

Exploiting Coarse-Grained Parallelism to Accelerate Protein Motif Finding with a Network Processor

Ben Wun, Jeremy Buhler, and Patrick Crowley
Department of Computer Science and Engineering
Washington University in St. Louis
{bw6,jbuhler,pcrowley}@cse.wustl.edu

Abstract

While general-purpose processors have only recently employed chip multiprocessor (CMP) architectures, network processors (NPs) have used heterogeneous multi-core architectures since the late 1990s. NPs differ qualitatively from workstation and server CMPs in that they replicate many simple, highly efficient processor cores on a chip, rather than a small number of sophisticated superscalar CPUs. In this paper, we compare the performance of one such NP, the Intel IXP 2850, to that of the Intel Pentium 4 when executing a scientific computing workload with a high degree of thread-level parallelism. Our target program, HMMer, is a bioinformatics tool that identifies conserved motifs in protein sequences. HMMer represents motifs as hidden Markov models (HMMs) and spends most of its time executing the well-known Viterbi algorithm to align proteins to these models. Our observations of HMMer on the IXP are therefore relevant to computations in many other domains that rely on the Viterbi algorithm. We show that the IXP achieves a speedup of 1.82 over the Pentium, despite the Pentium's 1.85x faster clock. Moreover, we argue that next-generation IXP NPs will likely provide a 10-20x speedup for our workload over the IXP 2850, in contrast to 5-10x speedup expected from a next-generation Pentium-based CMP.

1. Introduction

The oft-predicted shift among general-purpose processors (GPPs) away from superscalar organizations of increasing sophistication towards chip multiprocessors (CMPs) appears imminent, with all major desktop processor vendors planning to release dual- or quad-core processors in the near future. GPPs are beginning to adopt CMP organizations mainly because their designers can no longer achieve satisfactory performance improvements by increasing clock frequencies and cache sizes. In contrast, commodity network processors (NPs) have used CMP organizations since the late 1990s to exploit packet-level parallelism in networking workloads. This shift toward CMP in general-purpose processor organization invites comparisons between these processors and NPs.

The first generation of general-purpose CMPs is expected to employ a small number of sophisticated, superscalar CPU cores; by contrast, NPs contain many, much simpler single-issue cores. Desktop and server processors focus on maximizing instruction-level parallelism (ILP) and minimizing latency to memory, while NPs are designed to exploit coarse-grained parallelism and maximize throughput. NPs are designed to maximize performance and efficiency on packet processing workloads; however, we believe that many other workloads, in particular tasks drawn from scientific computing, are better suited to NP-style CMPs than to CMPs based on superscalar cores.

In this work, we study a representative scientific workload drawn from bioinformatics: the HMMer program [2] for protein motif finding. HMMer compares protein sequences to a database of *motifs* – sequences known to occur, with some variation, in a large family of other proteins. These motifs are represented as hidden Markov models (HMMs) [14], which allows HMMer to search for them in a protein using well-developed mathematical machinery for parsing discrete sequences with an HMM. Because HMMer works on a large database of motifs, each of which can be compared separately to a target protein, its computation can benefit greatly from systems with substantial coarse-grained parallelism. This computation is therefore a natural fit to network processor-style CMPs.

We have implemented *JackHMMer*, a version of HMMer that runs on an Intel IXP 2850 network processor. The IXP implements the Viterbi algorithm for HMMs [13], which is the core component of HMMer's search algorithm. This paper recounts our experience implementing JackHMMer, quantifies its performance gain relative to both the original HMMer and a hand-optimized version on a hyper-threaded Pentium 4, and draws lessons from our experience about how future NPs may be designed in order to better accelerate similar non-networking workloads.

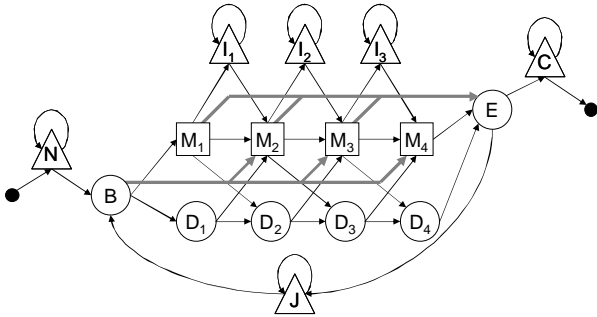


Figure 1: Example of hidden Markov model used by HMMer search tool, with motif length $m = 4$. Start and end states are represented by black dots. Square states emit amino acids in the motif; triangular states emit non-motif amino acids, while circular states are non-emitting. Any path from the start state to the end state of the model generates a protein sequence.

The remainder of the paper is organized as follows. Section 2 provides background on both HMMer and the IXP architecture. Section 3 examines the behavior of HMMer on a superscalar processor. Section 4 describes the implementation of JackHMMer and the techniques used to maximize performance on the IXP. Section 5 compares the performance of JackHMMer to that of a hand-optimized HMMer on the P4 architecture, while Section 6 suggests likely improvements to both architectures and extrapolates the expected speedup if our assumptions prove correct. Section 7 describes related work. The paper concludes in Section 8.

2. Background

In this section, we review both the problem domain and the target architecture for JackHMMer. We first give a detailed account of the HMMer application, its purpose, and its core search computation. We then describe the essential features of the Intel IXP architecture that we use to accelerate this computation.

2.1 Protein Motifs and HMMer

Proteins perform most metabolic and regulatory tasks in living cells. Families of evolutionarily related proteins exhibit *conservation* of a common amino acid sequence along part or all of their lengths. To help identify the function of an unknown protein, biologists look for strings of amino acids in its sequence that resemble the sequences of proteins with known functions. When a common sequence appears in multiple proteins, it is called a *motif*.

Because exact preservation of a motif’s sequence is rarely necessary to maintain its biological function, a motif may be encoded by slightly different sequences of amino acids in different proteins. Amino acids in the motif may change or may be deleted, and irrelevant “non-

motif” sequences may be inserted in the middle of it. All these forms of variation must be considered when seeking a motif in a protein sequence. Furthermore, several distinct copies of one motif may appear within a single protein.

To summarize the observed variability in a motif, HMMer describes it probabilistically using a *hidden Markov model (HMM)*. An HMM is a finite state diagram in which each directed edge from state q_i to state q_j is assigned a *transition probability* $p(q_j | q_i)$. If q_j is an *emitting state*, passing through it emits one symbol (i.e. one amino acid); each possible symbol α has an *emission probability* $e(\alpha | q_j)$. To generate a protein sequence of length n from an HMM, one begins in its initial state q_0 and traces a path that passes through n emitting states, and possibly some non-emitting states. The probability of the sequence is the product of all transition probabilities on the path, times the emission probabilities of all amino acids given the states from which they were emitted.

The structure of the HMM used by HMMer is shown in Figure 1. A motif of length m is comprised of m “match states”, $M_1 \dots M_m$, where M_k emits the amino acid at the motif’s k th sequence position. A parallel sequence of non-emitting “deletion states” states $D_1 \dots D_m$ allow any substring of the motif to be skipped, while another set of “insertion states” $I_1 \dots I_{m-1}$ allow sequence to be emitted between any two motif positions. The non-emitting states B and E anchor the motif’s endpoints. Finally, states N , C , and J respectively emit non-motif sequences before, after, and between two copies of the motif. In this way, the model generates an entire protein containing one or more copies of the motif. HMMer infers suitable transition and emission probabilities to describe a motif from a collection of example sequences for that motif.

2.1.1 Finding Motifs with the Viterbi Algorithm

To determine whether a protein sequence s of length n contains a motif matching the model M , HMMer calculates the probability that s is emitted by a series of states that form a path connecting the start and end states of M . Any path through the HMM with n emitting states can generate the emitted sequence s ; following the maximum likelihood principle [3], HMMer finds and evaluates only a single, most probable path. If this path has a high enough probability relative to the chance that s was generated from a null model containing no motif, then s is considered to “hit” M , and the path indicates which amino acid in the protein (if any) corresponds to each motif position.

HMMer uses the Viterbi algorithm [13], a well-known dynamic programming method, to compute the most probable path through an HMM M for a sequence s . Let $P(q, j)$ be the highest probability for any path through M

from state q_0 to state q that emits the string $s[1..j]$. If state q is non-emitting, we have

$$P(q, j) = \max_{q' \in M} P(q', j) p(q | q').$$

If state q is emitting, we have

$$P(q, j) = e(s[j] | q) \times \max_{q' \in M} P(q', j-1) p(q | q').$$

If q_e is the unique final state of M , then a most probable path for s has probability $P(q_e, s)$. This probability can be computed in time $\Theta(n/M)$, where $|M|$ is the total number of states in the model. Given space $\Theta(n/M)$ to store the full matrix of intermediate probabilities $P(q, j)$, one can recover a most probable path by tracing back the transitions chosen by the algorithm, starting with the end state q_e and continuing until the start state q_0 is reached.

HMMer's implementation of the Viterbi algorithm takes the form of a doubly nested loop. The outer loop iterates over each amino acid $s[j]$ of the sequence s , while the inner loop iterates over the states of the model M . This loop ordering is dictated by the data dependencies in the model of Figure 1. In the HMMs typically used by HMMer, m , the motif length, is on the order of tens to hundreds; hence, nearly the entire inner loop is spent calculating probabilities for the M -, I -, and D -states, with negligible time spent on the other states. In both our implementation, described in Section 4, and the original HMMer software, the inner loop is organized as a series of m iterations, each of which processes the states M_k , I_k , and D_k for one motif position k .

Two further implementation details are crucial to the feasibility of our implementation. First, HMMer carries out all computations in the log domain, so that the products in the Viterbi recurrence can be replaced by sums. Moreover, the log probabilities are scaled to integer scores, allowing the entire computation to be done in fixed-point arithmetic. Hence, the computation requires neither floating-point nor fast multiplication, which are lacking on the IXP.

2.1.2 Significance of the Viterbi Algorithm

HMMer's basic computation is as follows: given a protein sequence s and a database of motif models $M_1 \dots M_z$, compute the score of a most probable path for s through each M_j using the Viterbi algorithm. Subsequent computation identifies paths scoring highly enough to report, but this work is negligible compared to that required to search a large motif database. The time needed to read the motif database can also be made nearly negligible by storing each model in its binary, in-memory representation. Hence, nearly 100% of useful compute time in HMMer is spent in the Viterbi algorithm.

Although HMMer runs quickly on one protein and just a few models, its cost rapidly mounts in high-throughput bioinformatics use. Publicly available motif databases such as Pfam-A and SCOP contain around 10^4 models, all of which must be compared to a protein of interest to identify its component motifs. Typical applications of HMMer include motif identification in all of an organism's proteins (5×10^3 for bacteria to 2×10^4 for human), or, if discrete proteins have not been identified, in a translation of organism's complete genomic DNA (10^7 amino acids, equivalent to roughly 3×10^4 average-sized proteins, for even a small bacterium).

A modern superscalar CPU requires between 10^{-3} and 10^{-2} seconds to compare a typical model to a typical protein, leading to times on the order of 1-10 CPU-days for high-throughput HMMer computations. Moreover, databases of automatically, rather than manually, curated motif models, such as Pfam-B, can be an order of magnitude larger than those mentioned above. The biological community's desire to accelerate these computations can be seen in the development of massively parallel HMMer implementations for computing clusters [28], of FPGA-based HMMer accelerators [25] and of heuristics [29] to process more models per second at some cost to sensitivity. Developing fast architectures for HMMer can therefore significantly benefit bioinformatics.

The utility of accelerating the Viterbi algorithm extends well beyond its application to HMMer. Searches using hidden Markov models are useful for many applications that involve recognizing complex patterns in sequential data. A classic engineering application of HMMs is in speech recognition [13], where spoken words must be reconstructed from a noisy time series of individual phonemes. Related applications include text recognition [19], image processing and computer vision [19], and time-series analysis of scientific and economic data sets [23]. More recently, HMMs and related probabilistic techniques have been used to recognize patterns of behavior in network traffic, particularly for intrusion detection [24]. All of these applications involve comparing a large volume of information to a collection of HMMs using the Viterbi algorithm or its close relatives, so all are potentially amenable to parallelization using an implementation like that described in this work.

2.2 Packet Processing and the Intel IXP

The Intel IXP family of NPs [8] is designed to implement packet processing tasks within packet-based networking and communications equipment, such as router line cards, cellular phone base stations, wired and wireless access points, and security devices. Each IXP product targets a different link speed; the IXP 2850 supports line-rate packet processing at up to 10 Gbps. The

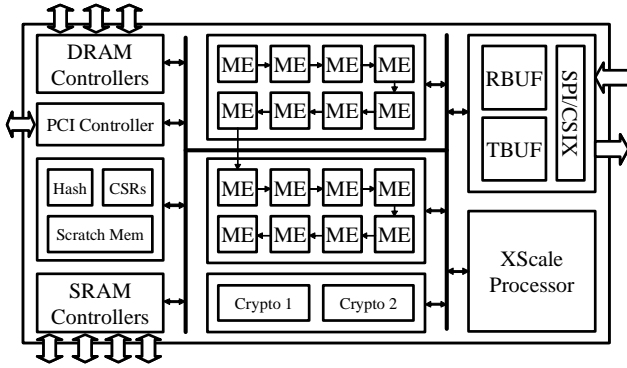


Figure 2: Organization of the IXP 2850 NP.

IXP, like other NPs, uses multiple processors, threads, and memory channels to increase per-packet computation while hiding memory latency.

Figure 2 shows the major components and top-level organization of the IXP 2850. It consists of a number of units, each linked to a shared interconnect. While it is not apparent in the figure, the interconnect is made up of multiple uni-directional command and data buses, each unit having a connection to each bus.

The IXP features a variety of resources, including multiple memories and both programmable and fixed-function units. The microengine (ME) clusters are the central resource. Each ME is a 32-bit, 6-stage pipelined processor. The MEs implement a small, RISC-like ISA tailored to packet processing. The XScale processor is a standard ARM-compatible processor that implements control and management functions on top of Linux or a commercial real-time OS. Other units provide critical functions or resources in hardware, including a configurable hash unit and 16KB of on-chip scratch memory. Each IXP integrates DRAM and SRAM controllers on chip. The IXP 2850 includes three Rambus RDRAM and four QDR SRAM controllers and channels for bulk and latency-sensitive data storage, respectively. Most units on the IXP 2850 are clocked at 1.4 GHz, with the exception of the XScale, interconnect, and memory controllers, which are clocked at 700 MHz or less.

Figure 3 shows a high-level view of the IXP ME organization. Each IXP ME provides hardware support for 8 hardware thread contexts, including register storage, multithreading ISA extensions, and a thread arbiter. Each ME has its own local data and instruction storage, both implemented as SRAMs. An ME communicates asynchronously with other units via I/O commands and transfer registers. A DRAM read, for example, is carried out by sending a read operation to the DRAM controller (via the Command Outlet FIFO) that specifies the desired address as well as the target incoming transfer registers to which the data should be delivered. Hardware signals are

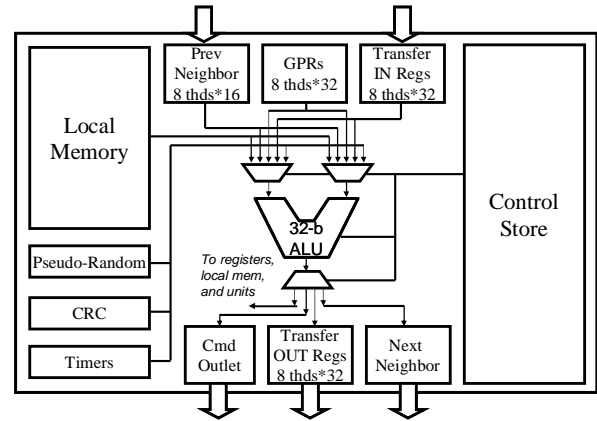


Figure 3: Organization of the IXP Micro-Engine.

specified in the ISA and are asserted when requested operations have completed. This message-passing style and the use of hardware signals allow ME software to initiate multiple external requests, without blocking so long as subsequent computation does not depend on the completion of these requests. This asynchronous memory interaction, which is a unique feature of the IXP ME ISA, is exploited in Section 4.4 to pipeline the Viterbi computation.

3. HMMer on a Superscalar Processor

As described in Section 2, HMMer execution is dominated by the Viterbi computation. Indeed, profiling HMMer using gprof [16] shows that 82.9% of execution time is spent in the function that implements the Viterbi algorithm; almost all of the remainder is spent in preprocessing of the model database, which can be moved off-line relative to the search. To better understand this crucial function, we used Intel’s Vtune [12] program to empirically characterize its execution on the Pentium 4.

The version of HMMer that we characterize here is not the standard distribution but a modified, highly optimized version that is described in section 5.2. For our experiments, we used an input protein of length $n = 544$ and a representative sample of motif models from the Pfam_ls database [4]. Using a different set of input proteins and motif models would influence runtime but (as we later show) would not materially change the program’s behavior.

3.1 Instruction and Data Locality

The Viterbi computation is a relatively small function, requiring only several hundred lines of C code. Consequently, instruction cache performance should be good. This is confirmed by Vtune results which show a trace cache miss rate of 0.001%. Data locality is also good, as there is a first level cache miss rate of 7% and a second level cache miss rate of 0.7%. These results show

that the 512 KB cache on our P4 can service the vast majority of memory requests.

3.2 Instruction Mix

Most instructions executed by HMMer are computation and logic instructions, which account for 58% of all dynamic instructions. Memory operations are the second most frequent class, with load and store operations representing 16.2% and 16.6%, respectively. Control flow instructions represent 9.2% of all operations. With its high cache hit rates, and a branch misprediction rate of only 0.3%, HMMer on the P4 is limited principally by the amount of ILP the processor can exploit.

3.3 Prospects for ILP

We used the SimpleScalar toolkit, version 3 [6] to investigate the prospects for ILP within HMMer by observing how instructions issued per cycle (IPC) changed as a function of issue width. In the simulations used to gather this information, sufficiently large caches, branch predictors, and reorder buffers were used at each issue width to avoid structural hazards. IPC peaks at around 3.6, but only when a very aggressive superscalar width of 128 is used. For narrower superscalar processors, an IPC of between 1 and 2 is more realistic. This is confirmed by Vtune, which shows the actual IPC on the hyper-threaded P4 to be 1.87 for 1 thread and 2.16 for 2 threads (1.08 per thread).

The available ILP in HMMer’s Viterbi computation is limited for two reasons. First, the core inner loop includes a number of loop-carried dependencies that limit opportunities for parallel instruction issue. Moreover, the inner loop contains around 50 instructions, many of which have data dependencies between them. These dependencies leave little room for out-of-order scheduling either within or between loops. The low level of available ILP in HMMer limits the potential for acceleration by complex out-of-order execution schemes.

3.4 Prospects for TLP

While parallelism between instructions in a single Viterbi calculation appears limited, there is no restriction on the number of distinct Viterbi calculations that can be carried out on distinct models simultaneously. In fact, because HMMer is frequently used to check one or several sequences against a large model database, there is an ample supply of thread-level parallelism if different models are processed in different threads. Indeed, the HMMer distribution includes a version of the program that uses the pthreads library [7] to exploit this parallelism on SMP architectures. Running our hand-optimized version of HMMer (described in Section 5.2) using 2 threads on a hyper-threaded Pentium processor increased performance by 14.8%. All comparisons to the

IXP in this work reference this dual-threaded version on the hyper-threaded P4.

To summarize, our observations of HMMer using Vtune and SimpleScalar suggest that it has a small cache footprint, easily predicted branches, and limited instruction-level parallelism, but massive coarse-grained parallelism. These characteristics tend to defeat efforts to accelerate the computation using the dominant design techniques for improving superscalar performance, but they are much more attractive for deployment on an array of small, relatively simple processors. JackHMMer therefore organizes its computation to maximally exploit HMMer’s coarse-grained parallelism on the IXP architecture.

4. JackHMMer: HMMer on the IXP

In this section, we describe the design and implementation of JackHMMer. The XScale control processor is responsible for dispatching jobs to the MEs, which perform the Viterbi calculation. Because current IXP implementations provide relatively little memory close to the MEs, a principal design challenge was to keep all MEs busy while retaining models and intermediate data in a shared, relatively high-latency memory. We addressed this challenge by minimizing the number of distinct commands processed by the memory controllers and by pipelining the Viterbi algorithm to overlap memory latency with computation.

4.1 Application Partition

JackHMMer works with a single input protein sequence s and a database of HMMs. The database is divided into *Viterbi packets* (M, θ) , one packet for each individual model in the database. Viterbi packets are created offline (for both the IXP and P4 versions) by laying out M in a binary format and computing a corresponding score threshold θ , such that a protein is considered to “hit” a model M only if its score against M is at least θ . At runtime, the XScale distributes these packets of work to the MEs, which compare each M to the sequence s and identifies those models whose score against s is high enough to trigger a hit. Individual Viterbi packets can be processed independently, so each ME acts as an independent worker that asynchronously accepts packets and returns results to the XScale.

For each job that triggers a report, a most probable path through the model must be reconstructed. Reconstruction of this path within the IXP is feasible but is deferred to future work, and it represents a small amount of the work (1% or less) in the full HMMer implementation. To make all comparisons fair, the results reported in this paper exclude the cost of reconstruction for both the IXP and P4 versions of HMMer.

4.2 Memory Requirements and Layout

In our implementation, each of the IXP 2850’s sixteen MEs can independently and asynchronously accept units of work from the XScale. However, the storage needed to complete each unit substantially exceeds the MEs’ local memory capacities, requiring the use of shared SRAM and DRAM memories.

To complete a Viterbi packet (M, θ) , a ME needs the model M and the input sequence s , plus storage for the intermediate values $P(q, j)$. A sequence of n amino acids occupies n bytes of storage, while the model for a motif of length m occupies roughly $104m$ bytes. If the IXP does not retain the full matrix of intermediate probabilities to compute the most probable path, its intermediate storage requirements are a further $3m+4$ 32-bit integers. The protein and motif lengths m and n are typically in the range 200-400, though longer examples of each can be found. Hence, both model and intermediate storage are typically too large to fit within the 2400 bytes of memory local to each ME in the IXP 2850.

The data used in the Viterbi computation is distributed across multiple memories attached to the IXP. This distribution splits the memory traffic of the computation among the IXP’s different memory controllers and data buses, thereby substantially increasing total available bandwidth. We place the models and intermediate storage in SRAM and DRAM respectively, principally because DRAM is the only memory large enough to hold intermediate results for all MEs if we choose to retain the full matrix. The common sequence s used with all Viterbi packets is accessed much less frequently than either model or intermediate data and so can be placed almost anywhere with minimal performance impact; currently, s is placed in scratch RAM attached to the IXP’s hash unit.

We lay out the model parameters and intermediate values $P(q, j)$ in memory so as to most efficiently support the loop structure described in Section 2.1.2. Each iteration of the inner loop processes states M_k , I_k , and D_k for one motif position k . We therefore pack together the model transition probabilities associated with these three states for one value of k into a contiguous block of memory that can be retrieved with a single multiword SRAM read operation. Similarly, we pack $P(M_k, j)$, $P(I_k, j)$ and $P(D_k, j)$ into a single 3-word block in DRAM for each k . Because the current sequence character $s[j]$ remains constant over each pass through the outer loop, we arrange the emission probabilities for M_k and I_k so that all the probabilities for one amino acid α are stored contiguously; for each α , pairs $[e(\alpha/M_k), e(\alpha/I_k)]$ are packed into a single block for each k , since the two values are always used together. All blocks are padded so that

reads and writes are properly aligned, avoiding costly read-modify-write cycles in the IXP’s memory controllers.

4.3 XScale and ME Interaction

The XScale implements a job dispatcher, which is responsible for sending Viterbi packets to the MEs and reading the results of each computation back out of DRAM. The XScale first reads the database from over the network and caches it in DRAM. This is necessary because our development platform lacks a local hard disk. When an ME becomes free, it signals the XScale. The XScale writes several control parameters into the ME’s registers, including a pointer to the next Viterbi packet, and signals the ME. The ME then transfers the packet into SRAM and begins its computation.

It may seem wasteful for the MEs to transfer models from DRAM to SRAM only to read them back from SRAM. There are several reasons for this organization. The first is memory alignment: DRAM can only be accessed on 8 byte boundaries, which is inconvenient and wasteful in our Viterbi implementation, whereas access from SRAM avoids such problems. Secondly, the bulk transfer leaves our design more flexible; a future implementation may make use of the IXP’s DMA controllers to achieve the same function without occupying the MEs. Finally, the transfer operations represent less than 1% of work done by the MEs.

4.4 Pipelining the Viterbi Algorithm

The MEs on the IXP lack any form of automatically managed cache and have local memories that are too small to hold all the data necessary to perform the Viterbi computation. To achieve good performance on this memory-intensive task, it is therefore essential to organize the computation so as to hide the long latencies associated with memory accesses. We use two techniques to reduce memory latency in our implementation: batching of reads and writes, and pipelining of memory operations.

The ME instruction set supports multiword read and write commands of up to 16 32-bit words to SRAM and up to 16 64-bit dwords to DRAM. Both these sizes are greater than the sizes of the transition probabilities (22 bytes), emission probabilities (4 bytes), and intermediate values (12 bytes) used for one motif position k in the inner loop of the Viterbi algorithm. However, actually reading and writing memory once for each k is inefficient for two reasons: first, it needlessly multiplies the number of distinct accesses queued by the SRAM and DRAM controllers; and second, it fails to take advantage of DRAM’s ability to transfer data in large multiword bursts. Multiplying the number of read and write commands increases the queue depth in the IXP’s memory controllers, leading to longer latencies for all accesses and stalls when the queues fill.

To maximize the efficiency of memory operations, we organize our computation so that data for multiple inner loop iterations can be transferred in a few large operations. All of the model transition probabilities needed to process M_k , I_k , and D_k for two consecutive motif positions k can be fetched by one multiword SRAM read command. A single SRAM read is also sufficient to load all the emission probabilities needed for 10 consecutive motif positions. Similarly, we can read and write all the intermediate probabilities for 10 motif positions using one DRAM burst read and one burst write. When the amount of data to be transferred at once exceeds the storage available in the IXP’s register set, we use the local memory of each ME to hold the data until it can be consumed by the computation (for reads) or written back (for writes).

Batching memory operations is effective in reducing total latency to the extent that fewer, larger operations, particularly to DRAM, require less total time to return the same number of bytes. However, batching cannot actually hide latency. Fortunately, the IXP architecture’s support for asynchronous memory operations provides a mechanism by which the Viterbi algorithm can be effectively pipelined. We use asynchronous operations to prefetch data ahead of when it will be needed – two motif positions ahead for transition probabilities, ten positions ahead for other model parameters and intermediate probabilities. Similarly, we issue a DRAM write for ten iterations’ worth of intermediate probabilities, then compute for a further 10 iterations before requiring that write to complete. Although such pipelining places great demands on the IXP’s limited number of transfer registers, we avoid this hazard by moving batched values to local memory as soon as they become available, thereby freeing their transfer registers for the next operation.

With the above optimizations, an IXP 2850 attached to commodity SRAM and DRAM chips and running one instance of the Viterbi algorithm per ME supports simultaneous operation of up to twelve MEs without saturating the command queues of the IXP’s memory controllers and with relatively few idle cycles in any engine due to memory latency (see section 5.4).

5. Experimental Evaluation

In this section, we compare the performance of the optimized HMMer application running on the P4 to JackHMMer running on the IXP 2850. The IXP’s 16 MEs can each issue 1 instruction per cycle; with a 1.4 GHz clock, this means it can perform 22.4 Gops/sec at peak. The P4, which can retire 3 micro-ops in 1 cycle and is clocked at 2.6 GHz, can retire 7.8 Gops/sec at peak. This means that if both implementations are CPU-bound, can find enough parallelism to keep their functional units

busy, and require similar numbers of instructions to execute HMMer, the JackHMMer implementation should outperform the P4 by almost 3x. Of course, these assumptions do not necessarily hold; hence, a more careful empirical comparison is needed.

5.1 Experimental Setup

JackHMMer was implemented on an Intel IXDP 2810 development platform [11] containing two IXP 2850 processors, of which we use only one. Attached to the 2850 are 768 MB of SDRAM and 32 MB of QDR SRAM. The operating system on the IXP is MontaVista Linux 3.1 [10]. The P4 implementation was run on a 2.6 GHz P4 with SUSE Linux 9.1 as an operating system.

JackHMMer’s Viterbi computation was run on the MEs of the IXP 2850, with the scheduler/job dispatcher run on the IXP’s XScale control processor. The reference implementation of HMMer was run entirely on the P4 and so does not require job scheduling.

The HMMer program performs several tasks in addition to running the core Viterbi algorithm. The most time-consuming, which accounts for up to 25% of running time on the P4, is reading in models from a database and converting them to a “log-odds” form required by the Viterbi algorithm. We eliminated this work from both P4 HMMer and JackHMMer by precomputing the log-odds form of each model and storing its binary representation; this optimization could be implemented in a production system. HMMer also performs post-processing of significant hits between models and the input sequence; while these operations constitute only a few percent of total running time on the P4, they are not currently implemented in JackHMMer. We therefore omit post-processing from HMMer’s cost on the P4 in our tests. A production implementation of JackHMMer would offload post-processing, which requires floating-point support, onto a host processor, where it would run in parallel with the much more compute-intensive Viterbi algorithm on the IXP.

JackHMMer and HMMer were tested by comparing the Pfam_ls database [4] (version 14.0) to 4 input protein sequences chosen at random from the SwissProt [5] database, with an average length of 497.5. The average number of positions per motif in Pfam_ls is 226, and there are 7459 models in the database.

5.2 Optimization of HMMer for the P4

To ensure a fair comparison between the P4 and our hand-written IXP assembly code version of HMMer, we improved the P4’s implementation of the core Viterbi algorithm (as distributed in HMMer 2.3.2) by manually optimizing the C implementation, then further editing the generated x86 assembly code. Unless otherwise indicated,

all performance comparisons in this section are to this optimized P4 implementation.

The major change in our C code versus stock HMMer was to alter the data layout in memory. Although the existing code already lays out the motif model and dynamic programming matrix to achieve good data cache locality, accessing all the data necessary to execute the inner loop of the Viterbi algorithm requires nine separate base pointers. Our modified layout, similar to that used by JackHMMer, recombines the various component arrays of model and matrix so that only five pointers need be maintained. Our implementation can therefore keep all needed base pointers in x86 architectural registers, eliminating the multiple memory reads needed to maintain these pointers in HMMer's original implementation.

We made numerous additional small changes to HMMer's Viterbi implementation to generate the best possible x86 assembly code using the Intel C Compiler (version 8.1.026). Our C-level changes produced assembly code for the Viterbi inner loop that we judged to be nearly optimal, except for one register spill and a number of missed opportunities to implement **max** operations with conditional moves, rather than with more costly branches. We did not use the x86 SIMD extensions because some operations, e.g., max, were not available in the 32-bit working precision required by HMMer. We corrected these deficiencies by hand-editing the assembly code and verified that our changes measurably decreased total execution time.

We believe that our final x86 assembly code for HMMer's Viterbi algorithm, particularly the code for the inner loop, is comparable in efficiency to the best implementation that could be hand-coded from scratch. In particular, the code is of comparable quality to our hand-coded IXP implementation. On a moderately sized model ($M = 544$), our implementation of the Viterbi algorithm runs 2.06x faster than HMMer's original implementation.

5.3 Comparative Performance

JackHMMer can match an input protein of length $n=544$ against the entire Pfam_ls database in 7.73 seconds, whereas optimized HMMer on the hyper-threaded P4 requires 14.05 seconds. In other words, the IXP achieves a speedup over the P4 of 1.82 on this computation, despite the latter's 1.85x faster clock. Section 5.6 will elaborate on why this is so.

Figure 4 shows runtimes for comparing protein sequences of various lengths to the entire Pfam database. As these figures show, the performance of both JackHMMer and HMMer is fairly insensitive to specific inputs. The time needed to process a single motif

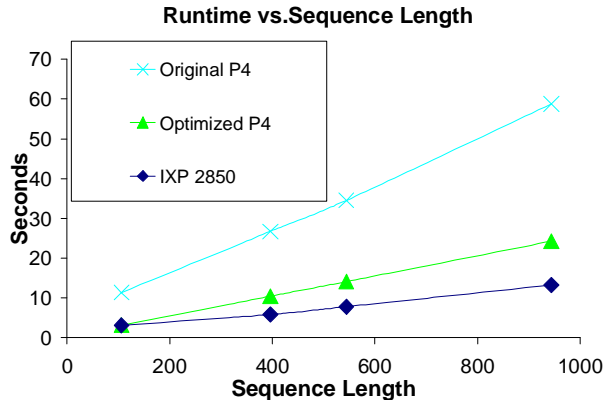


Figure 4: Runtime as a function of input sequence.

increases roughly linearly with motif size and sequence length, regardless of the actual model and sequence used.

While JackHMMer's throughput for processing models exceeds that of the P4, the latency required to process each single model is much greater on the IXP. When executing using only one ME, the IXP requires 79.76 seconds to run over our test data, while the P4 requires only 14.05 seconds. This result is unsurprising, since each ME of the IXP is substantially slower than the P4, though the IXP as a whole is faster. Fortunately, latency is relatively unimportant for non-interactive batch computations, such as running HMMer on a large database of models.

5.4 Performance Details for the IXP 2850

A useful measure of throughput for JackHMMer is "cells per second;" that is, how many entries in the Viterbi algorithm's matrix of intermediate probabilities can the system fill in one second? The size of this matrix is proportional to the product of the motif size and the protein sequence length, and the entire matrix must be filled in before the Viterbi computation can complete. Figure 5 shows what happens to the cells-per-second throughput metric as we increase the number of MEs used in the computation. As we increase the number of MEs from 1 to 12, our throughput increases nearly linearly, demonstrating the IXP's ability to exploit the available thread-level parallelism of HMMer.

Above 12 MEs, we begin to see diminishing throughput (due to SRAM contention as discussed below). The increase in throughput from 12 to 13 MEs is quite small (~1.3%, versus 12.2% were the increase linear). These observations imply that for the IXP 2850, we could use 4 MEs to provide additional functionality without sacrificing performance.

The bottleneck for the IXP 2850 implementation is at the SRAM control queues. We observe that, while the SRAM channels are 50% utilized, increased utilization

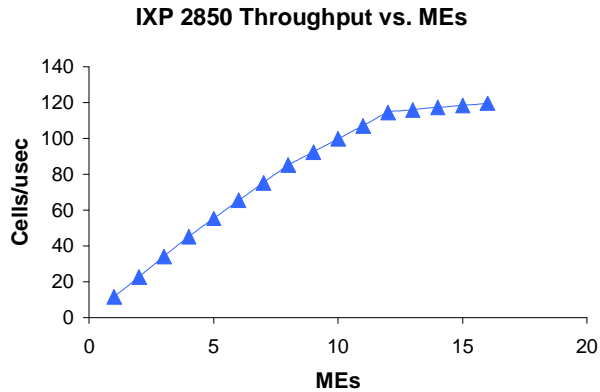


Figure 5: Throughput in cells/second of JackHMMer as a function of MEs used.

can be obtained only by using the maximum reference count of 16 32-bit words on every SRAM operation, rather than the 13 being used. This change would be awkward and difficult, with little if any advantage. Our observations suggest that we are saturating the SRAM controllers and are experiencing contention for its input queues. As a result of this contention, the MEs are idle almost 50% of the time. (Note that using multiple threads at each ME would not help due to full memory utilization.) A preliminary proposal for overcoming this barrier is discussed in the next section.

5.5 Scalability Limits of JackHMMer

Since JackHMMer is currently bottlenecked at the IXP’s SRAM memory controllers, we are unlikely to see great performance gains by simply increasing the clock speed of the MEs. One promising alternative is to put all the model data into the MEs’ local memory, eliminating the need to access SRAM. Although this approach cannot accommodate all models of useful size today, due to the limited local memory available on the IXP 2850, it may be attractive on a future IXP implementation with more local memory.

We have implemented a version of JackHMMer that keeps all its data in local memory, and have tested it using the subset of small models in Pfam_ls that will fit in local memory on the IXP 2850. This version achieves a 3.5x speedup over the P4. Figure 6 presents a comparison between this JackHMMer version and our original, SRAM-based implementation. The local memory implementation not only performs better, by a factor of 2.26x, but also scales better with the number of MEs. The regular JackHMMer implementation begins to see diminishing returns when using 12 or more MEs, whereas the local memory JackHMMer continues its linear increase in performance.

To assess whether local-memory JackHMMer’s performance is likely to scale with increasing clock speed,

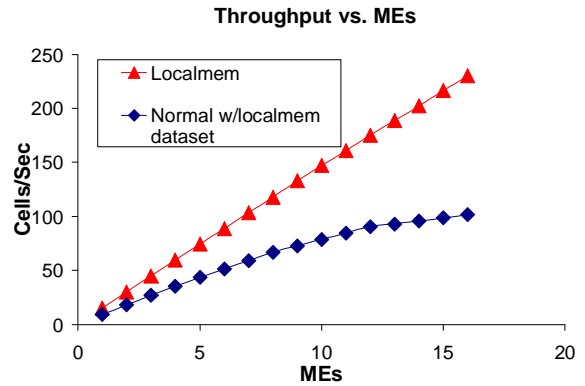


Figure 6: Throughput in cells/second of JackHMMer and Local memory JackHMMer

we investigated its performance on an IXP 2400, which has a clock speed of 600 MHz and 8 MEs (no significant changes were needed to the JackHMMer code). We observed a throughput of 51.87 cells/sec, compared to 229.94 cells/sec on the 2850. The 2850 therefore achieved a 4.43x speedup over the 2400, compared to an expected 4.66x increase given the former’s faster clock and larger number of MEs. These results suggest that local-memory JackHMMer is likely to scale well both with increasing clock speed and with increasing numbers of MEs.

5.6 Discussion

In this section, we have seen that JackHMMer on the IXP 2850 outperforms HMMer on the Pentium 4. This indicates that the IXP is a more efficient architecture for throughput-oriented Viterbi calculations. The IXP’s efficiency is due to a number of factors.

Significant coarse-grained parallelism. Running 16 computations in parallel proved more effective than harvesting ILP. The local memory JackHMMer also indicates that, given some improvements discussed in section 6.2, the 16-way simple CMP organization can be more effective at keeping ALUs active than the sophisticated superscalar pipeline in the Pentium 4.

Explicit, application-controlled mechanisms. JackHMMer is a good match for the kind of fine-grained control that programmers and code generators have when programming the IXP MEs: the data can be arranged in blocks for efficient memory usage; multiple memory channels can be used to increase bandwidth and reduce latency; and the working set size is known in advance. The application is able to increase ME utilization by requesting multiple blocks of data per request, by explicitly prefetching loop data in advance, and by building the loop body via software pipelining. Each of these improvements is enabled by the explicit nature of the ISA and the micro-architecture: the memory operations allow variable-sized requests and optional

blocking targeting specific memory controllers, and the ME pipeline is simple enough to schedule efficiently via software pipelining. Similar optimizations are not possible in the Pentium 4 due to (a) lack of guaranteed prefetching and (b) inability to effectively software pipeline due to a small number of architectural registers and the reordering carried out by the superscalar core.

Along with the superior performance and efficiency shown on this workload, an IXP system consumes less power and requires less space. The IXP 2850 is implemented in a 0.18 micron process and consumes 27.5 W typically and 32 W maximum. The P4 is implemented in with 0.13 micron process technology and typically consumes around 63 W.

While developing JackHMMer, it became clear that a small number of ISA modifications could have a significant impact on code efficiency. The addition of a conditional move instruction would remove branches from our inner loop, resulting in reduced instruction count and 20% fewer unfilled branch delay slots. Also, while the ME ISA has good support for bit and sub-word register manipulations, non-networking code like the Viterbi computation could achieve greater code density if arithmetic sign extension modes were added to these classes of instructions. The arithmetic shift right instruction already supports this mode, so it would likely be a small change that would improve our code density.

Of course, making use of the explicit, application-controlled mechanisms in the IXP typically requires programmer involvement. This increases the time and attention given to code generation. However, for performance critical code, the benefits often outweigh the costs. Implicit mechanisms can often hinder high-performance software development by keeping the performance-relevant details out of the view of the code generator. In this situation, the developer can neither maximize performance nor know with confidence that greater optimization is possible. We note that the cycle-accurate IXP simulator was critical in allowing us to identify system bottlenecks and work-arounds.

6. Estimates for Future Processors

In this paper, we have evaluated the performance of HMMer/Viterbi on two contemporary processor technologies. In this section, we explore whether the IXP is likely to remain an attractive competitor to the Pentium in this application domain. To this end, we next consider the features of next-generation processors and their likely effect on HMMer/Viterbi.

6.1 Intel x86 Family

In the past year, Intel has indicated that the major architectural advance in future IA-32 processors will

come in the form of integrating multiple cores onto a single die. In the next several years, we therefore expect IA-32 products to integrate small numbers (2-4) of superscalar cores onto individual chips.

Based on the performance results we have seen on the P4, we expect the transition to dual- and quad-core processors to improve HMMer/Viterbi performance by a factor of 2-4x relative to the current Pentium 4 performance, assuming that the cache hit rate remains high.

Other potential improvements include x86-64 and additional hyper-threading. Running HMMer on an Opteron processor, in both 32- and 64-bit mode, indicates that the move to x86-64 will yield a further 10% speedup. If additional threads bring the IPC to 3 (the maximum possible), this would yield another 38%.

These improvements, plus an increase in clock speed from 2.6 to 4 GHz, yield an expected speedup of 4.7-9.4x.

6.2 Intel IXP Family

In the next generation of IXP processors, we expect three forms of relevant resource scaling: increased ME count and clock frequency, more local memory at each ME, and increased ME issue width.

6.2.1 ME Count and Clock Frequency

As demonstrated by our local memory implementation, performance can potentially scale linearly with both ME count and clock frequency, assuming we could fit all model parameters into local memory.

6.2.2 Increased Local Memory

Based on comments made recently by the Intel IXP Architecture Team (at the 2004 Intel University Summit), the next generation of IXP may feature a writeable control store that enables unused locations (i.e., those not holding instructions) to be used as local data memory.

This resource is particularly significant for JackHMMer. If we assume an 8K control store, as currently found on the IXP 2850, and note that our current JackHMMer implementation uses only 566 control store entries, then this development would increase our effective local memory size from 640 words to over 8K words. Notably, this increases the effective local memory size without increasing the ME footprint or gate count.

JackHMMer's current data working-set size, for all models fewer than 1300 states in length, is around 7000 words. This implies that with 8K words of writable control store, the local memory version of JackHMMer will be usable for most models.

6.2.3 Increased ME Issue Width

The next-generation Intel IXP processor could double instruction throughput by organizing each ME as a statically scheduled dual-issue processor. Compared to the previous two advances, this development is the most speculative, since (to our knowledge) Intel has never hinted at plans to increase the issue width of the ME. However, there is precedent for such an increase among NPs, most notably Cisco's Toaster II network processor, which featured an array of 4-wide VLIW cores [9].

Increasing issue width via static scheduling would double peak compute performance while incurring moderate increases in ME footprint and complexity. For local memory JackHMMer, a dual-issue processor would effectively double performance, since the inner loop has been scheduled via software pipelining and is unrolled twice. In essence, the inner loop would finish in half the time.

6.2.4 Performance Relative to the IXP 2850

Based on our current JackHMMer performance on the IXP 2850, the modifications above would improve performance as follows.

Increased local memory: 2.26x. As indicated in section 6.2.2, we expect the next generation of IXPs to contain enough local memory to make our local memory JackHMMer usable on most models. Our experiments indicate that moving from our current JackHMMer implementation to the local memory version will yield a 2.26x speedup.

Increase ME count from 16 to 32: 2x. With the external memory bottleneck eliminated, performance should scale linearly with ME count.

Increase ME clock frequency from 1.4 GHz to 3 GHz: 2.14x. Performance would also scale linearly with ME clock frequency.

Increase ME issue width from 1 to 2: 2x. Our twice-unrolled, software-pipelined loop body would condense to about half its original size.

In all, we would expect the next generation IXP to see an aggregate speedup of 19.34x. If we drop the speculative suggestion of a dual-issue ME, leaving only the highly probable developments, we are left with a potential speedup of 9.6x relative to current IXP 2850 performance. While our characterization of future features is by no means certain, it seems likely that the IXP architecture will remain a competitive option in the future.

7. Related Work

JackHMMer builds on previous work in accelerating HMMer and in implementing bioinformatics applications on network processors. Commercial implementations of

HMMer have been developed that place the core Viterbi computation in reconfigurable FPGA hardware; one such system is TimeLogic's DeCypher engine [25]. TimeLogic uses several FPGA PCI cards in conjunction with a multiprocessor Sun Sparc host system to accelerate HMMer. They claim to achieve performance equivalent of 2600 1 GHz Pentium III processors for this application. If we assume that performance varies directly with clock speed between the P3 and P4, this means it would take roughly 540 IXP 2850s to equal the Timelogic system.

FPGA-based versions of HMMer, like JackHMMer, typically implement scoring of sequences against a model but require the host system to reconstruct the optimal alignment path. Unlike JackHMMer, they typically use a *reduced version* of the full motif HMM, in particular one that does not include the serializing feedback loop through the J state. With this change, FPGA implementations are able to invert the order of the two loops in the Viterbi algorithm and proceed in sequence-major, rather than state-major, order; however, they lose the capability to score multiple motifs in a single protein. In contrast, JackHMMer's Viterbi implementation preserves the exact semantics of the original implementation.

HMMer is not the first bioinformatics application to be ported to a network processor. Bos and Huang [26] implemented part of the BLAST algorithm for sequence-to-sequence alignment using an IXP 1200, an earlier version of the IXP architecture (6 MEs running at 232 MHz). They focused only on implementing BLAST for DNA sequences and on the initial filtering stage of the algorithm, which can be implemented efficiently as a lookup in a dictionary of strings. Their implementation achieved parity with a 1.8 GHz Pentium 4 processor, suggesting the promise of the IXP for accelerating fundamental bioinformatics computations. JackHMMer presents further evidence that this application domain is a good match for network processors.

8. Conclusion

When designing CMPs, architects can choose to replicate either a small number of complex, superscalar cores, or many simpler ones. We believe that the latter design, currently used by network processors, can be effective in domains beyond networking.

We have explored the relative merits of implementing HMMer, a scientific workload from the domain of bioinformatics, on both a network processor and a traditional superscalar, represented by the the IXP 2850 and the Pentium 4 respectively. The IXP, despite its 1.85x slower clock, achieves a 1.82x speedup on HMMer compared to the Pentium, thanks to: the application's high degree of coarse-grained parallelism, relatively

modest memory usage, and predictable access patterns, all of which enabled aggressive software pipelining of multiple processors. Other uses of the Viterbi algorithm, as well as other scientific computations with similar characteristics, are potentially attractive targets for acceleration on a network processor architecture.

While we worked relatively hard to hide the latency of memory accesses in JackHMMer, future developments in the IXP family of network processors seem likely to remove the anticipated bottlenecks that limit the scalability of ME clusters by greatly reducing the need for all MEs to access a shared memory. If these developments come to pass, we anticipate that future JackHMMer implementations could run 10 to 20x faster than the current version. We have shown a proof of concept model for this in our local memory implementation. In contrast, likely developments in general-purpose superscalar-based CMP CPUs seem likely to yield at most a 10x speedup. The clear path to potential performance improvement in network processors, along with their attractive compute density and high degree of user control over optimizations, suggest that a relatively modest investment in these architectures could make them a major force in accelerating scientific computing applications.

9. Acknowledgements

This work was supported by a gift from Intel Corporation and by NSF grants CCF-0430012, CCF-0427794, CNS-0435173, and DBI-0237902.

10. References

- [1] Intel Pentium 4 Processor Family Product Information. <http://www.intel.com/products/desktop/processors/pentium4>.
- [2] HMMER: Sequence Analysis Using Profile Hidden Markov Models. <http://hmm.wustl.edu>, 2004.
- [3] Hoel, P.G. Introduction to Mathematical Statistics, 3rd ed. New York: Wiley, 1962, page 57.
- [4] Alex Bateman, et al. The Pfam Protein Families Database. *Nucleic Acids Research*(2004) 32:D138-D141.
- [5] Boeckmann B., et al.: The Swiss-Prot protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic Acids Res.* 31:365-370(2003).
- [6] Austin, T.M., Burger, D.. The SimplScalar Tool Set, Version 2.0. <http://www.simplescalar.com>, 2003.
- [7] "Pthreads: POSIX threads standard", *IEEE Standard* 1003.1c-1995.
- [8] Chandra, P., et al. (2003). Intel IXP 2400 Network Processor: A Second-Generation Intel NPU. Crowley, P., et al. (Eds.), *Network Processor Design Issues and Practices vol. 1* (pp.259-275). San Fransisco, CA: Morgan Kaufmann Publishers.
- [9] Marshall, J. (2003). Toaster2. Crowley, P., et al. (Eds.), *Network Processor Design Issues and Practices vol. 1* (pp.235-248). San Francisco, CA: Morgan Kaufmann Publishers.
- [10] Montavista Software- Powering the Embedded Revolution. <http://www.mvista.com>.
- [11] Radisys: Embedded Systems and Solutions. <http://www.radisys.com>.
- [12] Intel 21555 Non-transparent PCI-toPCI Bridge. <http://www.intel.com/design/bridge/21555.htm>.
- [13] Rabiner, L.R. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77:257-86, 1989.
- [14] Haussler, D., Krogh, et al. eds, Proc. of the 26th Annual Hawaii Int'l Conf. on System Sciences, volume 1, 792-802, IEEE Computer Society Press, 1993.
- [15] Krogh, A. Hidden Markov models for labeled sequences. In *Proceedings of the 12th IAPR Int'l Conf. on Pattern Recognition*, 140-144, IEEE Computer Society Press, 1994.
- [16] Graham, S., Kessler, P., and McKusick, M. *gprof: A Call Graph Execution Profiler*. In *Proc. of the SIGPLAN '82 Symp. on Compiler Construction* (Boston, MA, June 1982), Association for Computing Machinery, pp. 120-126.
- [17] Intel C++ Compiler for Linux User's Guide. Intel Corporation, 2003.
- [18] *Free Software Foundation* (FSF), GCC Home Page. <http://www.gnu.org/software/gcc/gcc.html>, 2004.
- [19] J. Vlontzos, S. Kung . "Hidden Markov Models for Character Recognition", *IEEE transactions on image processing*, 1(4), oct 1992.
- [20] A. Kundu, Y. He, P. Bahl, "Recognition of handwritten word: first and second order hidden markov model based approach", *Pattern recognition*, 22(3), 1989
- [21] K. Aas, L. Eikvil, R.B. Huseby, "Applications of hidden Markov chains in image analysis", *Pattern recognition*, 32(4), p. 703, 1999.
- [22] , J. Li, A. Najmi, R.M. Gray, "Image Classification by a Two-Dimensional Hidden Markov Model" *IEEE transactions on signal processing*, 48(2), p. 517, Feb 2000.
- [23] T. Ryden, et al, "Stylized Facts of Daily Return Series and the Hidden Markov Model", *Journal of applied econometrics*, 13(3), p. 217, May 1998.
- [24] Dirk Outston et al. "Applications of hidden Markov models to detecting multi-stage network attacks." In *Proc. of the 36th Hawaii Int'l Conf. on System Sciences*, IEEE Computer Society Press, 334-344, 2003.
- [25] DeCypherHMM Solution. http://www.timelogic.com/decypher_hmm.htm, 2004.
- [26] Bos, H. and Huang, K. "On the Feasibility of Using Network Processor for DNA Queries". In *Proc. of the Third Workshop on Network Processors & Applications* (NP-3), pp. 183-195, 2004.
- [27] Intel Vtune Performance Analyzers. www.intel.com/software/products/vtune/
- [28] Chukkapalli G., Guda, C. and Subramaniam S. SledgeHMMER: A web server for batch searching Pfam database, *Nucleic Acids Res.* , 32:W542-544
- [29] HMMERHEAD, Unpublished; <http://www.cs.huji.ac.il/labs/compbio/hmmerhead/>