution on high-end general-purpose machines, and even more problematic due to both power and performance concerns for low-end embedded systems. Optimizations were then presented and analyzed to expose parallelism and substantially reduce the bandwidth requirements for real-time recognizers. A custom accelerator for the dominant Gaussian phase was then described and analyzed. The accelerator takes advantage of the low precision floating point requirements of Sphinx3 as well as creating a custom function unit for calculating Gaussian probabilities which is the dominant component of the Gaussian phase of Sphinx3. The accelerator has been synthesized for a .25u CMOS process and shown to improve on the process normalized performance of a Pentium4 system by a factor of 2, while simultaneously improving on the energy consumption by 2 orders of magnitude. Other work, not reported here, shows similar results for other phases of the speech recognition process. This is strong evidence that by incorporating a small amount of custom acceleration hardware, it is possible to perform real-time Sphinx3 speech recognition for the HUB4 language model on an embedded processor implemented in current technology.

References

[1] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, and J. M. Rabaey. Evaluation of a low-power reconfigurable DSP architecture. In *IPPS/SPDP Workshops*, pages 55–60, 1998.

[2] K. Agaram, S. W. Keckler, and D. Burger. A characterization of speech recognition on modern computer systems. In *Proceedings of the 4th IEEE Workshop on Workload Characterization*, Dec. 2001.

[3] T. Ananthamaran and R. Bisiani. A hardware accelerator for speech recognition algorithms. In *Proceeedings of the 13th International Symposium on Computer Architecture*, June 1986.

[4] R. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and V. Zue. *Survey of the State of the Art in Human Language Technology*. Cambridge University Press, 1995.

[5] J. G. F. David Pallett and M. A. Przybocki. 1996 preliminary broadcast news benchmark tests. In *Proceedings of the 1997 DARPA Speech Recognition Workshop*, Feb. 1997.

[6] T. Hain, P. Woodland, G. Evermann, and D. Povey. The cu-htk march 2000 hub5e transcription system. 2000.

[7] 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.

[8] C. Lai, S.-L. Lu, and Q. Zhao. Performance analysis of speech recognition software. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2002.

[9] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[10] R. Mosur. *Efficient Algorithms for Speech Recognition*. PhD thesis, Carnegie Mellon University, May 1996. CMU-CS-96-143.

[11] J. Pihl, T. Svendsen, and M. H. Johnsen. A VLSI Implementation of Pdf Computations in HMM Based Speech Recognition. In *Proceedings of the IEEE Region Ten Conference on Digital Signal Processing Applications (TENCON'96)*, Nov. 1996.

[12] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor. Piperench: a virtualized programmable datapath in 0.18 micron technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 63–66, 2002.

[13] M. Seltzer. Sphinx iii signal processing front end specification. http://perso.enst.fr/s̃irocco/ .

[14] S. Srivastava. Fast gaussian evaluations in large vocabulary continuous speech recognition. M.S. Thesis, Department of Electrical and Computer Engineering, Mississippi State University.

[15] Y. F. Tong, R. Rutenbar, and D. Nagle. Minimizing floating-point power dissipation via bit-width reduction. In *Proceedings of the 1998 International Symposium on Computer Architecture Power Driven Microarchitecture Workshop*, 1998.

[16] S. Young. Large vocabulary continuous speech recognition: A review. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 3–28, Dec. 1995.