CHARACTERIZING THE PARALLEL PERFORMANCE AND SOFT ERROR RESILIENCE OF PROBABILISTIC INFERENCE ALGORITHMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Vicky W. Wong

September 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mark Horowitz) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Daphne Koller)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis)

Approved for the University Committee on Graduate Studies.

Abstract

Probabilistic reasoning has become a popular approach for modeling systems with uncertainty and solving for the most likely solution based on the data available. It has been successfully applied to many exciting fields and its applications are expanding. However, there has been little work on how they map to modern and future computing systems. Continued scaling of VLSI circuit technology is driving processor design towards explicitly parallel machines as energy constraints and diminishing return of instruction level parallelism limit the performance gain possible with monolithic processors. Smart Memories is a chip multiprocessor (CMP) architecture that is designed to be efficient across a wide variety of applications. Using it as our evaluation platform, we characterize the performance of a set of probabilistic inference algorithms on a CMP system.

Our results show that probabilistic inference applications have plenty of data parallelism that is easily extractable in most cases. For some applications, hardware supported multi-context processors and fine-grain synchronization are necessary to achieve efficient parallel execution. Unlike parallel scientific benchmarks, these applications have lower compute to memory ratio as well as large working sets. Thus, high memory bandwidth is required and using multi-context processors to hide some of the memory latency is desired.

Besides the shift to CMP's, technology scaling is also fueling a growing concern on the reliability of future processor chips, as shrinking feature size and lower voltage make devices more susceptible to upsets from transient errors. Since probabilistic reasoning is designed to handle noisy or incomplete data, it raises the question of whether they are more robust than traditional programs, and if that warrants a different approach to soft error protection.

Our fault injection experiments confirm that the robustness of approximate inference algorithms makes them more resilient against transient errors compared to traditional benchmarks. In addition, the approximate nature of the computation enables low-cost fault recovery. With simple modifications in the software, we can further improve the percentage of soft errors masked. The errors that the algorithm cannot naturally recover from often point to critical sections of the program and we find that algorithm specific software level error protection can be very effective in increasing robustness while incurring little additional overhead.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As technology advances, the use of microprocessors continues to spread. In some of these systems, the applications assume a simplified and idealized view of the world where all the necessary data for the computation is known and available. In other systems however, it is important to recognize the uncertainty in either the environments or the collected data. In these applications, the input data may be noisy, so the data can not trusted to be totally accurate, or incomplete. Given the uncertainty in the input data in these problems, instead of requiring an exact solution, a "good enough" solution is often sufficient. One approach to incorporate uncertainty is probabilistic reasoning, which encodes knowledge as probability distribution and bases its computation on probability theory. Instead of computing an exact result, the goal is to compute the likelihood of certain events. For example, in a medical diagnosis system, one might query the probability of being infected by a certain virus given the symptoms and/or test results. The algorithms that perform these queries are called probabilistic inference algorithms. These algorithms are the basic building blocks for many probabilistic reasoning systems that also involve learning and planning.

Probabilistic reasoning is an old idea and has been applied in many fields. Much of the modern work on inference grew out of the early work done by researchers in artificial intelligence, starting with the early expert systems [1, 2, 3]. Since the 1960s, it has been applied in a variety of applications, such as medical diagnosis [4, 5] and speech recognition [6]. In recent years, this approach has been shown to successfully tackle difficult problems in growing fields like data mining [7], image analysis [8], robotics [9], and genetics [10, 11]. Probabilistic reasoning's ability to use information from a large amount of data, and using constraints between the data to compute the most likely answer has shown to greatly increase the robustness of many applications. Given the imperfect nature of most models and data for real world systems, probabilistic algorithms are likely to be used in even more applications and on larger datasets, and be a significant part of the workload in many future computing systems. Thus, this thesis attempts to characterize the computational requirements of these algorithms in order to understand how to adapt these algorithms to run more efficiently on future machines, and how to adjust the machines to run these applications more efficiently.

Computer performance has been driven by the exponential growth in the number of transistors that can fit on a chip. Each generation of circuit technology has enabled new hardware features that improved processor performance. While technology scaling continues at an amazing pace, recent technology constraints have caused a paradigm shift in processor design from uniprocessor to multi-core architecture that executes multiple instruction streams in parallel. This change has been driven by the need to create more power efficient processors. These new multiprocessor chips are quickly becoming the norm in personal computers. Based on the current trend, future machines will be explicitly parallel machines with increasing number of processor cores. Therefore, we need to understand how suitable multi-core architectures are for probabilistic inference algorithms.

Besides driving the migration to multi-core architectures, technology scaling has raised some concern in reliability of digital circuits. Although there has always been a possibility of a transient error occurring in the circuit, the probability was low enough that error protection was considered only in mission critical systems. However, as feature size shrinks, circuit components become more susceptible to errors caused by particle strike and noise from another signal or a power source. Thus, these errors may no longer be conveniently ignored. At the same time, there is a cost in performance, power and die area to achieve reliability, so it is beneficial to consider the trade-off between the overhead of error protection techniques and an application's sensitivity to transient errors.

For applications in domains such as scientific computing, networking and multimedia, detailed analysis on their runtime characteristics have led to various application specific architectures which match the machines to the types of parallelism in the applications. As number of applications that are built on probabilistic inference algorithms explodes, it is useful to perform the same analysis on this class of applications. This thesis explores the parallelism, compute and memory behavior, and error tolerance of these applications, and what machine features are needed for efficient execution.

1.1 Organization of Rest of Thesis

To provide a context for the performance aspect of our characterization, Chapter 2 first provides an overview of processor scaling and explains the reason for the current trend toward multi-core systems. It then describes a specific multi-core architecture that serves as the platform for the performance evaluation in this thesis. It also reviews how application characterization is usually done and which characteristics are important for our analysis. Having established the platform of our performance analysis, we also need to have a basic

understanding of the applications in order to interpret the characterization results properly. Chapter 3 first provides an introduction to different models and types of algorithms commonly used in probabilistic reasoning, and explains our choice of applications used for this thesis. With this understanding of the kind of processing required in these applications, Chapter 4 presents the performance characterization results. It provides an analysis of the parallel execution characteristics that are indicative of how well these algorithms will scale on future machines. Chapter 5 turns to the error tolerance of these applications. It first explains the rise of the reliability issue. It then shows the soft error masking ability of probabilistic applications and techniques that can improve the robustness for this type of applications. Lastly, we conclude this research in Chapter 6.

Chapter 2

Architectural Trends

Historically, performance of uniprocessors had been improving exponentially as circuit technology scales. The growth in the number of transistors on a chip has followed Moore's Law [12], which states that the number doubles roughly every 24 months. This rapid increase in number of devices has driven innovations in computer architecture that enable the performance gain. However, the continued scaling has raised some concerns in two areas: performance and reliability. In terms of performance, we have observed a slow-down in performance gain driven by advancement in circuit technology. The question is whether and how we can continue to convert additional transistors to higher performance. The concern with reliability is that as the transistors shrink in size, they become more susceptible to outside interference and transient errors would be more frequent. We explore the reliability aspect in Chapter 5. In this chapter, we focus on the implications of technology scaling on application performance and how this motivates the types of characterization that we present later in this thesis. To understand this relationship, we need to review how microarchitects had been able to drive performance of uniprocessors higher using more transistors and why

that is no longer easy to achieve. Then we discuss the emergence of chip multiprocessors (CMP) as a solution to address these challenges. To provide concrete performance numbers, we describe in more details a flexible CMP architecture called Smart Memories, which serves as the platform for our evaluation.

2.1 Technology Scaling and Evolution of Uniprocessor Architecture

The continued refinement of VLSI circuit fabrication technology means that the number of transistors that can fit on a reasonably sized die grows exponentially with each generation (as illustrated in Figure 2.1). This has enabled computer architects to implement novel hardware features that shorten the execution time of a program, either by reducing the latency of an existing operation or by increasing the number of concurrent operations. For example, on-chip caches and branch prediction reduces the memory access and branch resolution latency respectively, while pipelining, superscalar execution, out-of-order execution and speculative execution allow more instructions to be executed simultaneously. These enhancements made it possible for uniprocessor performance to improve exponentially as technology scales (as evident in Figure 2.2). However, this tremendous growth is slowing down and it is unlikely to return to the original pace. There are a few major factors that contribute to the slow-down. One is that the performance gain on a uniprocessor mostly came from executing more instructions in parallel. Since dependencies among instructions are inevitable, there is a limit to how many instructions can be concurrently executing (which is referred to as instruction level parallelism or ILP of a program) [13]. Moreover, each hardware feature that is necessary to extract the remaining parallelism that is yet untapped increases the energy requirement

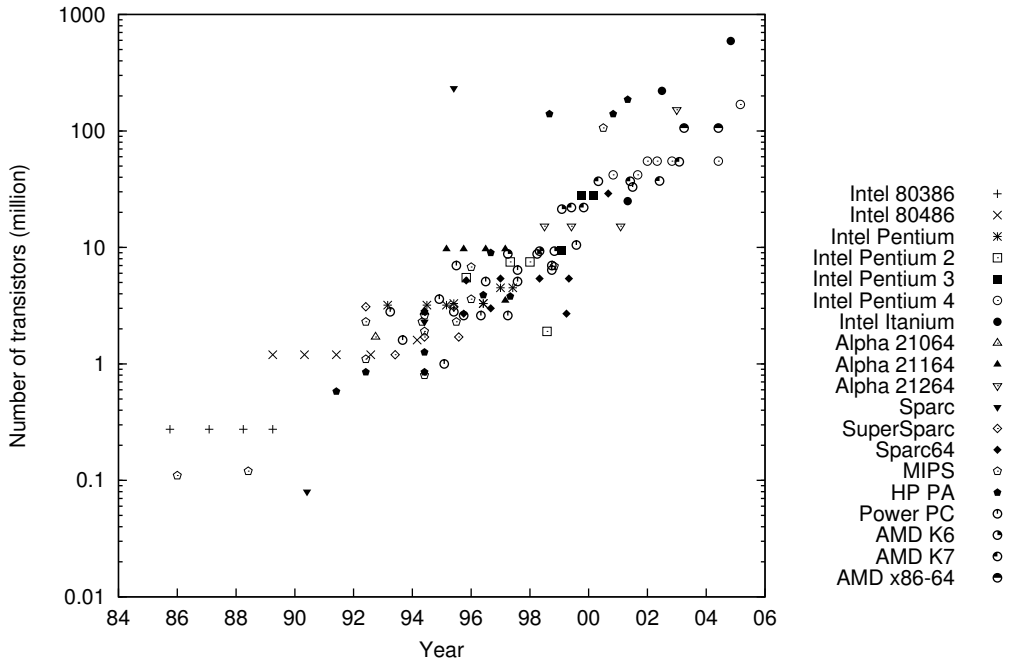

Figure 2.1: Number of transistors on various commercial microprocessors

of the chip. This is a problem for modern uniprocessors as the power consumption and heat dissipation in those chips is already approaching the economic limits. Besides the difficulty of further increasing instruction parallelism, global communication across the chip has also become an issue. Signals can no longer travel across the chip within a processor cycle and the energy spent on communication is also becoming significant. Figure 2.3 shows how processors became less power efficient as their performance improved. Computer architects noticed that by reducing the peak performance of a processor slightly, through removing the most energy inefficient mechanisms, one could build a processor core that is much smaller and consumes lower power than the original processor. Putting two of these on a die provides more performance than the original processor, and still dissipates less power. As a result, processor designs have been moving towards explicitly parallel architectures.

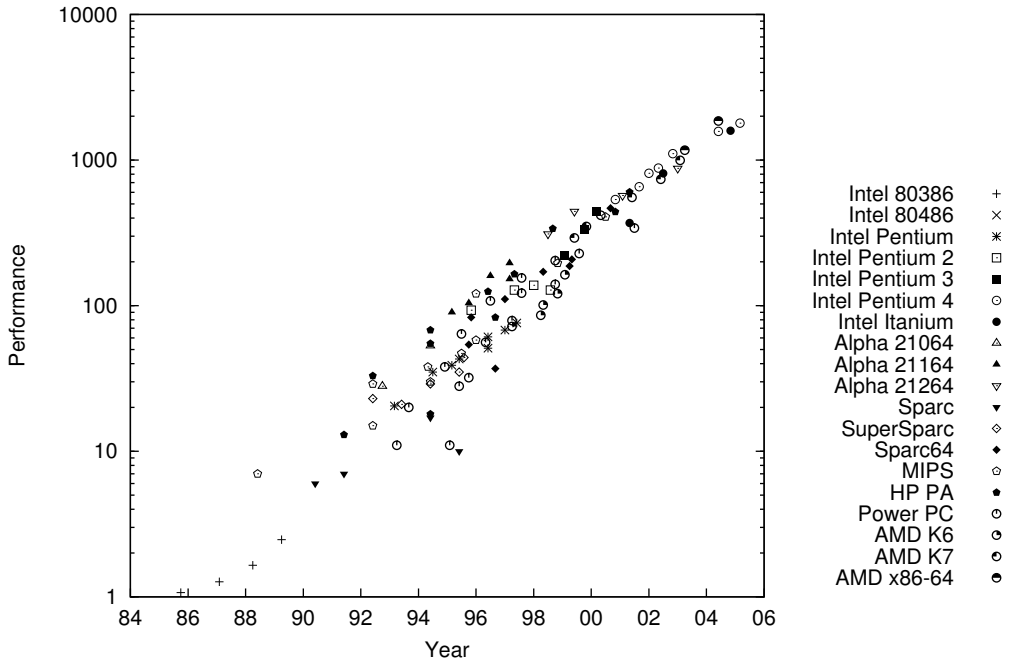

Figure 2.2: SPECint performance ratings for various commercial microprocessors

2.2 Extracting Parallelism in Explicitly Parallel Processors

While parallel architectures have been around for more than three decades, multiprocessors of previous generations were limited to supercomputers and servers, running scientific and financial applications for the most part. As circuit technology has reached the point where multiple processors can fit on the same chip and improving the performance of a single processor became harder, chip multiprocessors have become the norm for personal computers and laptops. A chip multiprocessor (CMP) is an explicitly parallel architecture composed of multiple processing cores that exploit data parallelism and/or task parallelism; often each processor is sophisticated enough to exploit ILP in the application. Data parallelism arises in applications that carry out the same operations on multiple pieces of data where the operation on each is largely independent on the others. On the other hand, task parallelism

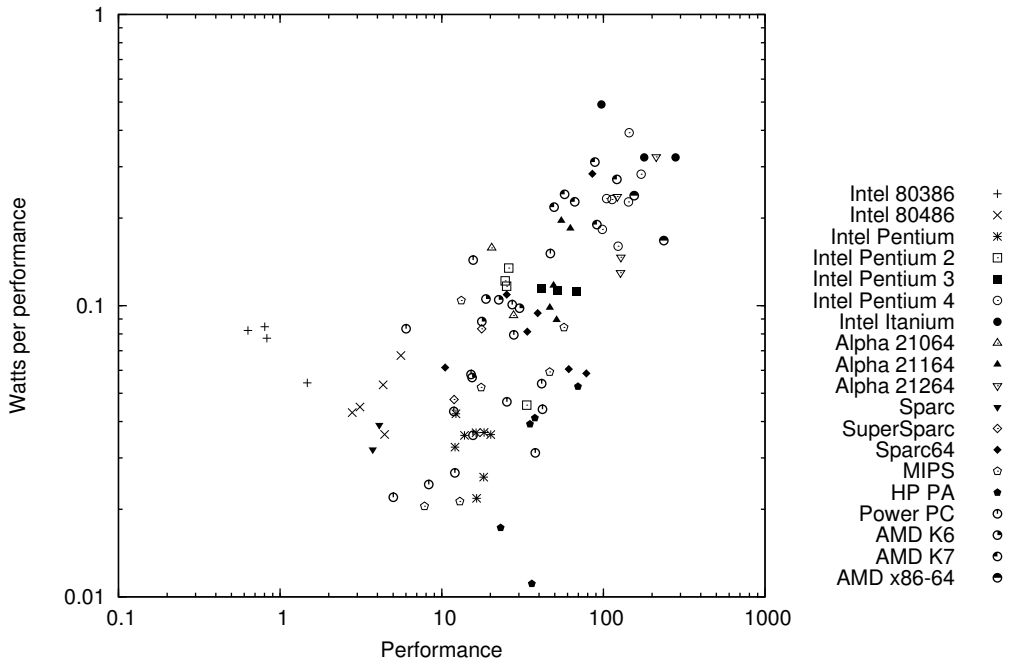

Figure 2.3: Power efficiency of various commercial microprocessors (both power and performance data are normalized by the technology)

occurs in applications where different phrases of a program can be executed in parallel. This can be orchestrated in a pipelining fashion where data is passed from task to task. Both data and task parallelism are a coarser level of parallelism than ILP and extraction of such parallelism has to be explicitly directed by the programmer and compiler.

There are different ways to exploit the data and task parallelism in a program. The most commonly used approach is the thread programming model. A thread is an independent instruction stream with its own control flow. For data parallelism, each thread executes the same sequence of code but operates on a different subset of the data. For task parallelism, each thread performs a different function, often on the output of another task. In both cases, the cooperating threads may need to communicate with each other, and synchronization is the means of coordinating accesses to shared data among threads. The advantage of the

thread programming model is that it is quite general; it is easy to implement any data parallelism, or task parallelism using threads. While implementing data parallelism with threads is generally straightforward, general threaded programs can be quite complex since ensuring correct synchronization in all situations, regardless of how the threads are scheduled and executed on the hardware, can be tricky.

For some data parallel applications that have predictable memory accesses, and where the operations on each data item are exactly the same, one can use a stream or vector programming model. Unlike the thread model, there is only a single instruction stream, which operates on multiple data streams simultaneously (SIMD). Since only one instruction stream is fetched, these types of execution engines are very efficient. The main limitation is that data dependent operations within a parallel region are highly restrictive and incur overhead. In addition the need to separate out memory references from normal computation means that these types of parallel programs tend to have a very different structure from their sequential counterparts and thus require large programming effort.

More recently, there is a lot of work on the transaction or thread level speculation programming model. The idea is to simplify the construction of a parallel program by allowing the programmer to optimistically designate tasks to be carried out in parallel without knowing all the dependencies among the tasks. During execution, software and/or hardware mechanisms dynamically detect and resolve any dependency that arises. This reduces the programming effort required to code a correct parallel program and potentially enables the programmer to extract parallelism from sources with complicated dependencies. Nonetheless, achieving optimal performance usually requires some dependency analysis and intervention by the programmer.

Each of these programming models requires different hardware features, so in the following section we describe a flexible architecture that supports all of them. We focus on the thread programming model since all the probabilistic reasoning applications that we considered could be nicely mapped into this model.

2.3 Smart Memories: A Modular Reconfigurable CMP

Smart Memories is a computer architecture project that aims to efficiently execute a wide variety of parallel programs and my work in that project led to the work on performance characterization of probabilistic inference algorithms. To evaluate probabilistic applications on a parallel machine, we would like to have a flexible underlying machine that would allow us to explore a number of different hardware mechanisms and ways to extract parallelisms from an application. The Smart Memories architecture provided a good evaluation substrate. The goal of Smart Memories was to design a general purpose processor that supported many parallel computation models. Our approach was that the architecture design should be driven by programming models, since how a parallel application is implemented not only dictates how computation is distributed but also determines the scheduling of communication. The programming models supported are multithreaded, stream and transactional programming, and I was responsible for exploring how to support multithreaded programs efficiently. To achieve the flexibility required to efficiently support vastly different programming models, we came up with a modular reconfigurable architecture. It is composed of building blocks we called tiles. The size of the tile is small enough such that propagation latency between any two points within the tile can be efficiently done in one cycle. Four tiles are grouped together to form a quad. The tiles within a quad are connected together by a low overhead intra-quad interconnection network. The intra-quad network is connected to a global

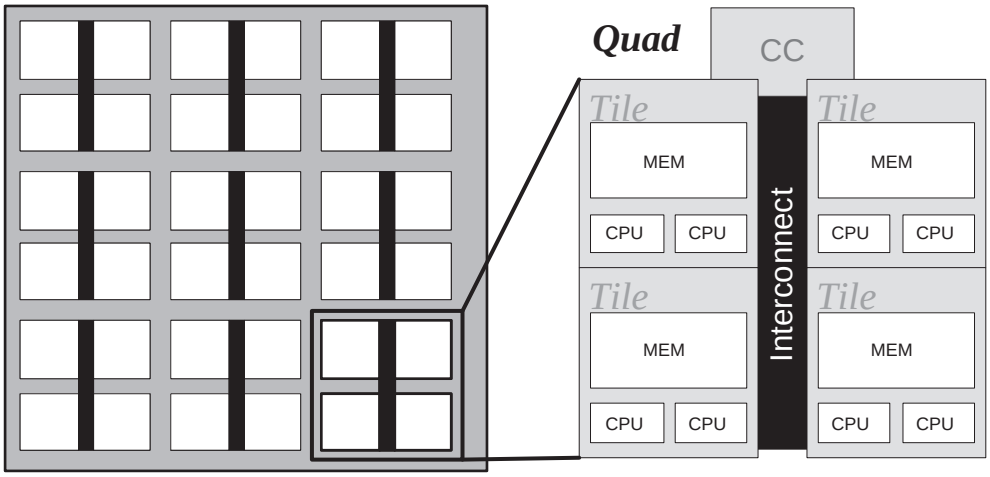

Figure 2.4: Smart Memories block diagram

dynamically routed network via a network interface called the cache controller. Figure 2.4 illustrates this hierarchical organization.

A tile consists of two processor cores, local SRAM memory and a crossbar network that connects them together, as well as to the intra-quad network. Each processor core is a simple in-order pipeline and can issue up to 3 operations per cycle (up to one floating point and two integer operations). The local SRAM memory is composed of memory mats. Each memory mat is an independent SRAM cell array that is one word (32 bits) plus five meta bits wide. Besides the usual read/write logic, each mat has some programmable logic that can perform atomic read-modify-write operation on the meta bits in parallel with a read/write operation on the word. The meta bits implement various meta-data often associated with memory states, such as cache line status, synchronization state, speculative state, and so on. In addition, a simple address generator allows auto-increment/decrement accesses for FIFO buffers.

Traditionally, a cache controller is responsible for the maintenance of caches. It tracks cache misses occurred in the local caches in its Miss State Handling Registers (MSHR)

	Memory mats	Cache controller	Instruction fetch
Multithreading	Caches (with fine-grain synchronization)	Cache refill, thread stall/wake-up	Independent fetch for each core
Streaming	Stream registers, scratch pads, instruction store	DMA engine	Combined fetch of microcoded instructions
Transaction	Caches (with speculative states), write buffers	Cache refill, commit & violation	Independent fetch for each core

Table 2.1: Configurations of Smart Memories for different programming models

and communicates with cache controllers in other quads or off-chip memory controllers to perform various cache coherence operations. In Smart Memories, the coherence protocol is programmable. Furthermore, cache coherence is just one of the roles of the cache controller. It can also use the MSHR to keep track of other types of thread stalls (which will be discussed below). In addition, it has a Direct Memory Access (DMA) engine that can perform memory gather/scatter operations without constant supervision from the tile processors.

The flexibility of Smart Memories comes from the programmability of the functions of memory mats and cache controllers, and the configuration of the instruction fetch engine. This enables the chip to execute programs of three parallel programming models: multithreading, streaming and transaction. Table 2.1 describes the high level configurations for those programming models. We will focus on the multithreading execution mode since it matches well with all the probabilistic inference applications that we studied.

In the multithreaded execution mode, the memory mats are used as instruction and data caches. Since there is no special tag mat, some mats store cache tags and the programmable logic is used to perform tag comparison, while other mats store the data part of the cache lines. The resulting structure acts like a normal cache. An access to a particular word is

controlled by the hit signal that is generated by the matching tag mat and the corresponding bits in the address that select the word from a cache line.

For multithreaded programs with irregular communication and synchronization, it is useful to support efficient fine-grain synchronization. This can be achieved by using the meta bits. Each word has an associated full/empty (F/E) bit that provides word-granularity synchronization [14]. Customized load/store operations access and potentially modify the F/E bit along side an access to the data word. By convention, an empty state blocks a load operation and a full state blocks a store operation. A blocked access causes the issuing thread of execution to stall, and a message to be sent to the cache controller (similar to a cache miss message), which keeps track of stalled threads. To facilitate thread stall and wake-up, we use another meta bit as a "waiting" bit to signal that some thread is blocked on the F/E state of that word. An synchronized access that updates the F/E state of a word with its waiting bit set triggers a message to the cache controller to wake up one or more blocked threads. Using the F/E bits, per word locks incur no additional latency since the synchronization operation takes place in parallel with the regular cache access. F/E bits can also be used to implement other synchronization constructs like barriers and semaphores.

In modern processors, particularly parallel processors, it is important to be able to overlap computation with communication or memory operations since a processor is idle while it is waiting for data required for the next operation. This is especially true as the gap between processor speed and memory speed continues to grow. Different programming models have different ways of hiding memory latency. Multithreading can achieve this hiding by enabling multiple contexts per processor. A context encapsulates the states necessary for a thread to execute on a processor. Each core in a Smart Memories tile can support two contexts in hardware. Only one context is active at one time. When a long latency

operation, such as a cache miss, is encountered, the processor switches execution to the other context. Depending on the frequency and length of stall events, a portion of the stall latency is hidden from the performance point of view, since it is overlapped with the execution of the other thread.

In order to evaluate how well these features work, we use this multithreading configuration of Smart Memories as the basis of performance evaluation for probabilistic inference applications in Chapter 4. The detailed specification of the configuration used will be described in that chapter as well.

2.4 Traditional Performance Analysis

Before we move on to the discussion of the applications used, we need to understand what characteristics of the application we will need to measure. Workload characterization is a way for computer architects to understand the performance and resource requirements of the target applications and narrow the suitable design space. It points out where a program spends its time, what are the potential bottlenecks, where extra hardware can bring the biggest improvement and so on. As computer architecture evolves, so does the kind of characterization that is the most relevant.

A program's performance is defined as the inverse of its execution time, which in turn is determined by three parameters: total number of instructions executed, clock frequency of the processor and the average number of instructions retired per cycle. Their relationship is expressed by the following formula.

$$time_{exec} = \#instructions \times \frac{1}{frequency} \times \frac{1}{\#instructions_{retired}/cycle}$$

Thus there are three ways to increase a program's performance: reducing the number of dynamic instructions in the program, increasing the clock frequency, and increasing the number of instructions that can be retired in a cycle. For a uniprocessor, the number of instructions depends on the instruction set architecture, which is usually fixed for general purpose computing, and the compiler, which presumably performs all the reasonable optimizations. For a multiprocessor, the number of instructions may increase due to the overhead of partitioning and distributing the workload, as well as any additional communication and synchronization that becomes necessary. The second term is clock frequency, which is the same for all programs, so it is often left out of performance measurement and the execution time is expressed in number of cycles. This leaves the number of instructions retired per cycle, which is referred to as IPC on a uniprocessor. The upper limit of IPC is the instruction issue width of the processor. That is only attainable if every issue slot is filled in every cycle and the pipeline never stalls. Unfortunately, neither condition is realistic. A program often cannot fully utilize all the issue slots because of dependencies on instructions already in the pipeline. These dependencies are broken into three classes. There are structural dependencies, where a hardware resource needed is being used by another instruction. There are control dependencies, which arise when there is a branch instruction in the pipeline whose outcome and/or target address has not been resolved. Lastly, there are data dependencies, where the result of an in-flight instruction is needed as an operand to this instruction. In a program, the parallelism of independent instructions that can be issued together is called instruction level parallelism (ILP).

These dependencies can severely limit performance because of long latency operations, most commonly memory operations and branches. In architectures with caches, the latency of a memory operation depends on whether the data resides in a cache and the access latency

of the caches and memory. A hit in the first level cache does not cause any pipeline stall, whereas a miss that requires an access to main memory may take hundred of cycles. Thus both cache miss rates and memory operation frequency are important statistics for understanding how the memory system performance affects IPC. Note that in some processors, memory store operations do not stall the execution pipeline (they are called non-blocking stores). Moreover, an out-of-order processor can re-order instructions and execute instructions that follow an outstanding memory operation but are independent of it. Therefore, these architecture features have to be considered when interpreting the statistics. For branch instructions, modern processors use branch prediction and branch target buffer (BTB) to execute speculatively past an unresolved branch instruction. However, there is a performance penalty when the prediction is wrong, as the speculative instructions are squashed, or when the target address is not cached in the BTB, in which case the early stages of the pipeline stall until the address is computed. Hence, both the branch mis-prediction rate and the BTB miss rate are indicators of performance degradation due to poor branch prediction.

When we move to multiprocessors, the equation above still applies but the IPC is now the number of instructions executed by all the processors in the machine. If the application uses multiple processors, the effective IPC is the sum of the IPC for each of the processors. Control and data dependencies are handled by explicit synchronization operations, which lower the available parallelism. The synchronization operation halts a processor for some time, which lowers its average IPC. In addition to dependencies, the distribution of workload across processors also affects the effective parallelism. An uneven distribution results in some processors completing their jobs before others and sitting idle while there is still computation to be done. Again the processors that are idle have lower IPC and thus they lower the overall performance. A similar situation occurs during parts of the program that are not

parallelizable, when only one processor can be utilized. The coverage of parallelizable code thus sets the upper limit of performance gain achievable with parallel execution. Aside from these parameters that affect the level of parallelism extracted, there are additional variables that affect memory system performance in a multiprocessor. While the workload of the program is partitioned on to different processors, the data needed is not necessarily partitioned. Thus the size of the total working set of a parallel application is a critical parameter. The amount of data sharing between processors is also important, since it affects how much communication takes place. How these parameters change as the number of processors grows are important factors that influence how the parallel performance scales. In the following chapter, we provide an overview of probabilistic inference and provide preliminary information about these issues for our applications.

Chapter 3

Applications Overview

Before we introduce the set of applications that we use in our characterization effort, we need a basic understanding of probabilistic graphical models and how they are used for inference. We start with the different types of graphical models and how the graphs capture dependencies between random variables. Then we introduce the concept of inference and describe typical queries that can be computed. There are a wide variety of inference algorithms so it is impossible for us to cover all of them. Instead, we consider the different classes of algorithms and briefly describe some of the popular ones. The applications we use are drawn from these different approaches. Each of these applications is described in detail to provide the background necessary for the discussion of the characterization results in the following chapters.

3.1 Probabilistic Inference Overview

In the field of artificial intelligence, knowledge representation and reasoning is an important part of any intelligent "agent". Intuitively, knowledge representation allows us to encode

information about a particular system in a form that can be utilized by an algorithm to reason about the state of the system, which enables the agent to make good decisions on future actions without human intervention. One possible way to do this is to use first-order logic to specify facts and rules that an agent follows to derive the current beliefs and desirable actions. The disadvantages of this logical approach are that it is impossible or too expensive to specify all possible conditions an agent may encounter and in practical situations an agent does not have access to the ground truth of all the facts it needs to know. In other words, there is uncertainty in both the model of the system and the perception of the physical environment. There have been various approaches to reasoning under uncertainty, including default reasoning [15], certainty factors [2] and fuzzy logic [16], but the only one that is used widely nowadays is probabilistic reasoning. The idea is to encode knowledge as probability distributions over random variables that model the system of interest. The relationship of how these variables affect each other is captured in a graph, which allows a compact representation. Given a graphical model, the basic task is inference – that is to infer information about some unobserved variables in the system. Inference is a building block for more advanced tasks such as learning and planning, and inference algorithms often form the cores of probabilistic reasoning applications.

3.1.1 Graphical Representation of Probabilistic Models

A graph is a natural representation of probabilistic models because it provides an intuitive way of visualizing the relationships among variables and also is an efficient way to capture all the information. If one were to use a table to store the joint distribution over N binary variables, it would require 2^N entries. Using the structure of a graph to encode correlations allows one to factor the distribution into localized representation where the size scales with

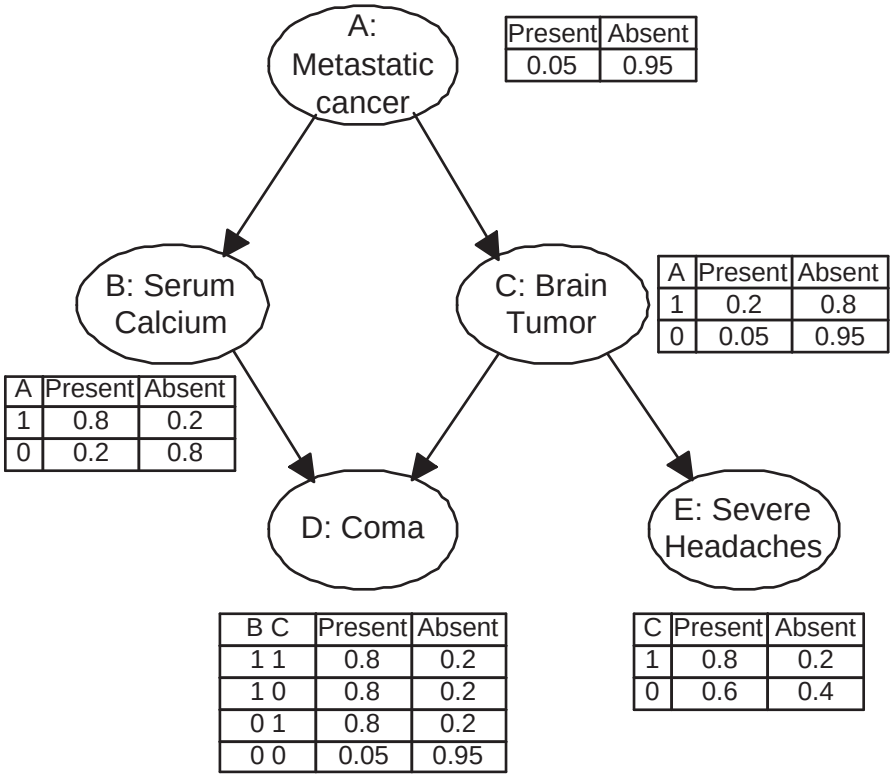

Figure 3.1: A simple Bayesian network

the in-degree of the node. The two most common graphical representations are Bayesian networks and Markov networks.

A Bayesian network [17, 18] is a directed acyclic graph in which nodes represent random variables and directed edges encode directed influences between nodes. A simple example is shown in Figure 3.1. Instead of spelling out the entire joint distribution over all variables, a Bayesian network encodes independence assumption between variables in the graph structure. Each node is associated with a conditional probability distribution (CPD) that captures the probability distribution of that variable conditioned on the values of its parent nodes. This structure encodes the assumption that a node is conditionally independent of

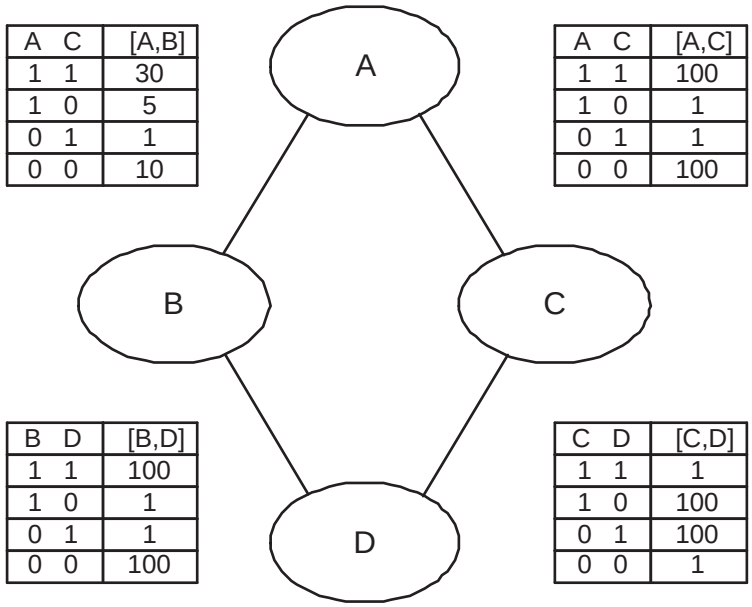

Figure 3.2: A simple Markov network

its non-descendants given its parents, and enables efficient representation of probability distribution over a large number of variables. For example, this network assumes that whether a patient has a severe headache depends directly only on whether a brain tumor is present. And if we already know that a tumor is present, observing whether cancer is present or whether the patient is in a coma does not change the likelihood of severe headaches. Now using this kind of graphical model, one can answer different types of questions about the joint distribution. The simplest kind of question is about the conditional probability or posterior probability given a set of observed nodes. Another type of question is to find an assignment to a set of variables that maximizes the posterior likelihood or the joint likelihood of the query variables. A key property of Bayesian networks is that the independence assumption is directional. For example, in Figure 3.1 nodes B and C are independent given node A, but dependent given both nodes A and D.

To express a mutual independence assumption, one can use a Markov network [19, 20],

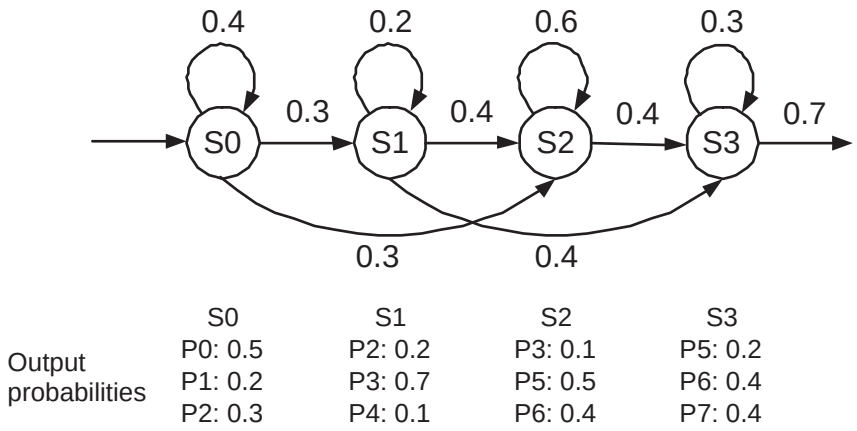

Figure 3.3: A simple Hidden Markov Model

also called Markov random field, which is an undirected graph. Similar to Bayesian networks, the nodes in a Markov network represent random variables and edges represent dependence between nodes. In the simple Markov network shown in Figure 3.2, B and C are independent given A and D, and vice verse. Because the relationship between connected nodes are mutual, parameterization is achieved via factors that are associated with cliques of nodes. Each factor is a function that maps each possible assignment to the variables in the clique to a value. Unlike Bayesian network, a factor does not necessarily describe a distribution and the values in a factor do not have to correspond to meaningful or well defined quantities. Inference on the joint distribution represented as a Markov network involves the use of a normalization constant to convert a product of factors into a distribution.

In addition to the models described above, there is a class of models called Dynamic Bayesian networks (DBN) [21, 22, 23] that are used for temporal processes. The Hidden Markov Model (HMM) [24] is the most often used example and in fact it predates the general DBN models. It is used to model the state and output of a dynamic system that follows the Markov assumption, which states that the current state only depends on the previous state

and the current input (or observation), and the current output only depends on the current state. Figure 3.3 shows a simple HMM. In its graphical representation, there is one node for each possible state of the system. A directed edge indicates valid transition between two states and is associated with a probability. Additionally, a probability distribution is defined over the possible outputs in each state. Since the state is often not directly observable, they are called Hidden Markov Models. Here, the type of questions we can answer with the model include "what is the probability of being at state X at time t", and "what is the most likely sequence of state transitions that lead to being at state X at time t".

3.1.2 Types of Inference

The goal of probabilistic inference is to answer questions relevant to the joint probability distribution represented by a graphical model. As we briefly touched on in the previous section, there are a few kinds of questions that we can answer with inference. Common types of queries include conditional probability, maximal a posteriori and most probable explanation. The simplest is conditional probability, which is to derive the value of $\Pr(\mathbb{Y} \mid \mathbb{E} = e)$, where $\mathbb{Y}$ is a set of query variables and $\mathbb{E}$ is a set of evidence variables. For example, using the network in Figure 3.1, one can inference the probability that a patient has metastatic cancer if severe headaches are experienced (i.e. $\Pr(A \mid E = 1)$). Another type of queries is to find the most probable assignments to a subset of the unobserved variables. Maximum a posteriori (MAP) and most probable explanation (MPE) are two variants of this type. In MAP, we want to find the values to a subset of the non-evidence variables ($\mathbb{Y}$) that would maximize the conditional probability of the assignments given the evidence ($\mathbb{E}$), i.e. solving for y such that $\Pr(\mathbb{Y} = y \mid \mathbb{E} = e)$ is maximized. This is useful, for instance, in diagnostic applications, where we are interested in the most likely combination of events

that would have the observed effects. MPE is a special case of MAP in which we are trying to maximize the overall probability of assignments to all variables, i.e. solving for y such that $\Pr(\mathbb{Y} = y, \mathbb{E} = e)$ is maximized and where $\mathbb{Y} \cup \mathbb{E}$ is the complete set of variables. MPE is considered as a separate type of inference since it is simpler than MAP and there are algorithms for MPE that cannot be applied to MAP.

There are two major types of inference algorithm: exact and approximate. As the name implies, exact inference algorithms (e.g. [25, 26]) computes the exact solutions to the queries. While these algorithms exploit the graph structures to do that efficiently in general, they have exponential time complexity[27], as the computation required is exponential in the number of variables in the largest factor or cluster. Consequently, they become intractable for large graphs. In those cases, approximate inference algorithms may be used to produce results that are only estimates of the true joint distribution, but are nonetheless useful in practice. Approximate inference algorithms take a variety of approaches but they can roughly be classified into two categories. In the first category, the algorithm conceptually uses a distribution that is easier to inference, which provides an approximation of the desired distribution. Many algorithms in this group are based on message passing (e.g. [28]). The second category includes applications that take a sampling approach and perform inference on a set of samples or instances instead of on the graph itself (e.g. [29, 30, 31]). For temporal processes, exact inference is often too expensive in practice because the space cost can become prohibitively large. Fortunately, the approximate methods for stationary models apply for temporal process as well.

3.2 Description of Applications

Given that there are so many different algorithms for probabilistic inference, it is impossible to cover them all in our characterization effort within a reasonable time frame. Instead, we select a set of applications that span the broad variety of approaches. Since most real world applications utilize approximate inference algorithms, we include only one exact inference algorithm in our set. It is called Recursive Conditioning and is a recently developed any-space any-time algorithm that solves for conditional probability queries on Bayesian networks. The other algorithms in our set are approximate algorithms that are applied to real problems. From the class of message passing algorithms, we have two applications that are based on Loopy Belief Propagation. One classifies web pages and the relationships between linked pages, and the underlying graph has a structure similar to the way the web pages are connected. The other solves a stereo vision problem and the graph in this case is always a four-connected 2-D grid. Then we have an algorithm based on importance sampling that track a robot's location on a map using data from the robot's sensors. Finally, we include a popular speech recognition engine's search and decode stage, which is a well-known application of Hidden Markov Model. In the rest of this section, we describe each application in greater details and give a high level view of the algorithmic process so that we can reason about how they map to a Smart Memories like multi-core architecture in the next chapter.

3.2.1 Recursive Conditioning (RC)

Recursive conditioning [32, 33] is an exact inference algorithm for Bayesian networks. It differs from earlier exact inference algorithms in that it allows a fine-grain trade-off between the amount of memory it needs and its runtime. This algorithm is based on the conditioning technique, which is a way to simplify exact inference. By instantiating certain variable(s),

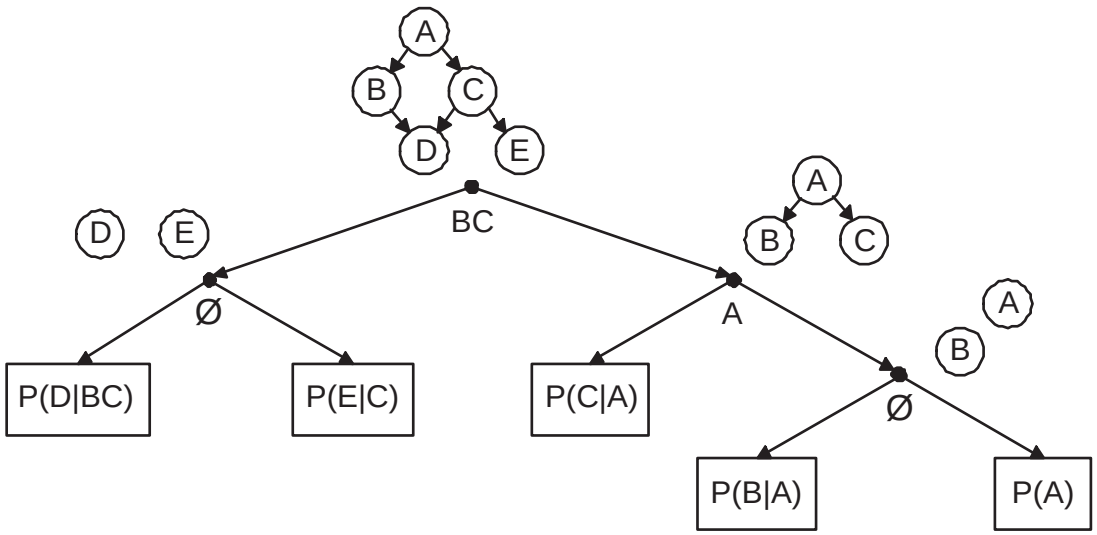

Figure 3.4: A d-tree for a five-node Bayesian network. Internal nodes are labeled by the conditioning variable(s) of that step and leaf nodes are labeled by the local CPD's in the Bayesian network.

i.e. assigning a value to an unobserved variable as if it is an evidence, it is possible to simplify the network structure by breaking dependencies. Inference is then performed on the simpler network for each possible value assignment to the conditioning variable(s) and the sum of all those results is the answer to the original query. The idea of recursive conditioning is to use conditioning to decompose a Bayesian network into two independent sub-networks recursively until we end up with individual nodes. This recursive process is encoded by a binary tree called a decomposition tree (d-tree), in which the internal nodes store the variables being instantiated at each step, and the leaf nodes correspond to the nodes in the original network with their respective local probability tables. Inference is then done by performing a recursive depth-first traverse of the d-tree.

Figure 3.4 shows an example of how a simple Bayesian network with five variables (A through E) can be decomposed recursively and the corresponding d-tree representation. The first decomposition step is to break the loop by conditioning on nodes B and C, which

separates the top part of the network (A, B & C) from the two disjoint nodes D and E. The sub-network of A, B & C can be furthered decomposed by conditioning on A. After that, only individual nodes remain and no more conditioning is needed. However, it is necessary to break the disjoint nodes apart with another level of branching in the d-tree, as each leaf node contains a single CPD. To perform inference, the algorithm traverses this d-tree recursively. At each internal node with a set of variables to be conditioned on, it iterates over all possible enumeration of the variable assignments. For each of those assignments, it records the assignment as an instantiation of the set of conditioning variables and then traverses the two subtrees. When the algorithm reaches a leaf node, a look-up operation is performed in the CPD using the set of instantiations recorded along the path. The returned values from the left and right branches of an internal node are then combined in a sum-of-product fashion. Sometimes, a subtree will be repeatedly computed under the same instantiation. In this example, the subtree of node P(A) and P(B|A) is traversed for each combination of values for variables A, B and C (since variables B & C are set at the root node). But changing the value of C does not affect the results computed in this subtree. Therefore, one can cache the results of various A & B instantiations the first time they are computed and re-use them later. The flexibility of varying how much caching is used differentiates recursive conditioning from other exact inference algorithms, as it allows a fine-grain trade-off between memory requirement (for the caches) and runtime (for recomputing results not cached). The pseudo code of this algorithm is shown in Figure 3.5.

3.2.2 Web page & Link Classification (WLC)

This is a program that classifies web pages and predicts relations between those pages [34]. It is one of two applications that are based on an approximate inference algorithm called loopy

```
RC (t, inst) {        // t: a d-tree node; inst: instantiations

    if (t is a leaf node) {
        return (table[index(inst)]);
    } else {
        if (t.cache != NULL && t.cache[index(inst)] != NULL) {
            return t.cache[index(inst)];
        } else {

            p = 0;
            for (each instantiation i of variable(s) to be conditioned at t) {    // loop parallelism
                inst.add(i);
                // task parallelism
                p += RC(t.left, inst) * RC(t.right, inst);
                inst.remove(i);
            }
            if (t.cache != NULL) t.cache[index(inst)] = p;
            return p;

        }
    }

}
```

Figure 3.5: Pseudo code of recursive conditioning

belief propagation (LBP). Originally, belief propagation was a message passing algorithm proposed for exact inference in networks that are polytrees, i.e. there is only one undirected path between any two nodes. Intuitively, a message encodes the "belief" of the sending node on its posterior (i.e. conditional on the observed variables) probability distribution. The receiving node uses this information to adjust its own belief and update the messages it sends. In a polytree, a two-pass algorithm can ensure that the information from every node is propagated across the graph by first propagating messages from all the leaves to a common root and then propagating messages from the root in the opposite direction. After these two passes, all the beliefs would have converged to the exact posterior probability distribution. It was later discovered that this algorithm can compute posterior probability distribution estimates for some loopy networks by iteratively propagating messages and

Loopy_belief_propagate (*nodes*, *edges*):

```
while (beliefs have not converged) {
    foreach n1 (nodes)      // phase 1
        foreach n2 (<n1,n2> in edges)
            message'_{n1→n2} = Σ_{n2} belief_{n1} / message_{n2→n1} × potential_{<n1,n2>}
    foreach n1 (nodes) {    // phase 2
        belief_{n1} = prior distribution
        foreach n2 (<n1,n2> in edges)
            belief_{n1} = belief_{n1} • message'_{n2→n1}
            message_{n2→n1} = message'_{n2→n1}
    }
}
```

Figure 3.6: Pseudo code of loopy belief propagation algorithm (as applied to pairwise Markov networks)

updating beliefs. Even though there is no guarantee that the beliefs will converge or that the beliefs will converge to values close to the true posterior distribution, for many problems in practice LBP is able to produce good estimates. The number of iterations required for convergence is likewise unpredictable, but it is partly dependent on the diameter of the graph as that affects how many iterations are needed for information to propagate across the graph.

The pseudo code of LBP is shown in Figure 3.6. The details of message computation and belief update are specific to pairwise Markov networks, in which only pair of directly connected nodes have factors defined, since both WLC and the following application to be described use such representation. A message from node A is a vector that encodes the current belief of node A's posterior probability distribution, which in turn is determined by its prior probability distribution and the messages node A has received from its immediate neighbors in the previous phase. For each edge, two messages are passed along it, one for

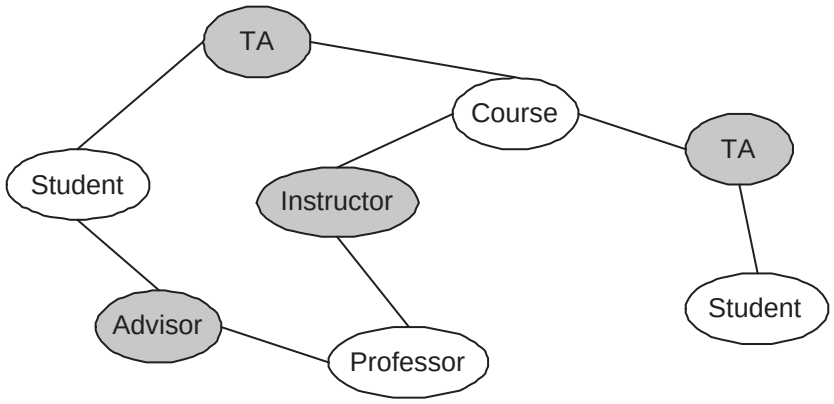

Figure 3.7: An example of the type of network used in WLC for web pages within the computer science department. White ovals are web pages and grey ovals are relationships between pages.

each direction. Each node first computes an outgoing message for each of its edges, in a sum-of-product fashion, based on its current belief vector, the last incoming message received along that edge and the potential matrix that encodes the pairwise factor. The last incoming message is needed so that we can exclude information passed from the recipient node from the outgoing message (otherwise the information is "double counted"). Each node then uses all the incoming messages to update its belief. This process is repeated until the change in each node's belief is below a threshold, at which point the beliefs are considered converged.

For the purpose of web page and link classification, the pairwise Markov network has two kinds of nodes: web page category and relationship category between two web pages. For example, web pages within the computer science department domain can be categorized as student, professor, course, research group and so on. In the same context, relationship types can be student-advisor, coarse instructor, member of a research group and so on. Each relationship node has exactly two edges that connect to the two related web pages, while a web page node can have any number of relationships. An illustration of the graph structure for this example is shown in Figure 3.7. Initial beliefs are determined based on keywords

contained in the web pages and in the links. Parameters used for initialization as well as the pair-wise potential defined for each web page-relationship pair of nodes are previously learned from real data. LBP is applied to this graph and the beliefs after convergence give the most likely classification for both the web pages and the relationships between those pages.

3.2.3 Stereo

This is an application that computes the horizontal displacement of each pixel from a pair of gray-level stereo images [35]. Those values can then be used to generate a depth map of the scene captured in the images. Finding the most likely per-pixel displacement for the whole image is a labeling problem and can be modeled as a Markov Random Field (MRF). The disparity map is represented as a four-connected 2-D grid. Each node in the graph corresponds to a pixel in the reference image. An energy function is defined to describe the cost of assigning a displacement value (label) to a pixel:

$$E(x) = \sum_{(p,q)\in N} V(x_p, x_q) + \sum_{p\in P} D_p(x_p)$$

where N are the edges in the grid. The construction of the energy function is based on the assumptions that the displacements of neighboring pixels tend to vary gradually, and the corresponding pixels in the stereo images should have the same intensity. Thus, $V(x_p, x_q)$ is the discontinuity cost of pixel p having a displacement of x_p and pixel q having a displacement of x_q, while $D_p(x_p)$ is the matching cost of pixel p and its corresponding pixel in the second image given a displacement of x_p. The optimal labeling is obtained from minimizing the energy function, which can be done by LBP[1]. Besides stereo vision, this framework can also

[1] Another popular algorithm for this minimization problem is Graph Cuts [36]

be applied to other vision problems such as motion estimation and image restoration.

The max-product version of LBP (as opposed to the sum-product one used in WLC) can be used to find the MAP estimate of the MRF, which is an approximate solution to the problem of determining the assignments that minimize $E(x)$. To simplify the computation, the algorithm can be transformed from performing max-product in probability space to performing min-sum in the negative log probability space. Thus, messages are computed according to this formula:

$$m'_{pq}(x_q) = \min_{x_p}\left(V(x_p, x_q) + D_p(x_p) + \sum_{r \in N(p)\setminus q} m_{rp}(x_p)\right)$$

where m_{pq} is the message from pixel p to pixel q and $N(p)$ are the neighbors of p. Each message is a vector and its dimension is defined by the number of possible displacement values. The term $D_p(x_p)$ only needs to be computed once from the raw image data. Since $h(x_p) = D_p(x_p) + \sum_{r \in N(p)\setminus q} m_{rp}(x_p)$ is independent of x_q, the minimization over x_p is the same regardless of x_q. In Stereo, $V(x_p, x_q)$ is defined to be $\min(s\|x_p - x_q\|, d)$, where s is a constant and d is the upper bound of the discontinuity cost. This allows the message computation to be done in three steps. The first step is to compute $h(x_p)$ and $\min_{x_p} h(x_p)$. The next step is to compute $f(x_q) = \min_{x_p}(s\|x_p - x_q\| + h(x_p))$, which can be done using a two-pass algorithm over the range of x_q. Finally, the message is computed simply as

$$m'_{pq}(x_q) = \min\left(f(x_q), \min_{x_p} h(x_p) + d\right)$$

At the end of LBP, the belief vector for pixel p is $b_p(x_p) = D_p(x_p) + \sum_{r \in N(p)} m_{rp}(x_p)$ and the value x_p that minimizes $b_p(x_p)$ is the approximate solution p.

Since the graph structure is fixed, a couple application specific optimizations are possible.

As described above, each node updates its belief with the messages from its immediate neighbors. In a four-connected 2-D grid, if we color the nodes in a checkerboard pattern of red and black, red nodes can only send messages to black nodes and vice verse. Thus having the messages from red nodes allow us to compute all messages from black nodes in the next iteration. These messages in turn allow us to compute all the messages from the red nodes in the iteration after next. Therefore, it is possible to update only half the messages in each iteration and achieve convergence at the same speed. Another optimization, generally called multi-grid, is aimed at speeding up the convergence of belief propagation through a hierarchical approach. The idea is to start with a coarse version of the graph and perform belief propagation to obtain a rough solution, which is then used to initialize the messages for belief propagation in the finer version. Starting from the original graph, each coarser graph is constructed from the previous level by combining four nodes (in a 2 $\times$2 block) into one. The data cost $D_p(x_p)$is defined as the sum of the data costs of the four nodes being combined. LBP is first run on the coarsest graph and the resulting messages at the end of the run are used as the initial messages for LBP on the graph one level finer. This in effect reduces the number of hops it takes for some information to propagate across the length of the image at the coarser level and thus helps speed up convergence. In fact, using multiple levels and operating in a coarse to fine fashion enables the algorithm to achieve a good quality solution with a small fixed number of iterations per level (ten iterations per level for our experiments). This also means that we do not have to test for convergence after each iteration.

```
Robot_localize_time_step (odometry, sensor):

    Compute robot motion parameters (angle and distance of movement) from current and last odometry

    for each particle p    // loop 1
      Update particle pose (p.x, p.y, p.theta) using trigonometric functions with robot motion parameters

    for each sensor reading (angle, range)    // loop 2
      Compute location of the point of reflection (s.x, s.y) relative to robot

    for each particle p    // loop 3a
      for each sensor point (s.x, s.y)
        Look up log-probability of detecting obstacle at sensor point relative to particle pose from map
    for each sensor point    // loop 3b
      If a large fraction of log-probabilities are improbable, mark sensor point as masked
    for each particle p    // loop 3c
      p.weight = sum of log-probabilities for all sensor points that are not masked

    if robot has traveled more than threshold distance
      for each particle p    // loop 4
        cumulative_weight_sum = cumulative_weight_sum + p.weight
      for each new particle p    // loop 5
        Generate new p by sampling a random number up to cumulative_weight_sum
    Compute mean and standard deviation of pose from set of particles
```

Figure 3.8: Pseudo code of robot localization algorithm

3.2.4 Robot Localization

Sampling importance resampling (SIR) belongs to a class of approximate inference algorithms called particle filters. Particle filters are used to estimate the posterior distribution of a partially observable Markov process by recursively updating a set of particles, which are sampled instances of the system state, using new evidence that becomes available over time. One of the many applications of particle filters is robot localization, where probabilistic models are used to tackle noisy sensor data and robot control. For example, the Carnegie Mellon Robot Navigation Toolkit (CARMEN) [37] uses SIR to estimate the location and orientation of the robot with a set a weighted particles [38].

The main data structure in this application is the array of particles. Each particle is consisted of a state, in this case the robot pose (x, y, theta), and a weight. In addition, the

program reads in a map of the environment that the robot explores. The map provides a probability for each discrete 2-D location that indicates the likelihood of a sensor detecting an obstacle at that location. The pseudo code of one time step of the robot localization algorithm is shown in Figure 3.8. The algorithm takes two sets of input: the odometry and sensor range readings (e.g. laser and sonar). The odometry specifies the where the robot believes it has moved to since the last time step and this information is used to update the states of the particles. The sensor range readings, along with the map, are used to determine the probability (hence the weight) of the particles. In each time step, the value and weight of each particle is calculated according to the odometry and sensor data respectively (loops 1-3). After the robot has moved a certain distance, which is a configurable threshold, the set of particles are resampled (loop 4-5). The new samples are drawn according to the probability distribution specified by the current weights. To do this, imagine an array of particles where the "size" of each entry is equal to the weight of the particle. Normally SIR draws a new set of N particles by randomly sampling N positions along this array. In this CARMEN implementation, the particles are generated from a low-variance walk starting from a random position and sampling N equally spaced positions. For each position, the array entry it falls into determines the values of the new particle. The result of this resampling process is a set of equal-weight particles, in which the number of particles representing a particular state is proportional to the original weight of that state. At any time step, the estimated location and orientation of the robot are the weighted means of the current set of particles.

3.2.5 Speech Decoding

The last application is the search and decoding phase of the Sphinx III speech recognition engine [39]. The front end of Sphinx III processes the acoustic signal and computes Gaussian

mixture scores according to the acoustic model. Since this part of the processing is a general DSP task and has been well studied, we focus on the back end that decodes the speech using a dictionary, a phonetic model and a language model. Phonemes are the basic units of sounds that compose all the spoken words and a phonetic model specifies what they are and how they sound like. The algorithm uses a tri-phone model where each phoneme has a left and right context. Each phoneme is modeled by a Hidden Markov Model (HMM), which, in the phonetic model we use, has 3 sub-phoneme states plus an exit state. The chains of phonemes that comprise words are organized as prefix trees, called lexicon trees. Figure 3.9 shows a snippet of a lexicon tree starting with the phoneme "S". The roots of these trees are all the possible initial phonemes of words included in the dictionary used. Words that share the same initial chain of tri-phones share those internal nodes in a lexicon tree. Each leaf node is unique and each path from the root to a leaf node in a lexicon tree corresponds to a distinct word. On top of the lexicon trees that encode the phoneme-to-phoneme transition, a language model governs the word-to-word transition. The language model specifies the likelihood of certain word combinations, e.g. the phrase "there is" is a lot more probable than "there it". We use a tri-gram model, which means phrases included in the model are at most three words long. Since most random word combinations are highly unlikely to appear in a speech, only a small portion of those that warrant a boost in probability are specified in the language model. Putting these altogether, the Viterbi algorithm [40] is used to compute the most likely path of traversal through a series HMM's given a sequence of acoustic scores (one for each audio frame of 10ms). In each audio frame and for each possible next state, the Viterbi algorithm computes the log probabilities, or the scores, of all the valid paths that lead to the new state using the score of the originating state and the transition probability. It picks the path with the highest score and records the path in the

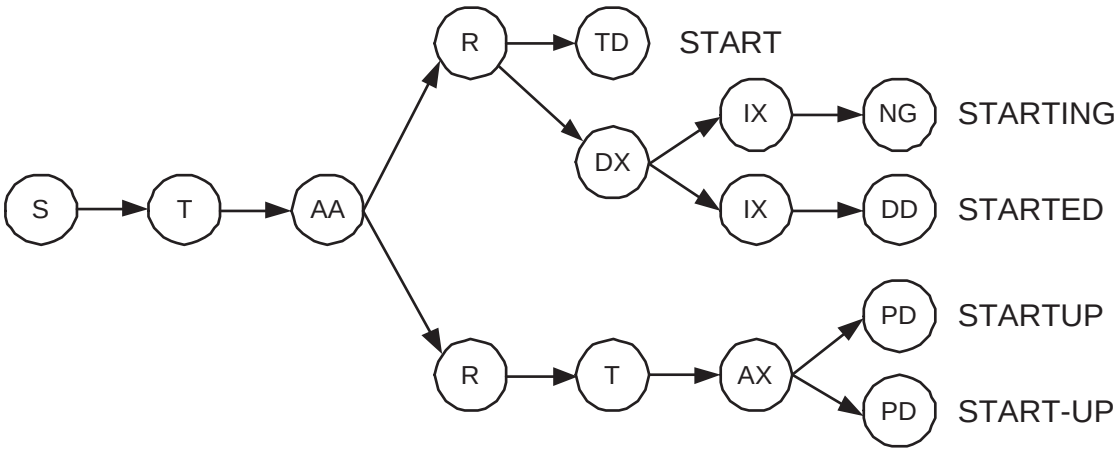

Figure 3.9: A snippet of a lexicon tree

new state. Thus, numerous Viterbi paths are pursued simultaneously during the decoding process. The Viterbi algorithm can be depicted as a trellis, as shown in Figure 3.10. The Viterbi path that has the highest score at the end is backtracked to yield the most likely sequence of words.

The pseudo code of the decoding process is shown in Figure 3.11. For each audio frame, the front end of the speech recognition engine computes a vector of scores for all HMM's given the acoustic signal. The backend has two main phases, evaluation and propagation. In the evaluation phase, each active HMM evaluates the probabilities of all possible transitions using the acoustic score corresponding to that HMM and chooses the path with the best score. The best score among all the active HMM's is used as the reference for the HMM level pruning. In the propagation phase, the algorithm considers each active HMM and determines whether it should be pruned or whether any child HMM needs to be activated. When a HMM transits to the exit state, the exit path score is passed on to the next HMM as the initial score. When the exit occurs at the last phoneme of a word, the cross-word transition probability is based on a trigram language model and is incorporated in the Viterbi path

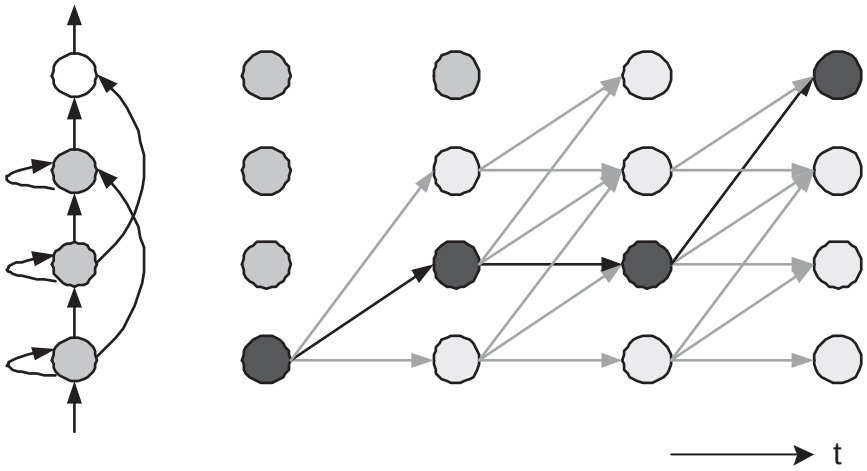

Figure 3.10: A graphical representation of the Viterbi algorithm

score when the subsequent word transition is recorded. There is a separate threshold for word level pruning. Word transitions that are above the word pruning threshold are recorded in a backpointer table, which is an array of records, each storing the word recognized, a pointer to the predecessor word's backpointer table entry, and the path score leading to this word. If too many entries are created, there is additional pruning to eliminate some entries. The backpointer table essentially stores the entire Viterbi history. At the end of decoding, the most probable path can be traced back from the final backpointer table entry.

3.3 Sequential Execution Profile

Before we discuss the parallelism in the probabilistic applications and their mapping to a multi-core architecture, it is useful to examine their performance on a modern uniprocessor, as this gives us hints on the architectural factors that have the biggest impact on performance. Conventionally, benchmarks come in two varieties. The "integer" programs are not very compute intensive and have irregular control flows and data access patterns. On the

```
Speech_Decode_one_frame (lexicon_trees[ ], viterbi_history):

    Obtain acoustic scores from frontend

    for each lexicon_tree in lexicon_trees[ ]    // HMM evaluation
        for each active HMM
            Update score of each state using acoustic scores and determine best score for this HMM

    if number of active HMMs > maximum_allowed_per_frame
        Determine histogram pruning thresholds
    else
        Use default thresholds

    for each lexicon_tree in lexicon_trees[ ]    // HMM propagation
        for each active HMM
            if HMM score is above threshold
                Mark as active for next frame
            else
                Reset HMM states
            if HMM is not a leaf node in lexicon_tree and HMM exit score is above threshold
                Compute entrance score for each child HMM and activate those whose scores are above
                thresholds
            else if word exit score is above threshold
                Update viterbi_history using probability for this word with respect to all possible predecessor
                words
    Prune entries from viterbi_history

    Find best word exit for each distinct word-final phone and determine which lexicon_tree to use for
    word transition
    for each distinct word-final phone
        Activate or update root nodes with matching left context in lexicon_tree
    De-allocate memory used for temporary data and reset per-frame book-keeping data
```

Figure 3.11: Pseudo code of the backend of the speech recognition software Sphinx III

other hand, the "floating-point" programs are more predictable in terms of branch outcomes and tend to make sequential and strided memory accesses. To find out if the applications outlined above fit into one of these general types, we profiled the sequential execution of each program on a 1 GHz Pentium-III (Coppermine) processor. Pentium III is a superscalar out-of-order processor that can retire up to three operations per cycle. It has two 16KB 4-way associative level 1 (L1) caches for instruction (I) and data (D), and a 256KB 8-way

associative unified level 2 (L2) cache. Figure 3.1 shows several notable characteristics for these applications: IPC, percentage of memory, floating-point and branch operations in the dynamic instruction stream, percentage of branches that are mis-predicted or incur a BTB miss, the L1 data cache miss rate, and the global L2 cache miss rate (i.e. L2 miss rate out of all data accesses). While there is a fair amount of variation in the statistics across the applications, we observe a few patterns. With the exception of RC, the IPC count and branch behaviors are closer to integer applications like those in the SPEC CINT benchmark suite. However, their memory system statistics are a lot more like floating-point applications such as those in the SPEC CFP suite. The high L2 miss rate suggests that any working set that is not captured in the L1 cache does not fit within a 256KB L2 cache and indeed the major working set ranges from just under 1MB to over 30MB in our experiments. An L2 miss requires an off-chip main memory access that typically takes 50 to 150 cycles, so L2 misses account for most of the performance degradation in Stereo, Robot Localization and Speech Decoding. Branch mis-predictions and BTB misses are another source of performance loss. In the Pentium III, the penalty for such events is 15 cycles on average (with a 11 cycles minimum). Thus, WLC's IPC is mostly limited by its frequent branch mis-predictions. For Stereo, Robot Localization and Speech Decoding, the performance loss from branches is significant but is secondary to the effect of L2 misses. The combination of moderate to high memory access frequency and frequent mis-predicted branches in these applications make it unlikely that their IPC, hence performance, will improve in future uniprocessor architectures. Therefore, we believe any performance benefit from advancement in circuit technology has to come from explicitly parallel implementations.

Application	IPC	Memory op	FP op	Branch	Mis-prediction & BTB miss	L1 D miss	Global L2 D miss
RC	1.56	42.9%	3.11%	8.2%	1.6%	1.5%	0.25%
WLC	1.06	34.0%	6.14%	17.5%	26.9%	0.8%	0.6%
Stereo	1.03	32.8%	14.0%	10.5%	9.0%	2.1%	1.26%
Localization	0.58	45.5%	20.73%	7.2%	23.6%	1.1%	1.04%
Speech	0.47	46.5%	N/A	9.7%	15.7%	4.7%	3.23%

Table 3.1: Sequential execution characteristics

3.4 Summary

We have presented an overview of probabilistic graphical models and how they incorporate knowledge with uncertainty. We have also introduced the concept of inference and described five probabilistic inference programs that are based on different approaches to solving the relevant queries:

- Recursive Conditioning (RC) – an exact inference algorithm that allows fine-grain trade-off between computation time and memory space
- Web Page & Link Classification (WLC) – LBP on an irregular graph
- Stereo – modified LBP on a four-connected grid
- Robot Localization – particle filter
- Speech Decoding – Viterbi search on HMM's

Lastly, we have examined the sequential performance of these programs on a modern monolithic microprocessor and observed that they are not suitable for out-of-order processors designed to extract implicit ILP to improve performance. Instead, we look at how to exploit parallelism explicitly in the next chapter.

Chapter 4

Characterization of Parallel Performance

This chapter looks at how well the probabilistic applications described in Chapter 3 map to a chip multiprocessor architecture like Smart Memories. While there has been in-depth work in using some parallel scientific applications to benchmark parallel machines [41, 42, 43], there have only been a few attempts to characterize the performance of parallel probabilistic applications [44, 45, 46, 47]. We attempt to take a broader look at this space, by exploring at a range of applications, and trying to understand the fundamental limitations in how these applications' performance will scale on future machines. Our first task was to determine how to extract parallelism from these applications.

There are three different levels of parallelism: instruction, data and task. Section 3.1's sequential execution profiles of these applications show that they do not have large amount of instruction level parallelism that can be extracted. While some of these applications have task level parallelism, there is not enough to scale beyond a few processors. The most

abundant parallelism in these applications is data parallelism, which is usually extracted either using threads or streams/vectors.[1] With threads, each thread is a separate instruction stream processing a subset of the data. They execute independently except when explicit synchronization operations bring them to a rendezvous. On the other hand, streams and vectors are Single Instruction Multiple Data (SIMD) constructs where the same instruction is executed on different pieces of data at the same time. SIMD works best for algorithms that have regular memory access patterns and few data dependent operations within the core loops. Since most of these applications do not have the properties for streams/vectors to have a performance advantage over threads, we use a thread programming model for ease of parallel implementation.

We first describe the source of thread level parallelism in each application. In cases where multiple plausible implementations exist, we discuss each option and compare their pros and cons. Then we present the analysis of the scalability, communication to computation ratio and temporal locality of the applications. We round up this chapter by discussing how these characteristics may change as the probabilistic algorithms evolve and as the datasets grow.

4.1 Background

Our approach to characterization is mainly influenced by the work on the SPLASH [41, 42] benchmarks. This suite of parallel programs was assembled for architecture studies of shared memory multiprocessors (SMP) and have since been used for performance evaluation on a variety of parallel architectures. They include both integer and floating-point applications and kernels from the scientific, engineering and graphics communities. These benchmarks are

[1] Note that data parallelism can be converted into ILP, as in dataflow architectures. However, the scalability of this approach has not been proven.

optimized for medium to large scale SMP's (8-128 processors) and were thoroughly analyzed for a wide range of architecture configurations. We use a similar set of characteristics as in [42] to examine the computation, memory and communication requirements of parallel probabilistic inference algorithms.

More recently, the SPEC organization has developed SPEC OMP [48], a parallel benchmark suite based on the OpenMP application programming interface (API) [49]. The suite is composed of most of the programs in SPEC CFP2000, which is a suite of floating-point scientific and engineering programs, and an integer program that uses a genetic algorithm. It is designed to provide a portable set of tests for evaluating commercial SMP's. As such, there have been several studies on their performance characteristics on commercial servers (e.g. [43, 50, 51]).

While probabilistic inference algorithms have been popular in many applications, there has not been a lot of work on parallel implementation of these algorithms and evaluation their parallel characteristics. In [44], Kozlov and Singh considered different ways of implementing the clique tree algorithm, an exact inference algorithm, in parallel. Their evaluations showed that a static implementation that exploited only in-clique parallelism and was optimized for data locality achieved good speedup on shared memory multiprocessors for networks with various structures. Namasivayam et. al. [46] presented a different parallel implementation of the clique tree algorithm that exploits topological parallelism and showed that the algorithm is scalable to 256 processors. Liu et. al. [45] parallelized the learning algorithm for Module Network, which is an extension of Bayesian network where variables that have similar set of dependencies are grouped into the same module and dependencies (hence edges) are defined between modules. Lastly, Chen et. al. [52] assembled a set of data-mining bioinformatics benchmarks that include three programs based on hill-climbing

structural learning of Bayesian networks. They analyzed the parallel performance of these benchmarks on a 16-way SMP and found two of the three only achieved speedups of 6 and 8 because of load imbalance and high level 2 cache miss rates. Given all initial characterization, these results are not surprising but we are interested in looking a little deeper at these issues. These studies focused on one particular algorithm and/or one particular architectural characteristic. In contrast, we try to present a systematic characterization of a range of applications that are based on various probabilistic algorithms.

4.2 Parallel Implementations

In this section, we describe the inherent thread level parallelism in the applications introduced in Chapter 3 and discuss implementation options. As mentioned earlier, all the algorithms have large amounts of data parallelism. To take advantage of this parallelism, we need to divide the data into smaller chunks that have similar workload and distribute them to the threads. As we will see, the two key issues that limit effective parallelism is being able to evenly divide the work and minimize the necessary communication. In Stereo, Robot Localization and Speech Decoding, it is straight forward to partition the data. In RC and WLC, there are multiple plausible approaches and these will be discussed in greater details in later subsections.

4.2.1 Applications with Simple Partitioning

Stereo

In each iteration of belief propagation, the messages to be computed for each node depend only on the messages from the four immediate neighbors. Since the algorithm alternates

between two halves of the nodes divided by a checkerboard pattern, those messages from the neighbors are always computed in the previous iteration. Hence there is no dependency between any messages being computed in the same iteration and they can all be computed in parallel. Moreover, each message (and by extension each node) requires the same amount of computation, so we simply need to divide the nodes equally among the threads. The 2-D grid structure of the graph leads naturally to partitioning it into rectangular blocks. This minimizes the communication required at the fringe nodes of each block.

Robot Localization

This algorithm consists of a series of loops that iterate over the set of particles, the sensor readings or both. Each weighted particle is an independent sample of the system state and thus can be updated in parallel. Each sensor reading is also an independent piece of evidence. Thus, each loop can be parallelized by simply dividing the iterations among threads. Barrier synchronization is required between parallel phases of the algorithm and for a few reduction operations (to compute sums or maxima).

Speech Decoding

The decoding algorithm is driven by a set of lexicon trees, which are composed of HMMs. There are two types of lexicon trees: unigram trees for dictionary words and filler trees for non-word noises like coughing and "UH". There are multiple copies of each type of tree for word exit transitions to allow for integration of the language model. The number of lexicon trees is small to limit the computation and memory requirements – in our datasets there are three of each type. Thus there is not enough parallelism at the lexicon tree level. This leaves the parallelism at the HMM level where there are typically thousands of active

HMMs. Evaluation and propagation of HMMs are independent of each other and so can be done in parallel. Synchronization is necessary when updating global counters and the backpointer table that records the Viterbi history, and to prevent simultaneous updates to the same HMM (which is only possible when an active HMM is the child of another active HMM). Since the lists of HMMs are constantly changing, we simply distribute them among the threads in a round-robin fashion. Each lexicon tree keeps track of its list of active HMMs so for simplicity we parallelize the process within each tree. It would be more efficient to parallelize across the aggregate set of HMMs and we discuss this issue more later in the chapter.

Partitioning data in the other two applications is more complex and they are described in detail in the following sections.

4.2.2 Recursive Conditioning

In this algorithm, there are two main sources of thread-level parallelism. One source resides in the loop that implements the conditioning part of the algorithm (loop parallel implementation). For each conditioning variable, the posterior probability is computed for every possible value of that variable. This corresponds to multiple traversals down the same part of the d-tree. As an example, we refer to Figure 3.4. The set of conditioning variables at the root node contain B and C. If both are binary variables, there are four possible combinations of variable assignment and therefore four d-tree traversals from the root node. In this implementation, an intermediate result that will be cached may be accessed by more than one thread. Without any synchronization, that result may be redundantly computed multiple times by different threads simultaneously since these threads might get to that node of the tree before the first thread has computed the value to be cached.

Since this caching is critical to performance, one could use a task parallel decomposition, decomposing the Bayesian network into sub-networks that can be solved independently. This corresponds to solving non-overlapping sub-trees in parallel. Referring to Figure 3.4 again, the traversals down the left subtree containing nodes D and E can be carried out in parallel with the traversals down the right subtree with nodes A, B and C. In this approach, synchronization and communication are necessary when results from sub-trees assigned to different threads are combined. The big issue with this approach is load balancing. Balancing workload will be tricky, since d-trees are often imbalanced in number of recursive calls.

D-Tree Generation

There are many possible d-trees for a given Bayesian network, as there are many ways to decompose a graph. For example, another d-tree for the network in Figure 3.4 is shown in Figure 4.1. For sequential implementation, the optimal d-tree would be one with the minimum number of recursive calls. However, that d-tree may not be suitable for a parallel implementation due to the limit of loop parallelism or imbalance in the tree structure. For loop parallel implementation, a d-tree with a large root node that requires a large number of iterations is ideal, as that allow even partitioning for a large number of threads. For task parallel implementation, a d-tree that is more balanced is desirable as that simplifies load balancing. Note that the computation load is dependent on the cardinality of the variables, so it is the balance of the computation requirement across two sub-trees that is important and not that balance of the tree structure. In our d-tree examples, if all variables are binary, the d-tree in Figure 4.1 requires fewer recursive calls, but the d-tree in Figure 3.4 is preferred for both parallel implementations. For the performance comparison, we generated a large number of d-trees and chose the one that was most suitable for the implementation being

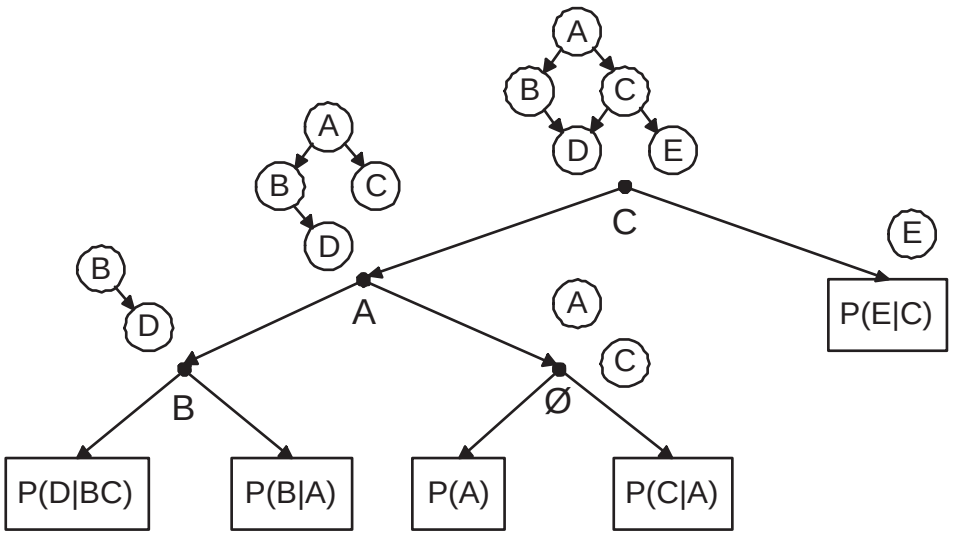

Figure 4.1: A different d-tree for the same example network shown in Section 3.2.1

evaluated and yet requires comparable number of recursive calls to the minimum, and thus is still an efficient solution.

Loop Parallel Implementation

The loop parallel implementation exploits the independence of computation for each instantiation. In this implementation, iterations of the loop in the root node are statically partitioned among the threads. Each thread maintains a partial sum for its share of iterations. These partial sums are then combined after the end of the loop, and synchronization is not required within the loop for correctness. The redundant computation of cached intermediate results by multiple threads in parallel results in reduced performance of the parallel application.

To eliminate redundant computation, fine-grain locks can be used to synchronize accesses to individual cache entries and prevent a thread from computing an intermediate result that is already being computed by another thread. This requires one lock per cache entry.

The lock is held while a thread is accessing or computing that entry, which prevents any other thread from attempting to recompute the same cache entry during the computation. Obviously, this lock converts some redundant execution time into synchronization stall. Fortunately, we can hide some of the synchronization stall time with multithreading, by running another thread on that processor. However, there is still overhead in acquiring and releasing the lock.

Task Parallel Implementation The task parallel implementation exploits the parallelism across branches of the d-tree. In this implementation, a thread spins off the computation for one of the two branches when there is an idle thread and the height of the current node is larger than a configurable threshold. Each thread iterates the whole loop, but only traverses down one branch. At the end of the loop, the parent thread waits for the child thread to finish in order to compute the intermediate result and continue the recursive process. Since balanced d-trees are often substantially more expensive in terms of computation required, we can improve load balancing by spinning off more threads in bigger subtrees or by having each thread switch to the opposite branch after a certain number of iterations, thus averaging the workload.

In Section 4.3, we compare the performance of the different implementations we just described and choose one for the characterization discussion.

4.2.3 Web page & Link Classification (WLC)

Like Stereo, WLC uses a message passing approach to estimate the solution of inference problem. Since the messages sent from a node depend only on the belief of that node and the messages it has received previously, the computation of messages from different nodes

can be carried out in parallel. Similarly, the update of individual belief is independent of each other. Hence we can parallelize the algorithm by dividing the nodes among threads. Unlike Stereo, WLC operates on irregular graphs, so the graph partition is not obvious.

Graph Partition Strategy

The way the graph is partitioned among threads affects both load balancing and the amount of communication. Consequently, a good partition strategy should balance the workload of sub-graphs while minimizing the number of edge-cuts. To achieve this, we can use the multilevel k-way partitioning program [53] in the METIS package [54]. This program computes a k-way partitioning of an irregular graph with weighted nodes and edges such that edge-cuts are minimized while balancing the sum of weights of the nodes in each partition. It first builds successively coarser graphs by collapsing adjacent vertices. Then it creates an initial partitioning of the coarsest graph, which is refined subsequently at each level. For our purpose of load balancing, we define the weight of each node to be the number of edges connected to the node, as the workload of a node is proportional to the number of messages it computes and receives. The edges have uniform weights since the amount of data communicated along each edge is the same.

Since partitioning only needs to be done once for our experiments, it is determined outside of the inference program and its computation time is not included in the results discussed in Section 4.3. As a percentage of the runtime of the inference program on one processor, the overhead of METIS varies between 1% and 6% for up to 32 partitions on the datasets we used. If dynamic partition is necessary, we could use a parallel version of the METIS partitioning algorithm [55] so that the overhead would not grow with the number of processors.

Synchronization Granularity

In the original version of the algorithm, all the nodes are updated exactly once in each iteration and the updates are synchronous, i.e. it appears that all nodes are updated in parallel using the data generated in the previous iteration. In the parallel implementation, this means using barriers to synchronize threads between iterations as well as between the message update and belief update phases. This is not required by the algorithm, however, which will converge to the same result regardless of the order in which the nodes are updated and the frequency of the updates. This allows a lot of freedom in orchestrating the execution and synchronization of the parallel threads. When a load balanced partition is not possible, we have the option to use asynchronous updates and allow each thread to iterate within its sub-graph at its own pace. This eliminates any processor stalls that would result from load imbalance, but may result in some processors performing useless work on a converged partition. Another effect is that a partition with a smaller workload would perform more frequent updates, which in some graphs may help propagate information faster. This implementation requires some fine-grain synchronization to control accesses to cross-partition messages to prevent race conditions. We achieved this by adding a fine-grain lock for each cross-partition message and requiring a thread to hold the lock while reading or writing the message.

4.3 Results of Characterization

Since the Smart Memories test chip has not been manufactured yet, we use the architecture level simulator for evaluation purposes. We describe the setup of the simulator as well as the base configuration assumed in the following section. Then we discuss three performance

characteristics that we think are most important for multi-core processors, as well as how they grow with data set size. The first one is scalability, where we look at the level of concurrency in the applications and limits to parallel efficiency. For RC and WLC, we also compare the different implementations described in the previously section. The second characteristic is temporal locality, which affects cache efficiency and thus the bandwidth pressure on the memory system. Lastly, we consider the overall communication requirement and the communication to computation ratio.

4.3.1 Evaluation Platform

The application performance will be evaluated in the Tensilica simulation environment [56] set up for the Smart Memories Project. In particular, we use the Tensilica instruction set simulator to execute the programs and collect relevant statistics. It is an extensible simulator that accommodates user-defined instructions and interfaces, which are used to simulate fine-grain synchronization and other custom memory system operations. It also has a protocol that enables simulation of multiple execution cores connected to each other and to a highly configurable memory system via the user defined processor interfaces. We use the compiler that is automatically generated for the simulator to compile the programs, which are all written in C/C++.

We have already covered the overall architecture of Smart Memories and its multithreading configuration in Section 2.3. For efficiency, the simulator models caches (which are the functions of the local tile memory) instead of individual memory mats and it models coherence operations at the granularity of messages instead of individual words/bytes. All components are configurable and the parameters for the base configuration are listed in Table 4.1. The parameters were set to match what we can reasonably expect on a real chip.

	Size	Latency (cycles)	Requests/cycle
L1 D cache	32KB, 2-way	2	1
L1 I cache	16KB, 2-way	2	1
Cache Controller	16 MSHR, 2 RSHR	Probe/Search: 2 Cacheline read/write: 6	1
Network	–	2	1
Memory Controller	16 MSHR	2	1
Main Memory	As needed	100	1/10 cycles

Table 4.1: Base memory system parameters for Smart Memories Simulator

We have not configured a L2 cache since having one would mean smaller or less associative L1 caches on Smart Memories, and the sequential profile in Section 3.3 indicates that the L2 cache would not be able to capture much of the data that misses in the L1 cache. Main memory's latency in a real machine varies depending on the memory access patterns. For the circuit technology that Smart Memories targets, the latency may range from 50 to 150 cycles. Since the performance gap between processor and memory has been growing as technology scales, the memory latency is only going to increase and thus any performance impact it has will be amplified. The cache controller follows MESI coherence protocol with load and store merging whenever possible. It also supports up to 10 non-blocking stores. For our analysis, we focus on the performance statistics that are collected after the initialization of the necessary data structures, since we are interested in how the core algorithms perform and the initialization phase typically has drastically different runtime behavior due to memory allocation and I/O operations.

4.3.2 Scalability

The scalability of an application is first determined by its level of concurrency, which in our studies is the thread level parallelism. Secondary factors that limit scalability includes load imbalance, parallel implementation overhead, communication and synchronization overhead.

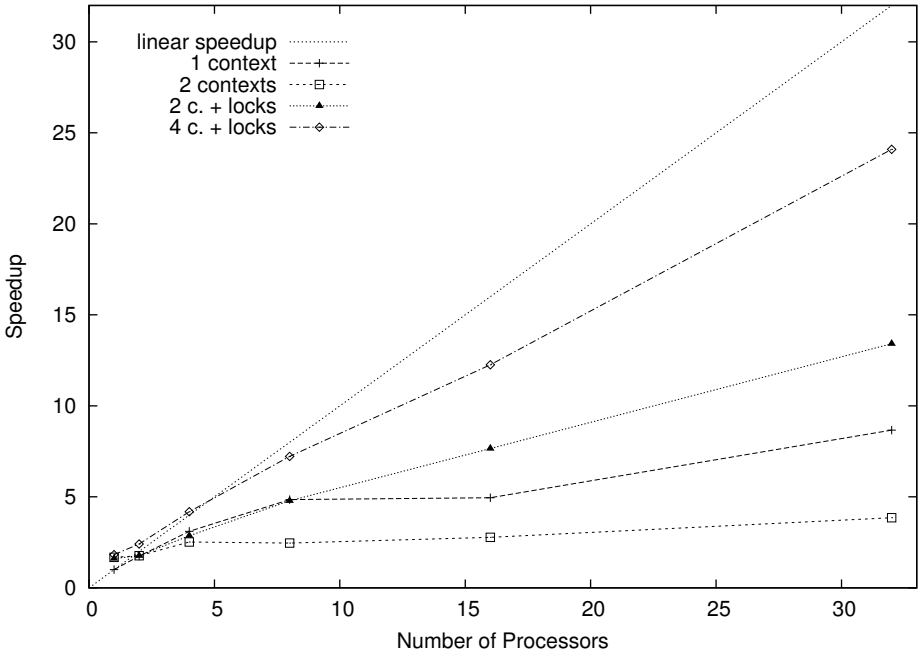

Figure 4.2: Performance of loop parallel implementations of RC

We will see how these factors interact and impact performance. We first discuss the two applications that have multiple plausible implementations: RC and WLC.

4.3.2.1 Recursive Conditioning

In the previous section, we have described two types of parallelization for RC: loop parallel and task parallel. Recall that in loop parallel implementation, we distribute the multiple traversals of the d-tree (from the root node) to all the threads. The parallel traversals have a potential drawback of redundantly computing an intermediate result that is cached and re-used in the original single-threaded execution. For this, we have proposed a variation where a fine-grain lock for each cache entry in the d-tree is used to ensure only one thread computes the result to be cached. Figure 4.2 shows the speedup (relative to the run time of the sequential implementation) of these two loop parallel implementations for the CPCS-360

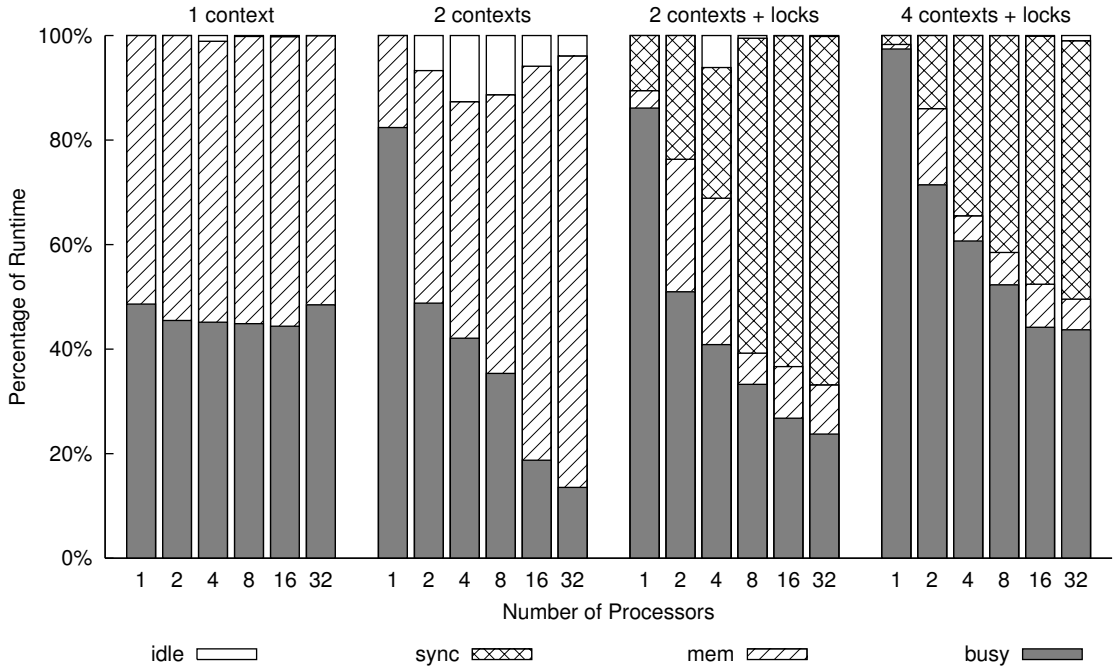

Figure 4.3: Runtime distribution of loop parallel implementations of RC

network and Figure 4.3 shows the corresponding normalized runtime distribution, which is the average percentage of time each processor is executing, is stalled on memory or synchronization operations, or is idle. For the main thread, it can only be idle when it is waiting for secondary threads to finish computation before it can proceed to a sequential section of the program. For the secondary threads, they are idle when they do not have a task assigned. The d-tree used employs full caching, which amounts to about 6.9MB of storage. Let's first consider the base version without locks. With only one context per processor (labeled "1 context"), the scales poorly beyond 8 processors. As observed in Figure 4.3, the runtime distribution stays virtually the same and does not offer an explanation for the drop in parallel efficiency. The culprit is the extra work that results from redundantly computing some cached entries. This amounts to 57% more instructions executed at 8 processors,

203% more at 16 processors and 279% more at 32 processors. The issue of redundant computation is also one of the reasons why employing two contexts per processors (labeled "2 contexts") causes performance degradation in all but the 1-processor configuration, instead of improving efficiency by reducing the effective memory stall time. The other reasons are the increased service time of memory requests due to higher demand and idle time due to load imbalance, as evident in Figure 4.3. The fact that load imbalance is a factor in the 2-context configuration but is negligible in the 1-context configuration warrants explanation. We partition the iterations assuming that each iteration requires the same amount of work, which is true only if no caching is done. With two threads sharing a processor, and since whether cache entries are redundantly computed or re-used is dependent upon the timing of the accesses, the likelihood of load imbalance is increased in this case.

For the loop parallel implementation with fine-grain locks, multithreading is essential to hide some of the synchronization stall time. With two contexts per processor (labeled "2 c. + locks" in Figure 4.2), the performance improvement is clear at 16 and 32 processors, where redundant work used to dominate the computation in the base version. The speedup is still not very good since there is a significant amount of synchronization stall that cannot be hidden. We can further reduce the amount of stall visible by increasing the number of contexts to four per processor. At the same time, we need to double the number of MSHR entries in each cache controller from 16 to 32 (such that there is at least one entry per thread) and increase the associativity of the data cache to 4-way to avoid a surge in conflict misses. (Further increase in MSHR buffers and associativity has little effect on performance.) With the boost in number of threads and in memory resources to support them, parallel efficiency is greatly improved and performance scales well to 32 processors. The 4-context speedup curve is a bit odd in that the growth from 16 to 32 processors is superlinear relative the

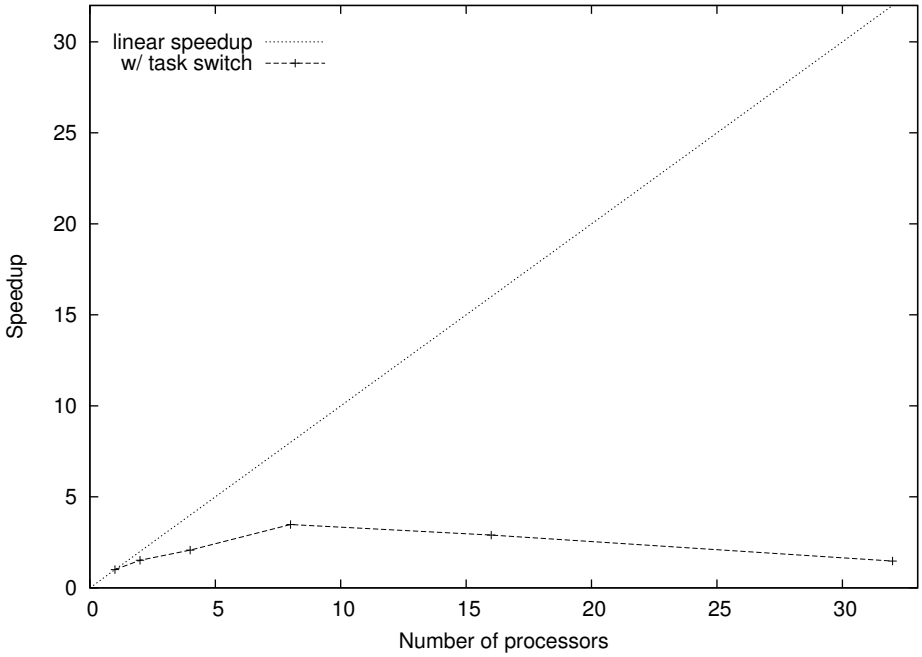

Figure 4.4: Performance of task parallel implementation of RC

growth up to 16 processors, which is the opposite of what one would expect. The reason for the unusual performance improvement comes from a lower miss rate in the data cache, due to constructive interference by chance, and thus there are fewer memory stall cycles.

The other type of parallelization for RC discussed in Section 4.2.2 is task parallel implementation. In this case, the partitioning of workload is done across the binary d-tree. The basic idea is that from any internal node, the traversal down the left branch and the traversal down the right branch can be carried out in parallel. Starting from the root node, the algorithm would spawn a new thread to traverse one branch while the original (parent) thread traverse the other. The intermediate results from the two branches are then combined when both threads return to the node the two branches join. If the amount of computation required for one branch is significantly more than the other, one thread will

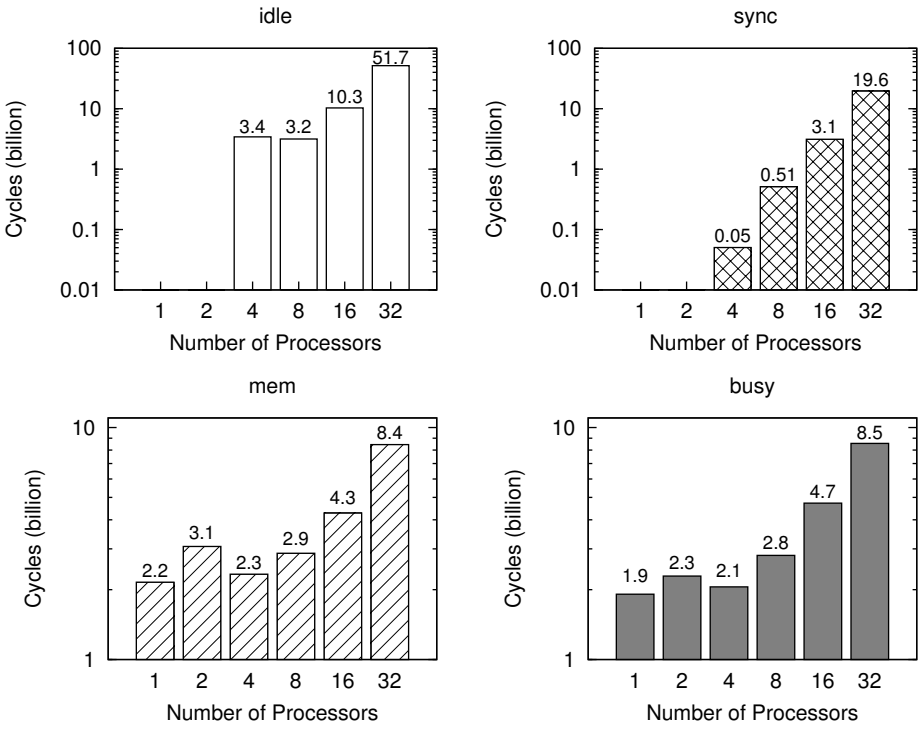

Figure 4.5: Breakdown of execution time for RC's task parallel implementation

have to wait for the other. Unfortunately, we found this to be the case in essentially all d-trees that are comparable to the most efficient d-tree in terms of total computation required. Consequently, we modify the implementation such that a thread switches from traversing one branch to the other after half of the iterations in order to balance the workload. We also make sure that more threads are spawned in the bigger subtree. This implementation has other issues however. Figure 4.4 shows the speedup and Figure 4.5 shows the runtime distribution (in total number of processor cycles) of this task parallel implementation for the CPCS-360 network [57]. There are a few factors that contribute to the poor scalability. One is that because of the task switching, the threads no longer traverse non-overlapping parts of the d-tree, which means redundant computation of cached results is possible. We observe significant increase in instructions executed due to redundant work for 8 threads

and above (45% at 8 threads to 388% at 32 threads). In this case, since there is already synchronization between each pair of threads that split ways from a node, adding synchronization to eliminate redundant work would drastically reduce the number of ready threads. Even though multithreading would be able to hide some of the stall cycles, without complex thread scheduling and/or migration it is difficult to evenly distribute ready threads among processors. Another problem is that even with task switching, there is still significant load imbalance between pairs of cooperating threads that results in idle time and synchronization stall time. This is also related to caching within the d-tree, as it saves some computation from certain iterations but not the other.

On the whole, exploiting loop parallelism within the root node is a scalable approach. The simple implementation that ignores sharing of cached intermediate results between threads can achieve close to linear speedup in d-trees where there is minimal sharing or when accesses to shared data are rarely made at the same time. When this is not the case, fine-grain synchronization combined with multithreading can recover most of the performance loss. From our experience, exploiting task parallelism in a d-tree is only mildly beneficial for the amount of parallelism available on a multi-core system. The difficulty in load balancing and the complexity of managing fine-grain tasks add to the parallelization overhead, thus reducing the efficiency of the algorithm. In contrast, the loop parallel implementation is simple and achieves much better performance. For this reason, we use this implementation for the rest of the analysis.

4.3.2.2 Web page & Link Classification

In Section 4.2.3, we discuss the parallel implementation of WLC in terms of partitioning and synchronization between partitions. Let's first confirm our conjecture that the METIS

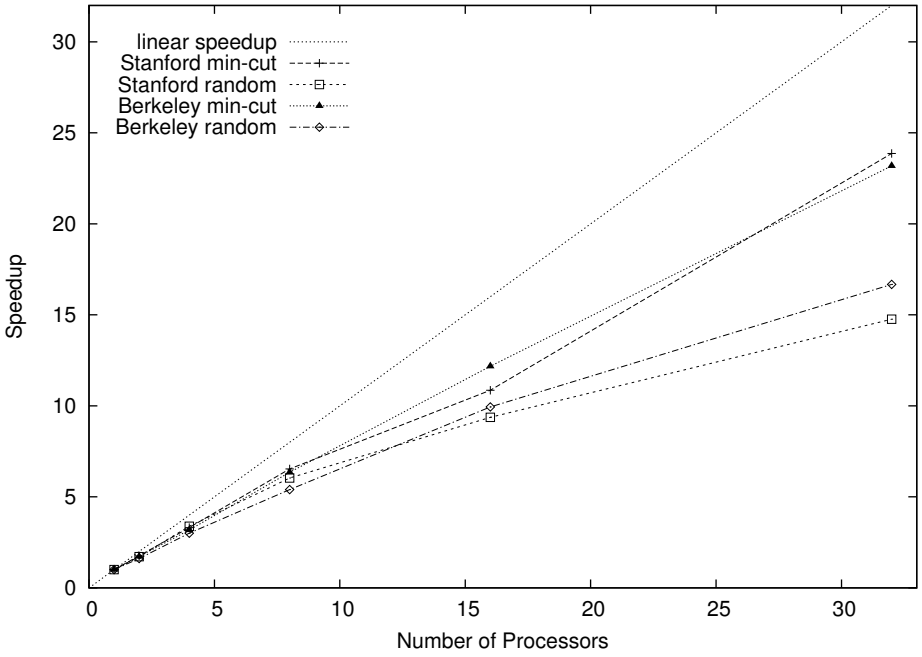

Figure 4.6: Performance comparison between weighted min-cut random partitioning in WLC

weighted min-cut partition algorithm is a good choice for partitioning the graph. To do that, we compare it against an algorithm that randomly assigns each node in the graph to any partition with equal probability. Messages and beliefs are updated synchronously as in the original serial implementation and barriers are used to synchronize threads between iterations. Figure 4.6 shows the speedup of WLC on the Stanford and Berkeley datasets (relative to the run time of the serial implementation) using the two different types of partitioning, running on single-context processors. The corresponding runtime distribution is shown in Figure 4.7. For all parallel configurations, using the weighted min-cut partitioning consistently outperforms the one with random partitioning. For small number of partitions, the difference is relatively small, but the difference grows as the number of partition increases. We can confirm that this performance difference is mainly due to load balancing by noting

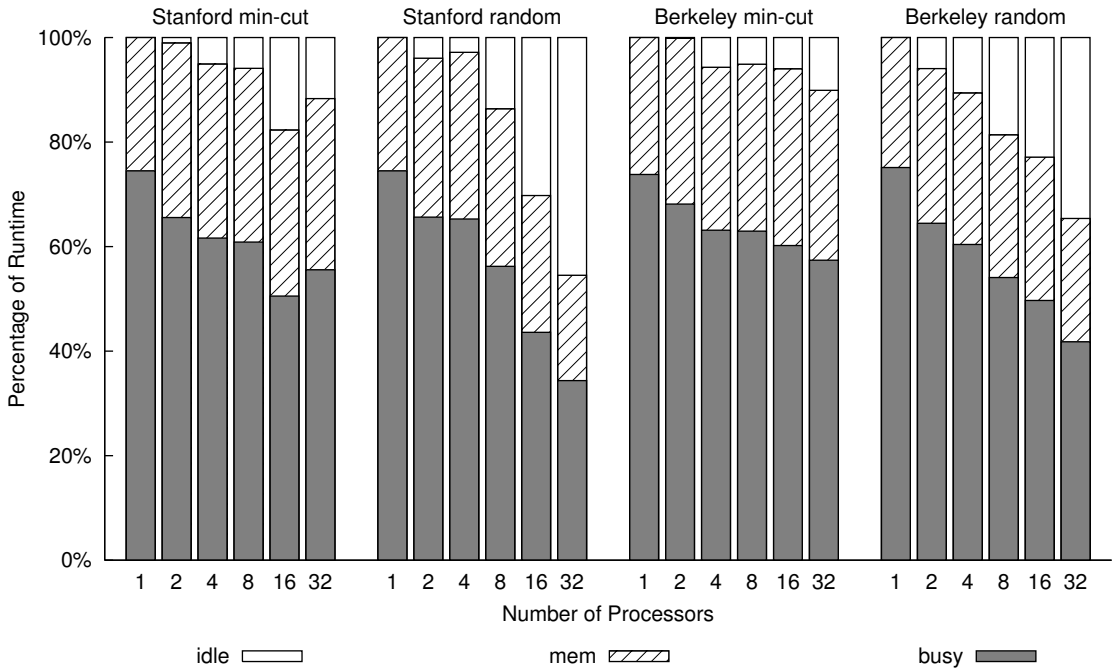

Figure 4.7: Runtime distribution of WLC using weighted min-cut and random partitioning

the increased portion of idle time in Figure 4.7.

While load balanced partitioning clearly has its advantage, an alternative synchronization strategy that removes the idle time – that is, the barriers – can reduce the performance loss due to imperfect partitioning. It may even improve the convergence behavior of the algorithm since it alters how often (relatively) updates are propagated. Recall that we do need to use fine-grain locks for cross-partition messages to ensure data integrity. Figure 4.8 shows the performance comparison between the "barrier" version using min-cut partitioning and the "fine-grain lock" version using random partitioning. Removing the barriers speeds up the algorithm significantly for the Stanford dataset at up to 16 processors but its performance for the Berkeley dataset is worse. In the case of the Stanford dataset, the performance gain is due to reduction in the number of iterations required to converge. Between 2 and 16 processors, the "FG lock" version requires 18 to 21 iterations (averaged

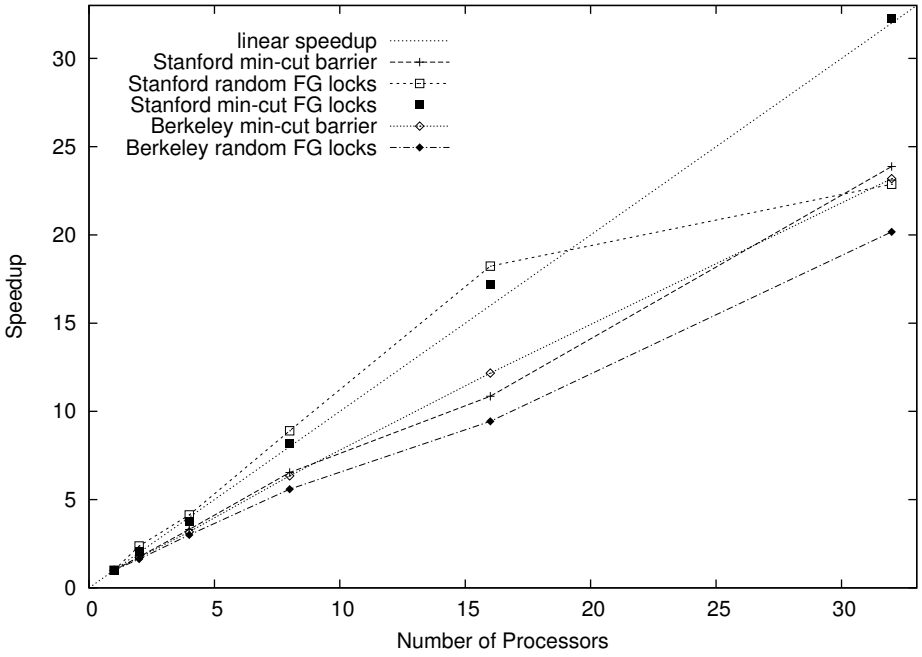

Figure 4.8: Performance comparison between implementations with and without barriers

over all threads) to converge as opposed to 25 in the "barrier" version. Unfortunately, more frequent updates do not always lead to shorter convergence time. This is the case at 32 processors for the Stanford dataset, when the average number of iterations increases to 30, and for all configurations for the Berkeley dataset, where the number of iterations goes up from 13 to between 14 and 18. In the Stanford scenario, using the min-cut partitioning with the barrier-free implementation lowers the average number of iterations to 20 and achieves linear speedup. However, the same combination fails to improve convergence speed for the Berkeley dataset. (For clarity of the graph, this data is not shown in Figure 4.8 since the numbers are very similar – but slightly worse – than those for "Berkeley min-cut barrier".) Whether a particular graph benefits from using the barrier-free implementation can only be determined by experiments since it is so far not possible to figure out the convergence behavior of LBP based on graph structure alone.

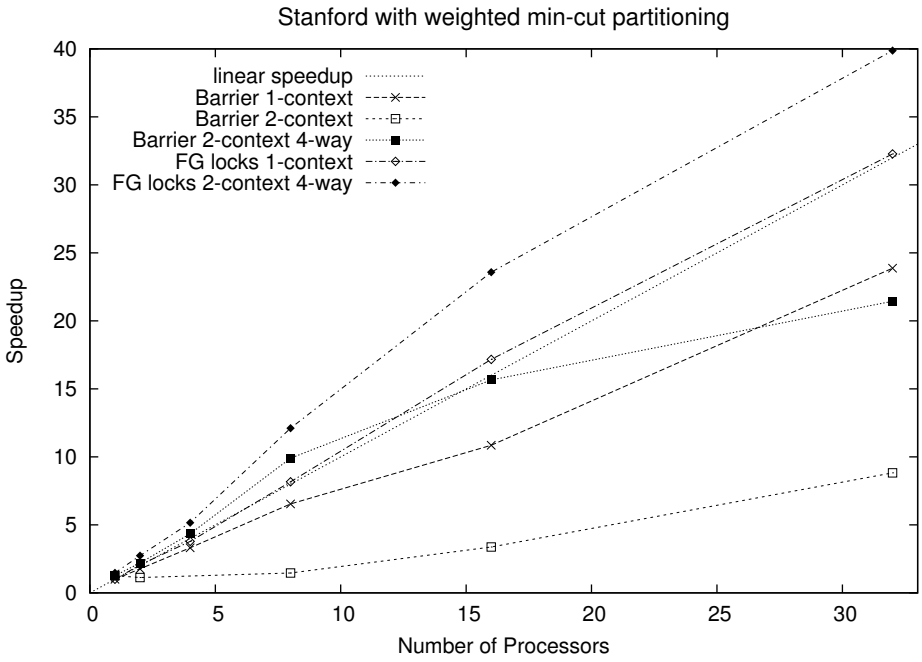

Figure 4.9: Performance comparison between 1- and 2-context processor configurations for both WLC implementations (4-way: a 4-way associative data cache; default is 2-way)

For both implementations, memory stall time accounts for a significant portion of the runtime. Again, we can use the hardware contexts support on Smart Memories to hide some of that stall time. If we simply switch to two-context processor cores though, the performance actually drops. The culprit is that having four threads on a tile sharing the same data cache causes the data miss rate to jump up due to conflict misses, and further increase the memory stall time. Thus, the two-context configuration only makes sense if we also reconfigure the data cache to double the associativity to 4-way. Figure 4.9 illustrates this effect clearly for the synchronous implementation (we show the Stanford dataset only since both datasets have similar behaviors in this regard). We also observe degradation in performance at 32 processors with the two-context 4-way configuration. The reason is that the weighted min-cut partitioning algorithm cannot achieve a balanced 64-way partition and

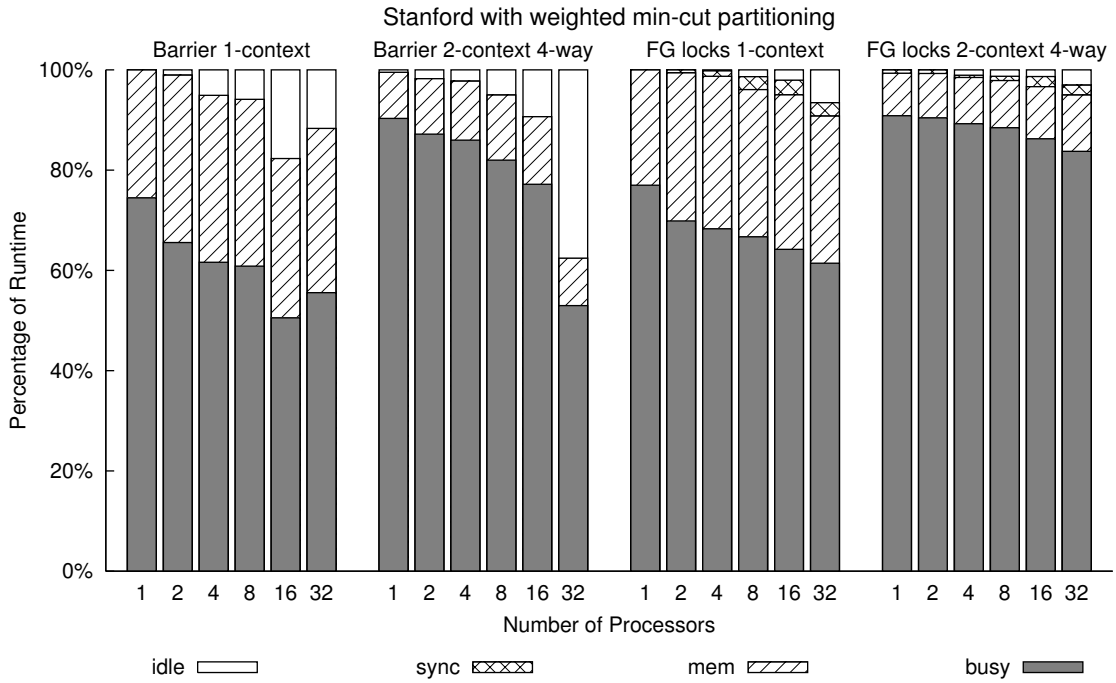

Figure 4.10: Runtime distribution comparison between 1- and 2-context processor configurations for both WLC implementations (4-way: a 4-way associative data cache; default is 2-way)

the load imbalance results in a dramatic increase in idle time, as shown in Figure 4.10. This problem does not exist for the asynchronous implementation with fine-grain locks, so the two-context 4-way configuration consistently improves performance as expected.

In general, using a good partition algorithm like weighted min-cut would give us predictable convergence time and speedup. Further speedup may be possible with the barrier-free implementation, but it is data dependent and is unpredictable. This implementation is most likely to be beneficial when a balanced partitioning is not possible as it avoids the idling of processors. In some cases, it also reduces the number of iterations required for convergence since information may propagate across the graph faster with free-running partitions. In any case, the effectiveness of this implementation can only be learned from experimentation.

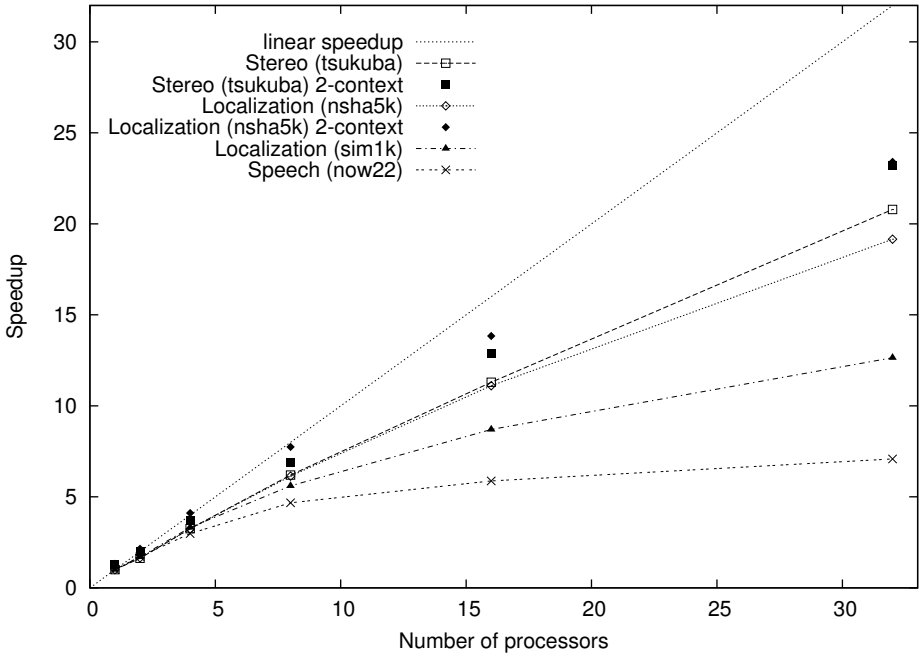

Figure 4.11: Performance of Stereo, Localization and Speech Decoding

4.3.2.3 Stereo, Localization and Speech Decoding

We now consider the three applications that use very simple data partition schemes: Stereo, Localization and Speech Decoding. The parallel performances of these applications are shown in Figure 4.11 and the corresponding runtime distribution in Figure 4.12. Two datasets are shown for Localization since the number of particles used in the filter has a significant impact on the parallel efficiency. There are 5000 particles for the dataset "nsha5k" and 1000 particles for "sim1k". We can see that Stereo and Localization-nsha5k have good scalability while Localization-sim1k and Speech Decoding do not perform well above 8 processors. For Stereo and Localization-nsha5k, their performance can be improved by using hardware multithreading to reduce the memory stall time, as shown in Figure 4.11. In the case of Localization-nsha5k, the two-context configuration also has the associativity

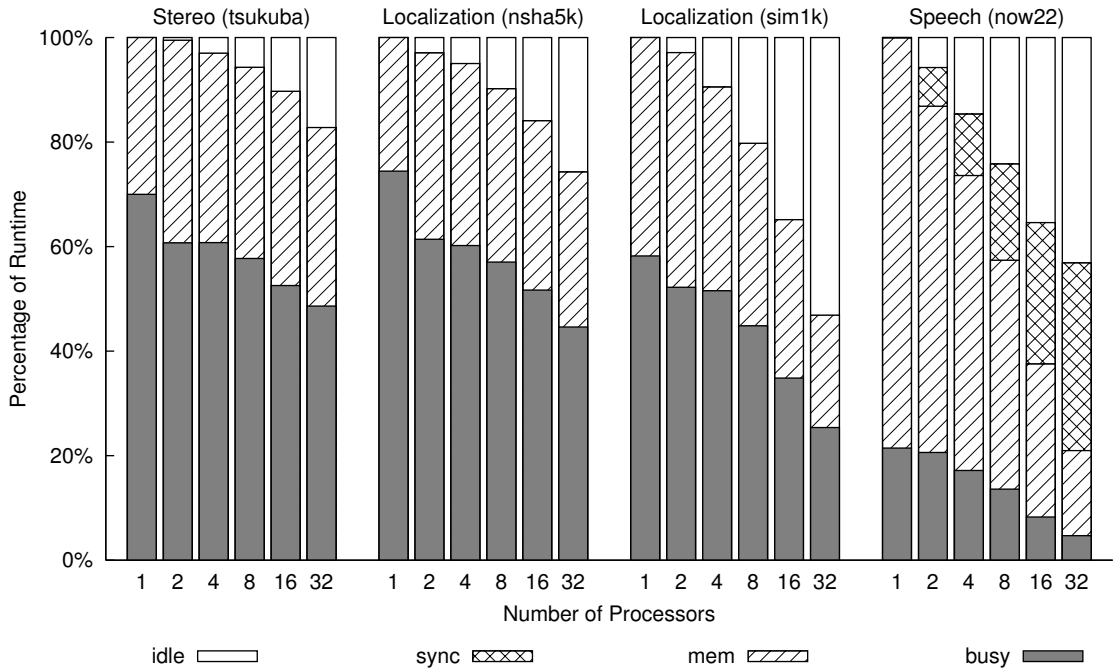

Figure 4.12: Runtime distribution of Stereo, Localization and Speech Decoding

of the data cache doubled to 4-way and the number of MSHR entries doubled to 32 so that the memory system would not become the bottleneck. Comparing to Localization-nsha5k, the runtime distribution of Localization-sim1k (shown in Figure 4.12) clearly points to idle time as the reason for its lower efficiency. In this application, the main source of idle time is sequential computation that is performed by only one thread in between parallel phases of the algorithm. These include several reduction operations, the processing of odometry data and the initialization for re-sampling. They account for about 3% of the single-thread runtime for "sim1k", which limits the ideal speedup to about 16 at 32 processors. In short, there is a time minimum for this application that is set by sequential sections. As the number of particles increases, more processors can be utilized effectively, until the execution time starts approaching this minimum.

In Speech Decoding, there are a few factors that combine to limit the performance.

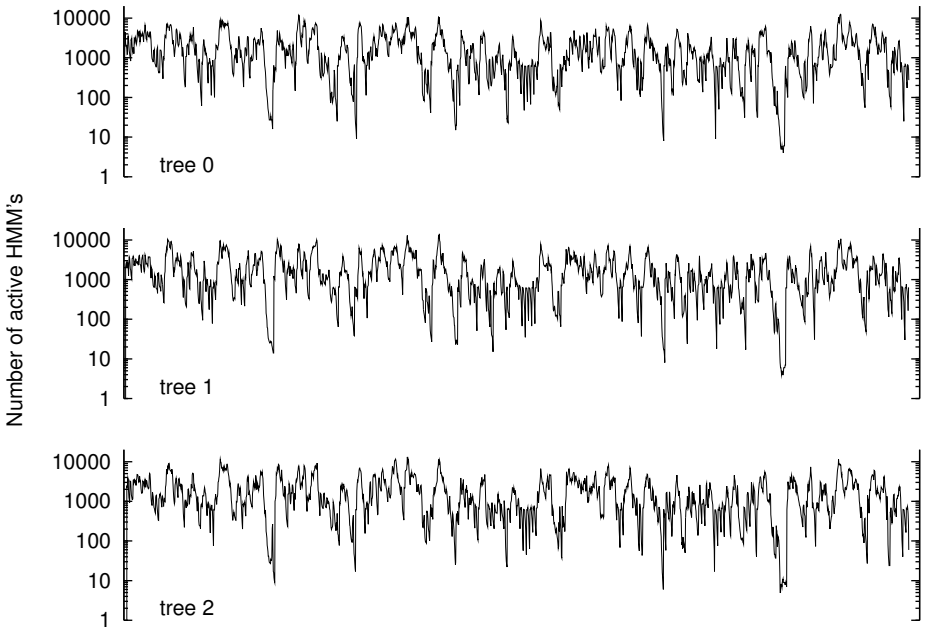

Figure 4.13: Variation in number of active HMM's for decoding of dataset "now22"

From Figure 4.12, we can see that the growth in both idle and synchronization stall cycles contributes to the poor performance. As we will see these limitations come from the way the algorithm was coded and not from a lack of parallelism of the application. These limitations can be divided into three categories: sequential computation, load imbalance and synchronization wait. The sequential sections of the program amount to less than 3% of the single-processor runtime but they contribute 31% of the idle cycles running at 32 processors. Load imbalance accounts for the rest of the idle time and most of that comes from the HMM propagation step and the word transition step. The HMM propagation step is also the source of most of the synchronization stalls. To determine if origins of these inefficiencies are inherent to the algorithm and how one may reduce the idle and stall cycles, we need to consider the program in greater details. Two-third of the sequential computation takes place in histogram pruning and Viterbi history pruning (26% and 39% respectively)

and the rest comes from memory de-allocation and other clean-up tasks. Taking clues from a hardware implementation of Sphinx [58], one way to parallelize pruning is to modify the pruning criteria. Instead of using the highest HMM/word/path scores from the current audio frame after HMM evaluation is complete, those from the previous frame are used. This means that pruning at all levels can be absorbed into the parallel evaluation, propagation and transition phases, which would eliminate half of the sequential computation. This also affects which hypotheses survive but is shown to have negligible effect in decoding accuracy in [58]. The HMM propagation step suffers from both load imbalance and synchronization stalls. The former is due to the additional processing required only for word-final HMM's that incorporates the language model (LM) into the path score. The imbalance is made worse by the simple scheme of parallelizing the computation within each lexicon tree because of the high variability in the number of active HMM's, as shown in Figure 4.13. To alleviate the load imbalance, the LM look-up step needs to be separated from the HMM propagation step and the lexicon trees should be combined for parallelization. The LM look-ups can be parallelized by assigning the HMM's from the same word (with different histories) to the same thread. This way of HMM distribution would also ensure that the threads would update different entries in the Viterbi history and would not need to synchronize. The remaining synchronization can be eliminated by pre-assigning new Viterbi history entries instead of allocating them as needed. The last major source of load imbalance is the word-transition step, where the best score for each word-final phoneme is determined and then used to activate the root nodes of a lexicon tree that have the matching left context. Since the number of word-final phonemes is small, there is not enough parallelism for the computation to be distributed evenly. One way to address this is to combine this phase with the LM look-up phase so that root node activation is done after path scores are computed. Some

synchronization would be necessary to prevent race conditions in modifying a root node but the critical section would be small.

4.3.2.4 Summary

Our results have shown that these applications indeed have plenty of data parallelism that can be exploited on a chip multi-processor system. With the exception of Speech Decoding, the data parallelism is easily extractable without much change to the program. Both Stereo and Robot Localization scale well to 32 processors with straightforward partitioning and simple barrier synchronization. In RC and WLC, scalability is achieved with the use of fine-grain locks and multiple hardware contexts per processor, to eliminate redundant work and improve load balance respectively. For Speech Decoding, the algorithm needs to be modified slightly and restructured in order to reduce the dependency that limits the available concurrency. This suggests that taking parallelism into account early in the algorithm development would be important for future applications.

4.3.3 Temporal Locality

As shown in Chapter 3, these applications have a large percentage of memory operations, so memory performance is critical. Temporal locality allows the re-use of data that is already loaded into the first level cache. An application with no temporal locality accesses each piece of data only once. In that case, any data needed has to be fetched from memory and the cache provides no benefit. Thus, temporal locality greatly affects the efficiency of the cache and an application's memory bandwidth and latency requirements. One often talks about an application's working sets, which is the set of data used during a certain phase of computation. How the major working set changes with the number of threads is

an important factor in determining how memory bandwidth needs to scale with the number of cores in a system.

4.3.3.1 Asymptotic Analysis

In RC, the major working set is the d-tree and all the threads traverse the entire trees in the loop-parallel implementation. Parts of the d-trees may be skipped if previously cached results are available. This is the case even in the sequential implementation though, thus in the best case the total working set remains unchanged. The reduction of the per-thread working set comes from the fact that each thread computes a subset of the computations that are recursively combined to generate the final result. Each thread therefore accesses only a subset of the entries within leaf nodes and internal nodes with caches. However, the reduction in the size of working sets is not proportional to the number of threads under a realistic assumption of multi-word cache lines, as the entries a thread accesses are likely to be dispersed across cache lines. As a result, each thread has only a slightly smaller working set than the original working set in the single thread implementation. Since this is a recursive algorithm, there is temporal locality in the part of the d-tree that it is traversing at any moment. Thus cache miss rate is low for this application.

Both Stereo and WLC iterate over a Markov network. The Markov network is partitioned among the threads so the working set of each thread scales down with the number of threads. In Stereo, the network is a four-connected grid and the per-thread working set size is inversely proportional to the number of threads. Partitioning is not trivial in WLC given the non-uniform graph structure and also depends on the partition algorithm. For the problem size we use though, the partitions are roughly even for up to 64 partitions and hence the per-thread working set size scales down roughly linearly as well.

The particle filter algorithm underlying Robot Localization mainly operates on an array of particles. There is also a temporary two-dimensional array used in weight computation. Both arrays are divided up into blocks and each thread always processes the same block. Obviously the size of this working set scales inversely as the number of threads. However, the set of particles are resampled occasionally and when that happens, the algorithm essentially generates and switches to a new array of particles. This means that even when all particles assigned to a thread can fit in the local cache, there is temporal locality only between resampling.

Speech decoding mainly operates on a set of lexicon trees, each of which is a tree of HMMs. For each audio frame, there is a list of active HMMs, which are distributed to the threads evenly. This does not necessary mean great temporal locality though, as the list of active HMMs changes every audio frame and is sometimes reordered mid-frame for pruning. As a result, there is not really a stable working set of HMMs. Moreover, the algorithm makes random accesses to a large language model that further reduces the overall temporal locality.

4.3.3.2 Empirical Analysis

We can observe all the effects of temporal locality discussed above when we vary the data cache size on each tile. Figure 4.14 shows the average data cache miss rate for 16 caches shared by 32 processors for data caches ranging from 16KB to 256KB in size (each). Note that the miss rate (Y axis) is in plotted log scale. We see that Speech Decoding as expected has the highest miss rate and does not benefit from larger caches because of its poor locality. In Stereo and Robot Localization using 5000 particles, we see very little drop in miss rate because the main working set does not fit in the caches even with 4MB of total caches on

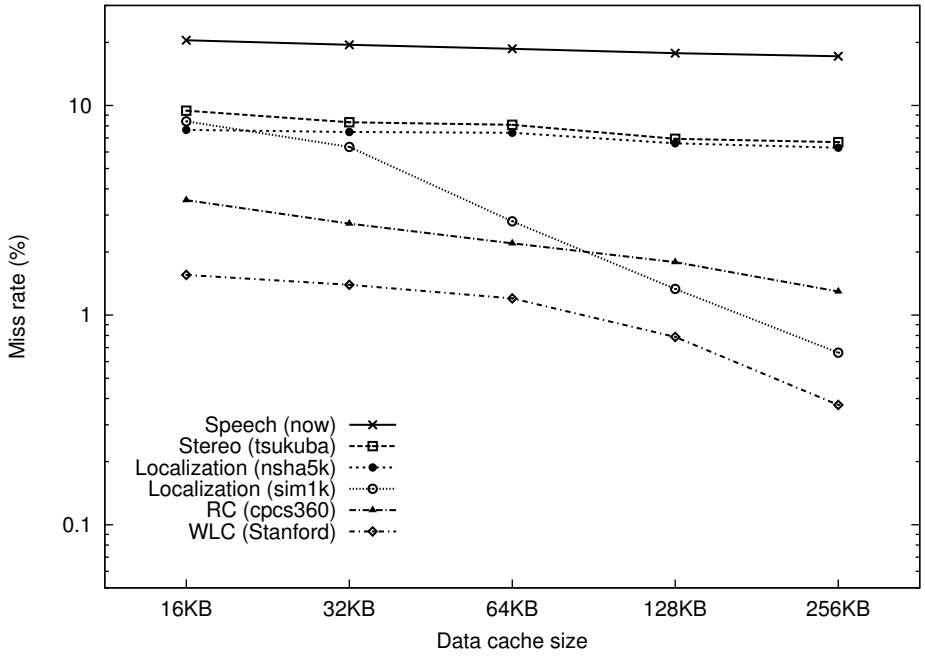

Figure 4.14: Effect of data cache size (per tile) on miss rate in a 32-processor configuration

chip. However, there is an obvious drop in miss rate in WLC and Robot Localization using 1000 particles, and the bend-over takes place right where the main working set starts to fit in the on-chip caches. Lastly, RC does benefit from bigger caches since there is temporal locality from the recursive traversal of the d-tree.

As we have observed, most of these applications have high data cache miss rates. This together with the fact they also have high memory operation frequency explains the large portion of memory stall cycles we saw in Section 4.3.2. On average, the number of instructions between data cache misses is small – 10 in Speech Decoding, 28 in Stereo, 58 in RC and Robot Localization, and 153 in WLC. Unlike traditional scientific parallel benchmarks that have high computation to memory ratio and regular memory access patterns, most of these applications either have low data re-use rate or have a lot of random memory accesses. Stereo is the only one that is likely to benefit from "blocking" – the technique of dividing

data into blocks such that a block fits in the level 1 cache and working on one block for a while before moving on to another one. However, it may be necessary to increase the number of iterations of belief propagation when blocking is employed, since blocking means that propagation between blocks happens less frequently than before. Although WLC can also be blocked in a similar way, the convergence time would probably be lengthened and the effect would be greater than that in Stereo since Stereo's multilevel approach allows global information to be exchanged at the coarser levels. Since these applications experience a lot of memory stalls, the memory system performance is more important than the computation performance. Thus, it is critical that the memory system has enough bandwidth and buffering resources to handle large number of outstanding cache misses. In addition, it is also desirable to have multi-context processors that can effectively hide memory latency. These hardware features significantly improve processor utilization and hence performance.

4.3.4 Traffic and Communication

We have already seen how temporal locality affects miss rate and thus affects the traffic to other caches and to the lower levels of the memory hierarchy. Besides memory accesses, communication between threads also generates traffic. As we scale the number of processors, how traffic and communication grows give us a sense of the bandwidth requirement of an application. This is important because bandwidth may be the limiting factor in an application's scalability.

4.3.4.1 Asymptotic Analysis of Communication to Computation Ratio

We first consider how communication and its ratio to computation grow asymptotically as the number of threads increases and then discuss the observed traffic growth in the next

section. We exclude RC and Speech Decoding from this section since the communication between threads in these two programs are dependent on the particular dataset it is processing and are therefore difficult to reason asymptotically.

WLC and Stereo

Both of these applications are based on LBP. In belief propagation, the amount of computation performed per thread scales with $\frac{E}{P}$, where E is the total number of edges and P is the number of threads, since the computation of both messages and beliefs scale with the number of edges. The communication required depends on the number of edges that cross partition boundaries, which depends on both the structure of the graph and the partitioning algorithm. In WLC, the number of edges that are cut by the weighted min-cut partitioning algorithm grows approximately as a function of $logP$ in both the Stanford and Berkeley datasets up to 64 partitions. Thus, total communication required scales as $ElogP$ and the communication to computation ratio scales as $logP$ approximately. In Stereo, the graph is always a four-connected grid and is partitioned into blocks by alternately slicing vertically and horizontally into twice the number of partitions. Under this partitioning scheme, the number of edge-cuts grows as a function of $\sqrt{E}(\sqrt{P}-1)$. Therefore, the communication to computation ratio grows as a function of $\frac{P(\sqrt{P}-1)}{\sqrt{E}}$. In both Stereo and WLC, the communication to computation ratio increase even if the dataset size scales with the number of processors. We will see whether this is an issue when we analyze the experimental data in the next section.

Robot Localization

In this algorithm, communication between threads is only required for reduction operations, weight computation and resampling. The amount of data communicated during reduction operations is negligible when the number of particles (N) is orders of magnitude larger than the number of threads (P), which is true for CMP's. Data communicated per thread during the weight computation scales as $\frac{P-1}{P}$, since each thread computes $\frac{1}{P}$ of the sensor data and subsequently accesses all of them. The per-thread communication required in the resampling process is dependent on the weight distribution and the random starting position, but scales as $\frac{N}{P}$ on average. Computation performed per thread scales as $\frac{N}{P}$ throughout the algorithm. Thus the communication to computation ratio scales approximately as $\frac{P-1}{N}$. Even if N does not scale with P, as long as N is significantly larger than P, the communication to computation ratio remains small.

4.3.4.2 Empirical Analysis

First we look at the total on-chip traffic. Figure 4.15 shows the total number of cache miss, write back and coherence messages sent throughout the execution of the programs as we scale the number of processors. The data for Speech decoding is plotted on a second Y-axis, as its volume of traffic is a lot larger than the others. Here we see that the total traffic grows significantly only in Speech Decoding and Recursive Conditioning. Note that the scale of this graph is at tens to hundreds of millions. Now contrast that with the number of messages that are for read-write sharing only, which is shown in Figure 4.16. Read-write sharing is the non-read-only cache-to-cache traffic that implies the communication of modified shared

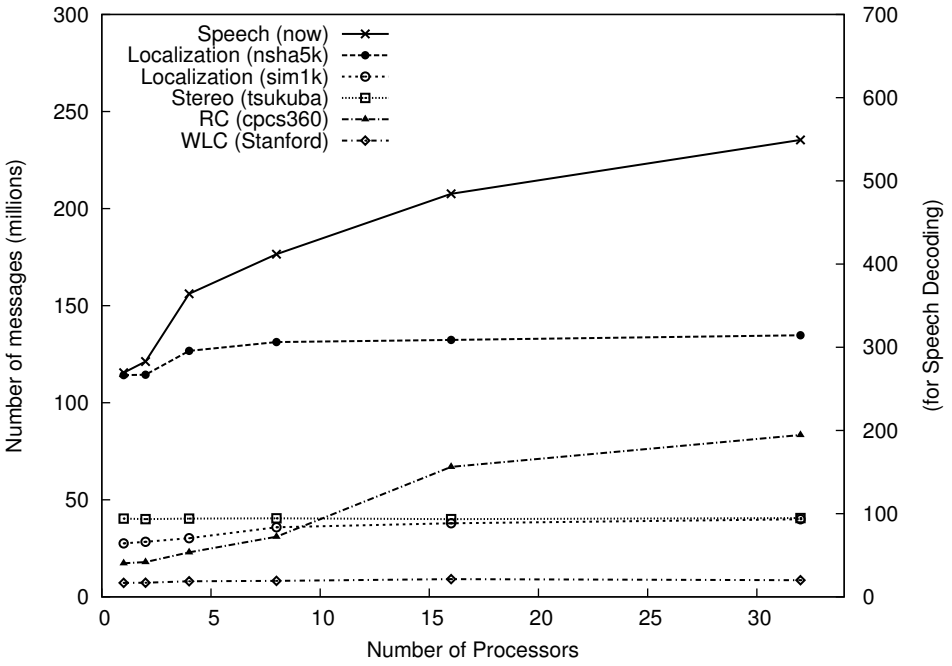

Figure 4.15: Growth in all misses, writebacks and coherence traffic

data.[2] Note that the scale of the Figure 4.16 is more than one order of magnitude smaller compared to the Figure 4.15. Hence communication is a small percentage of the total traffic in all these applications. Even though the asymptotic communication to computation ratio of WLC and Stereo grows and is not improved by scaling with the dataset size, the absolute amount of stall due to inherent communication is minuscule compared to the amount due to capacity and conflict misses. Thus communication is not the primary limiting factor of scalability and is not likely to be an issue in the future. Similarly, there is only a small increase in traffic in Localization as the number of threads increases, as thread to thread communication – which takes place mostly for resampling, – is dominated by memory traffic. In RC, traffic grows with the number of threads since each thread has to traverse the entire

[2]Since the granularity of cache-to-cache transfer is a cache line, false sharing – when processors access different words of a cache line – is a possibility, and the data includes both true and false sharing communication.

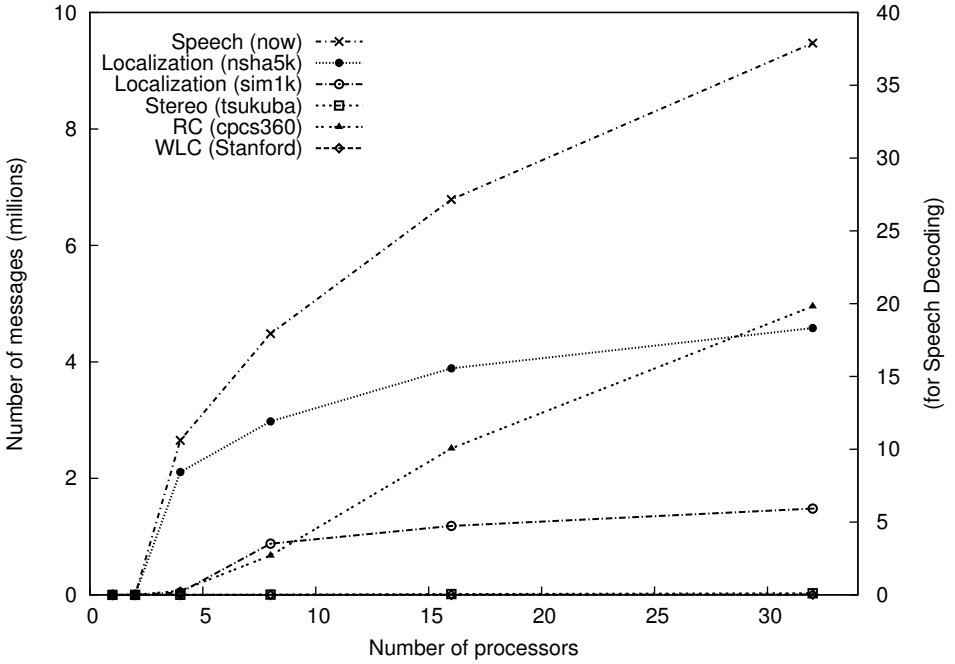

Figure 4.16: Growth in read-write sharing coherence traffic

d-tree. For any non-trivial network, the entire d-tree would not fit into the local data cache. However, each thread caches the part of the d-tree that it is traversing. With more threads, there is also a drastic increase in the portion of requests that are satisfied in another thread's cache or merged with an existing request. This increases the on-chip traffic but lowers the traffic to the off-chip memory. The growth in communication is from the sharing of results cached in the internal nodes of the d-tree. In Speech Decoding, the growth in communication here is due to the movement of HMM's as a result of the round-robin distribution scheme. In summary, most of the traffic generated in these applications is from memory misses of private data or read-shared data. Given that read-write sharing traffic is a small portion of the total traffic, it will not become a major bottleneck for performance even though communication grows as the number of processors increases.

4.4 Looking Ahead to Future Applications

In this chapter, we have examined three fundamental performance characteristics as regard to parallel performance – scalability, temporal locality and communication – of five probabilistic inference applications. These applications have plenty of data parallelism for general purpose chip multiprocessors that are coming to market in the near future. Except Speech Decoding, the data parallelism is easily extractable for thread parallel execution, while Speech Decoding requires some algorithmic changes in order to garner enough independent tasks for efficient execution on more than a few processors. We have seen how multithreaded execution can improve performance within the scalable range of configurations by effectively hiding memory and synchronization latency. These cases also show the importance of adequate memory and network bandwidth, as these applications tend to have large data sets and high miss rates. The availability of hardware supported fine-grain synchronization operations are also shown to be very useful in addressing application-specific performance issues, such as eliminating redundant computation in RC and alleviating imbalance in load partitioning in WLC.

While these applications have data parallelism like traditional floating-point benchmarks, their computation to memory ratios are clearly lower and thus they are not as compute intensive. Their memory access patterns are also less predictable, so they do not easily lend themselves to techniques like blocking, which improves data locality, and prefetching, which reduces effective memory latency. Hence supporting multiple hardware contexts per processor for latency tolerance is highly desirable. The high percentage of memory operations and high level 1 cache miss rate observed in most of these applications are also unusual in typical parallel benchmarks. This suggests that the focus of the architecture design should be the memory system when targeting these types of algorithms, as the ability of the memory

system to keep up with the demand for memory bandwidth and resources has the biggest impact on the application's performance.

The parallel performance issues discussed in Section 4.3.2 also highlight some important factors of algorithmic design that ease the parallelization process, which are not different from other parallel programs. While Speech Decoding has inherent data parallelism, the algorithm does not naturally decompose into independent tasks of similar workload. One culprit is the use of global data structures that are accessed and updated in a non-uniform way, e.g. the Viterbi history table in Speech Decoding. This is undesirable for parallel implementation because any potential access conflicts require synchronization, which not only affects performance but also makes the application harder to debug. Thus it is preferred to organize the global data or order the accesses such that the data is partitioned (at least virtually) along with the computation to be done on the data. Another important consideration is the effect of Amdahl's Law, which specifies the limit of parallel speedup in terms of the fraction of the program that is parallelizable. The 1% of serial code in the sequential implementation would turn into 25% in a 32-thread implementation, so minimizing the fraction of serial code is always part of the parallelization process. However, it is sometimes much simpler to develop an algorithm with this in mind from the beginning than to restructure the algorithm later on. Lastly, it is beneficial to make load balancing as simple as possible. For data-parallel algorithms, this means having an efficient way to divide the data into chunks that require the same amount of processing, and in some cases this requires some extra care in the construction phase of the data structure.

Larger Datasets / Graphs

The most obvious development is the increase in the size of the problems to be solved by these applications. This can be an increase in the number of entities that are modeled or an increase in the details of the systems being modeled. One prominent example is the availability of very high resolution images in medical, military, entertainment and other applications. This means the algorithms operate on much larger graphs. The upside is that there is more data parallelism, which in general means the problem can be scaled to a larger system. For message passing algorithms (like WLC), a potential issue is whether the increased complexity of the graph would affect the effectiveness of load partitioning while minimizing communication between partitions. Even though fine-grain synchronization can be used to recover some performance loss due to imperfect partitioning, there is a limit to how much imbalance can be tolerated. Fortunately, to avoid over-fitting of data or extraneous dependence between variables graphs with fewer edges are generally preferred. Therefore we expect well-balanced partitioning can be efficiently done for most graphs in the near future.

A Hierarchy of Inference Methods

Another direction of advancement in this field is the incorporation of multiple inference tools in the same algorithm. One example is simultaneous localization and mapping (SLAM), which is the problem of determining the location of a robot at the same time that the robot is creating a map of its environment using a number of landmarks. A way to solve this problem [59] is to combine a particle filter that estimates the robot's position with a set of Kalman filters that estimate the location of the landmarks based on each particle's estimate on the robot's position. Applications similar to this [60] that build on a hierarchy of inference methods would have great scalability in large scale CMP systems because of

the independence of each inference tool. Hence there is a hierarchy of parallelism that can be efficiently exploited by assigning group of cores to each tool with a coarse level of load balancing and exploiting data parallelism within each group.

Even though some applications, like Robot Localization and Speech Decoding, can already be done in real-time, the need to speed up probabilistic inference algorithms is still relevant for the foreseeable future as continued automation of a wide variety of tasks would demand more and more complex reasoning functions to be done in real-time. Taking advantage of the data parallelism in these algorithms and mapping them onto chip multiprocessors would be a good way to achieve the speed-up desired.

Chapter 5

Characterization of Soft Error Resilience

With shrinking device size and increasing complexity, soft errors are becoming an issue in the reliability of digital systems. To make efficient robust systems, it is important to understand how soft errors affect the quality of output for the target applications. Probabilistic inference applications are interesting since they produce non-exact results and yet are useful in many different fields. They employ approximate probabilistic algorithms that are designed to make them robust against noisy or incomplete input data. This robustness should make them more resilient to transient errors and thus require less protection. Moreover, the approximate nature of the computation also provides the opportunity of cheaper and faster fault recovery. In this chapter, we examine the inherent soft error resilience within approximate inference algorithms using instruction level fault injection experiments. We then propose software adjustments that take advantage of the approximate nature of these algorithms to reduce the chance that a transient error causes an observable error in the program output and

analyze their effectiveness. Lastly, we demonstrate how a low cost software level protection mechanism that is enabled by the probabilistic nature of the algorithm can significantly improve the robustness of the program.

5.1 Concern for Robust Computing

As circuit technology continues to scale and the complexity of integrated circuits increases, there has been a rising concern about the impact of transient errors (also known as soft errors) on these complex digital systems [61]. Shrinking devices and lower voltages mean circuit elements are more susceptible to upsets from noises (e.g. coupling and power supply) and particle strikes (e.g. neutrons, alpha particles). Although how the probability of these single-event upsets (SEU) would scale with technology is still being debated, most researchers agree that reliability is no longer just an issue for high availability and mission critical systems.

While most of the error detection and correction techniques that have been proposed ensure the hardware is correct and thus are independent of the software code being run, the instructions executed are not all of the same importance. For example, SPEC2000 integer benchmarks have been shown to mask 50% or more of single-event errors [62]. Hence the level of protection required varies during program execution as well as across different programs. Understanding which parts of the execution are more resilient to transient errors would allow us to potentially provide lower overhead (in performance or power) error detection and correction.

Since approximate probabilistic inference algorithms are designed to make them robust against noisy or incomplete input data, this robustness may make these types of algorithms

more resilient to transient errors and thus require less protection. In addition, the approximate nature of the computation can potentially provide the opportunity of cheaper and faster fault recovery. To determine if our intuition is true, we use fault injection to characterize the soft error resilience of these applications. Previous works have shown that transient faults in the microarchitectural states do not necessarily lead to erroneous program output. In [63] and [64], the authors identified types of instructions that are resilient to transient faults because errors in these instructions do not affect the outcome of the program. Studies such as [65] and [62] used fault injection in an RTL model to analyze effects of transient faults in modern microprocessors, but the focus of both is the vulnerability of various hardware structures. Rather than working at this low level, we randomly fault instruction results to test the resilience of the application to errors.

5.2 Related Work

In parallel to our effort, Li and Yeung [66] analyze the fault tolerance of applications based on soft computations, which include two multimedia decompression algorithms, a Support Vector Machine learning algorithm and a LBP algorithm. They used dynamic instruction slicing to identify soft instructions within those applications and found them to account for 62% of the instruction stream. Their fault injection experiments showed that with periodic checkpointing of the program counter, register file and stack, 96% of all injected faults could be tolerated. This is similar to our results presented in this chapter. Since the LBP algorithm used in their studies is the same as WLC in our application set, our analysis only uses the other three approximate inference programs (i.e. Stereo, Localization and Speech Decoding).

Software based fault injection has been frequently used in dependability analysis and

evaluation of fault tolerant mechanisms. Most studies focus on the detection and correction of transient faults (e.g. [67, 68]). They typically use general purpose benchmarks as the test software. Other studies focus on the operating system or runtime system [69, 70]. In contrast, our work focuses on the inherent error masking at the application level for a particular class of algorithms. In terms of fault injection methodology, our approach is similar to that used by Wang et. al. in [62] to analyze the level of transient fault masking in SPEC CINT2000.

Various software based transient fault protection have been proposed before. There are compile-time techniques such as a source-to-source compiler proposed by Rebaudengo et. al. in [71] and Error Detection by Duplicated Instructions (EDDI) proposed by Oh et. al. in [72]. Both rely on redundancy of computation and additional comparisons for detection. Due to the indiscriminate replication, the overhead of these techniques is high in both runtime and memory space. Another software fault tolerant technique is checkpointing [73, 74], which have been studied extensively in the context of parallel and distributed systems. These recovery schemes take a system wide approach to establish a globally consistent checkpoint. We apply the checkpointing concept for recovery within an individual application and take into account the application's natural ability to recover from most data error, which allows us to reduce the overhead considerably.

5.3 Inherent Soft Error Resilience

Based on our understanding of how the inference algorithms work, we can reason intuitively how they can tolerate or mask a transient error. For Stereo, since it is based on belief propagation that operates on a Markov model, a transient error in any belief or message vector will be "corrected" in the following iteration, assuming that iteration is error free.

Application	Program output	Output error metric	Threshold
Stereo	Disparity map (384 x 288 pixels)	Percentage of pixel error	0.01%
Localization	Mean pose <x, y, θ> (250 time steps)	# Time steps where mean differs by more than 1 std. dev. per error	1
Speech	Decoded speech (76 words, 26 seconds)	Word error rate	5%

Table 5.1: Metric for comparing program output

Similarly, a soft error that causes computation to be skipped in one iteration will have little impact as long as there are enough iterations for information to be propagated throughout the graph. Robot Localization is based on a particle filter that consists of hundreds to thousands of particles, so a soft error that affects the state of any particular particle is unlikely to have a big impact on the mean pose of the robot. Although an erroneous weight can potentially skew the mean, the weights are updated every time step using new odometry and laser readings, and the algorithm also discards laser readings that seem improbable. In Speech Decoding, there are numerous active hypotheses being considered at any point during the decoding process. Since only one of them will be the eventual result, a transient error will not affect the quality of decoding as long as it does not cause the best hypothesis to be pruned. With a high level understanding of where the error resilience comes from, we quantify the level of resilience in the following subsections.

5.3.1 Comparison of Soft Error Resilience against SPEC CINT2000

To confirm our belief that approximate probabilistic inference algorithms are more resilient to transient errors than traditional programs, we compare the level of error masking in these applications against a set of general purpose benchmarks. In [62], Wang et. al. found that

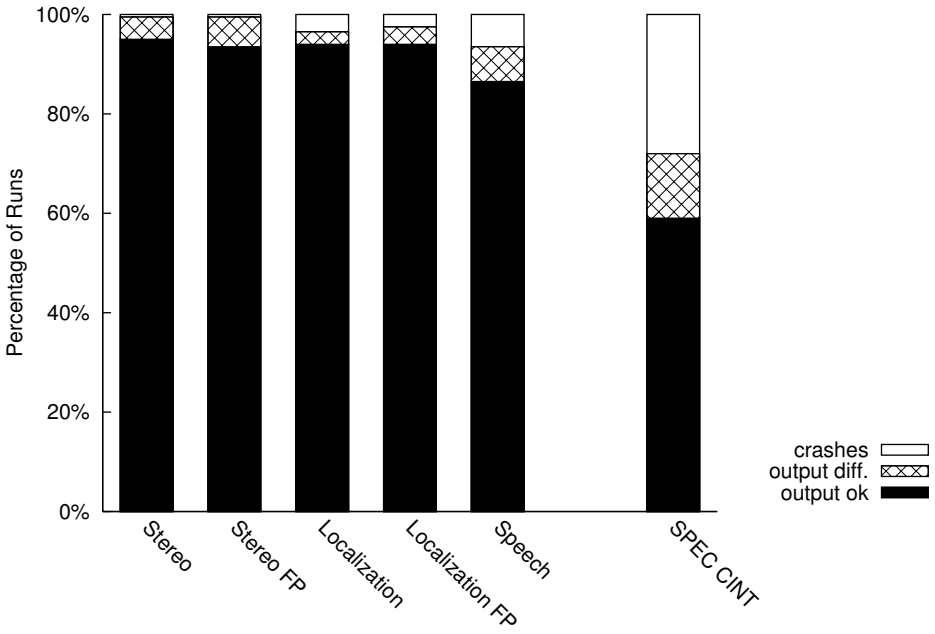

Figure 5.1: Breakdown of outcomes in single bit-flip fault injection experiments

on average 60% of single bit-flip error in an output register are masked at the program level across 10 SPEC CINT2000 programs. An error is masked when the program output has not been affected by the error. We carried out similar single bit-flip fault injection experiments using the SimpleScalar functional simulator [75]. In each experiment, an instruction is dynamically selected at random and a random single-bit error induced into the output register of that instruction. After the injection of an error, the program continues to execute until completion or termination. If it completes successfully, its output is compared to the reference output from an error free execution. If they are the same, we consider the injected fault masked. Table 5.1 lists the type of output for each application and how we quantify output deviation (the threshold column is explained in the next subsection). In Figure 5.1, we plot the breakdown of outcomes from 200 experiments for each application. Since two of these applications use mostly floating-point data and the floating-point format means

most bits in the mantissa do not affect the represented value significantly, we did a separate set of experiments to make sure this does not account for the difference in error resilience. In these experiments (labeled with a suffix of "FP"), only the sign bit, the exponent bits and the most significant bit of the mantissa are considered for a bit-flip when an error is injected in floating-point data. In all cases, the approximate inference applications mask significantly more of the injected error compared to SPEC CINT programs. For the floating-point applications, the ratio of masking is not really affected by how bit selection is done, which suggests the errors are masked at the algorithmic level.

5.3.2 Comparison of Soft Error Resilience across Applications

It is not clear from the single-error fault injection experiments how the behavior differs among the inference applications, so we performed a set of multiple-error experiments. In each experiment, one or more instructions are dynamically selected at random throughout the execution of the program. For each selected instruction, a single bit-flip error is induced into the output register. We compare the outcome of the experiment to the reference output as before, but since these algorithms produce approximate solution, we also set thresholds that define some levels of acceptable deviation to provide a sense of how many of the experiments produce an output that is different from the reference but can be considered "good enough".

Figure 5.2 shows the breakdown of the outcomes of error injection experiments for fault injection probability of 1e-9 to 5e-9 (each has 400 runs). At the same fault rate, Stereo is the least susceptible to crashes whereas Speech Decoding is the most sensitive. This is expected since Stereo is the most compute intensive among the three and Speech Decoding is the least, so in Speech Decoding it is more likely that an injected error would affect non-approximate data, particularly control-related data and pointers. To estimate the ratio of instructions

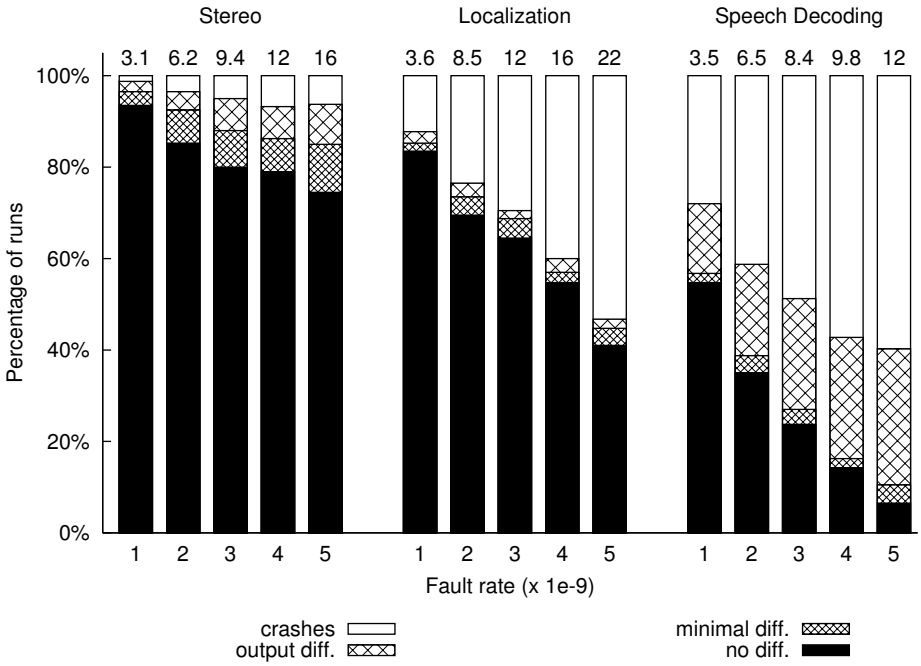

Figure 5.2: Error injection results at varying fault rate (numbers above bars indicate number of errors injected averaged over completed runs)

that are critical to the completion of the program (i.e. a soft error manifested at such an instruction will cause the program to crash), we fitted our data to a simple model that assumes each fault injected is independent and has a probability p of crashing the program. The independence assumption is reasonable given the low fault injection rates. While this is clearly not always true, Figure 5.3 shows it matches our data well. [1] The values of p obtained confirm that most of the dynamic instructions in these applications are resilient to soft errors.

[1] Since there are not enough data points at high number of faults, they are not used for the curve fitting.

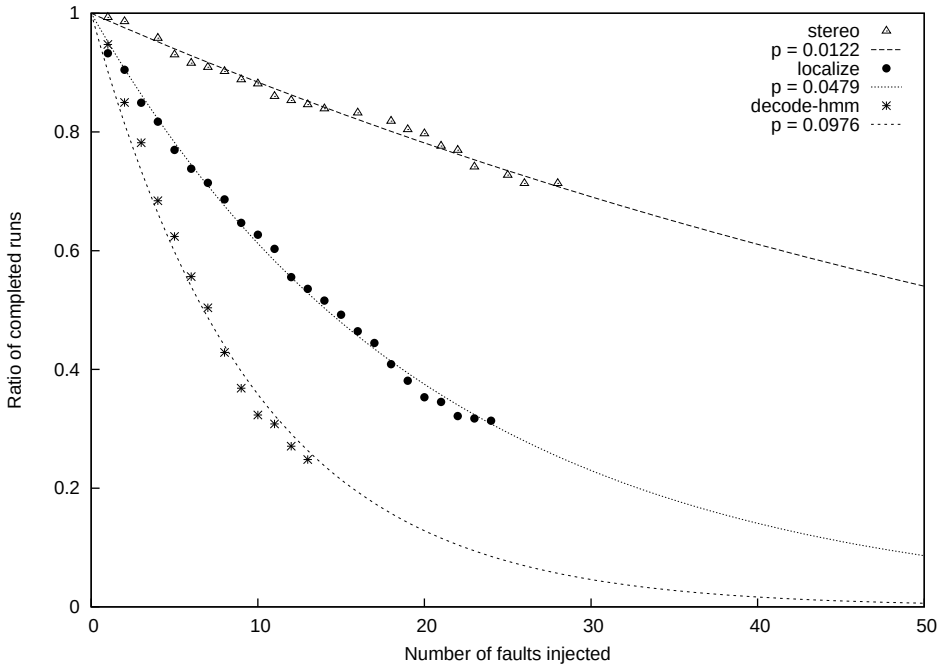

Figure 5.3: Ratio of completed run: model $(1-p)^x$ vs. data

5.4 Improving Error Resilience by Reducing Program Crashes

Although program crashes may be considered a better outcome than silent data corruption as they make the presence of error obvious, such abrupt termination denies any possibility of recovery that the algorithm may be capable. We can reduce the number of crashes by relaxing the normal correctness requirements. The idea is to detect various program termination conditions and make adjustments to the state of the program appropriately to remove such conditions. If the termination is triggered by a data error, it is just a matter of choosing a sensible value to replace the offending data since these algorithms naturally recover from inaccurate data in most cases. On the other hand, crashes caused by control flow errors require more explicit protection and recovery. We explore a pared down version of

checkpointing to reduce this type of crashes. Obviously, successful completion of execution does not mean the output is "correct" and we show how one application's ratio of masked transient errors can be boosted with a simple software level protection mechanism that targets a particular critical part of the algorithm.

5.4.1 Program Sanity Checks

Given some understanding on the various kinds of data used in a program, it is often possible to deduce properties that the data must adhere to. For example, values that represent probabilities are always non-negative, whereas values that are log-probabilities should be negative. Typically, this knowledge is applied in the form of assertions to aide debugging. These same checks can also detect some data error caused by transient faults in the hardware. In probabilistic applications, instead of terminating the program, we can use the knowledge about domain and range of data to substitute a reasonable value in place of the erred data. This allows the algorithm the chance to recover from the data error in subsequent iterations as new or updated evidence is incorporated.

Another way to increase the robustness is to identify data that are particularly critical to the success of the algorithm. Examples include the odometry data in Localization and the thresholds for pruning in Speech Decoding. For these data simple sanity checks in the software can offer great protection. For instance, odometry data can be checked against the physical limit of the robot's speed and if the data is out of range, either the program re-evaluates the sensor or the update should be skipped. As another example, pruning thresholds can be redundantly computed and pruning of an entire Viterbi path may be delayed until the subsequent audio frame to make sure the path score again falls below the threshold.

5.4.2 Exception Handling

Similar to program sanity checks, exceptions are often handled as program terminating errors. However, most exceptions can be viewed as data errors. Since approximate inference algorithms can recover from some data errors, we can customize exception handling to take advantage of the natural error resilience and allow the algorithm to complete its execution. Arithmetic exceptions like overflow and divide-by-0 often can be ignored. For memory access exceptions, a default value can be returned in case of a load and the operation can be skipped in case of a store. Invalid instruction exceptions require more processing in order for the program to complete execution normally. The next section discusses one way to recover from such an exception.

5.4.3 Checkpoint and Restart

We observe from our fault injection experiments that a majority of program crashes are the result of control flow errors. Recovery from control flow errors can be achieved by restarting the program execution at a well-defined point, such as the function entry point or the beginning of a loop iteration. For a general program, this normally requires a checkpoint of all program visible states, including the memory address space of the process, and thus the overhead of a checkpoint creation is significant. On the other hand, probabilistic inference algorithms tend to be structured such that computation can be repeated or even skipped with little effect on the end results. This means that in the event of a restart, it is only necessary to roll back a small amount of critical data like the stack pointer, the loop index and other loop-variant data. A simple checkpointing scheme is to backup the program counter and the integer registers, which only takes tens of cycles. While we may ignore the modifications to the program data in the memory space, we need to monitor the allocation

of memory pages so that those pages can be freed at a restart. This is necessary to prevent or reverse an out of virtual memory error, which is not uncommon if the control flow error leads to an infinite loop

5.4.4 Effect of Crash Reducing Techniques

We tested the techniques described in the previous sections and repeated the fault injection experiments. In our implementation, we converted most of the built-in program sanity checks to either substitute a reasonable value or attempt to restart from the last checkpoint. We modified exception handling to ignore arithmetic and memory access exceptions, and to initiate a restart in cases of invalid instruction exceptions. We inserted checkpointing instructions at natural locations of the programs. In Stereo, it is the beginning of each level of initial belief computation and the start of each belief propagation iteration. In Localization, it is the beginning of a time step whereas in Speech Decoding, it is the start of an audio frame. On average, a checkpoint is created every 50, 15 and 1.7 million instructions for Stereo, Localization, and Speech Decoding respectively.

Eliminating program crashes makes sense if the probability that a silent data corruption will change the program's outcome is small. This is the case for Localization, where we estimate only 1 of 200 errors will change the program's output. In contrast, for Speech Decoding roughly 1 in 11 errors will change the output and thus the technique works less well. This result is shown in Figure 5.4. For all applications, there is a drastic reduction in the proportion of crashes. The percentage of runs that produce correct output also improves, though that is simply because more runs complete. The ratio of "correct" runs to completed runs stays about the same. To improve the quality of output, we can use control flow error detection techniques so that restart would be initiated earlier, which would minimize the

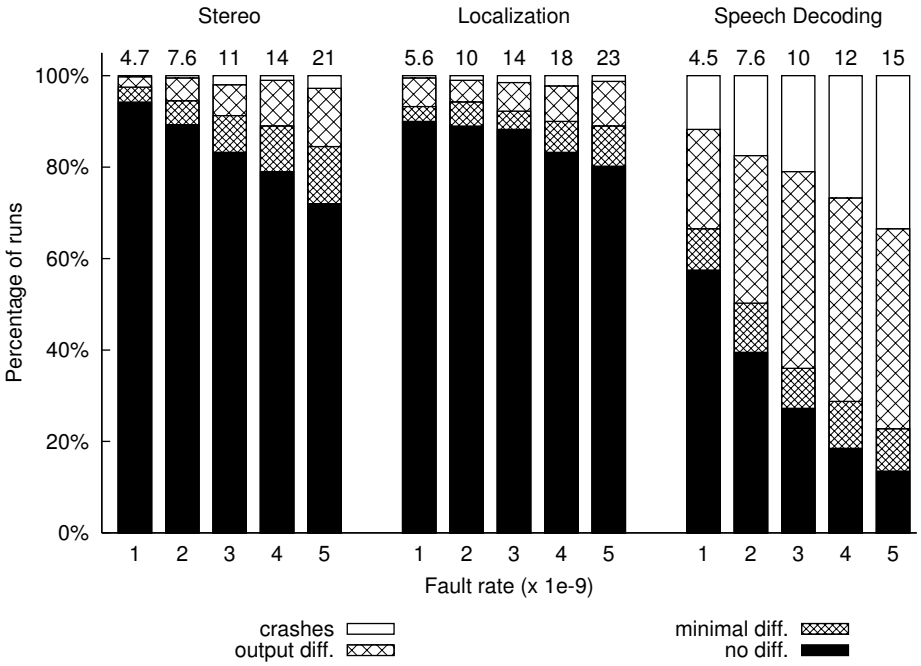

Figure 5.4: Error injection results after software adjustments (numbers above bars indicate number of errors injected averaged over completed runs)

amount of data corrupted. Improvement can also be made in the types of program sanity checks. For example, periodic data structure consistency checks can be done to detect critical data errors that may require additional repair before a restart.

Although crash reduction techniques greatly improved the ratio of completed runs for Speech Decoding, the quality of the output is by a large margin the worst among the three applications. While Speech Decoding has more control related instructions and data pointers that are less tolerant to soft errors, we wonder if a critical part of this application is responsible in a majority of the bad-output cases. When we take a closer look at those cases, we notice that most of them are omission errors, i.e. some words are missing but the remaining words are decoded correctly. Digging deeper, we found that omission errors are often a result of Viterbi path pruning that incorrectly removes most of the active paths.

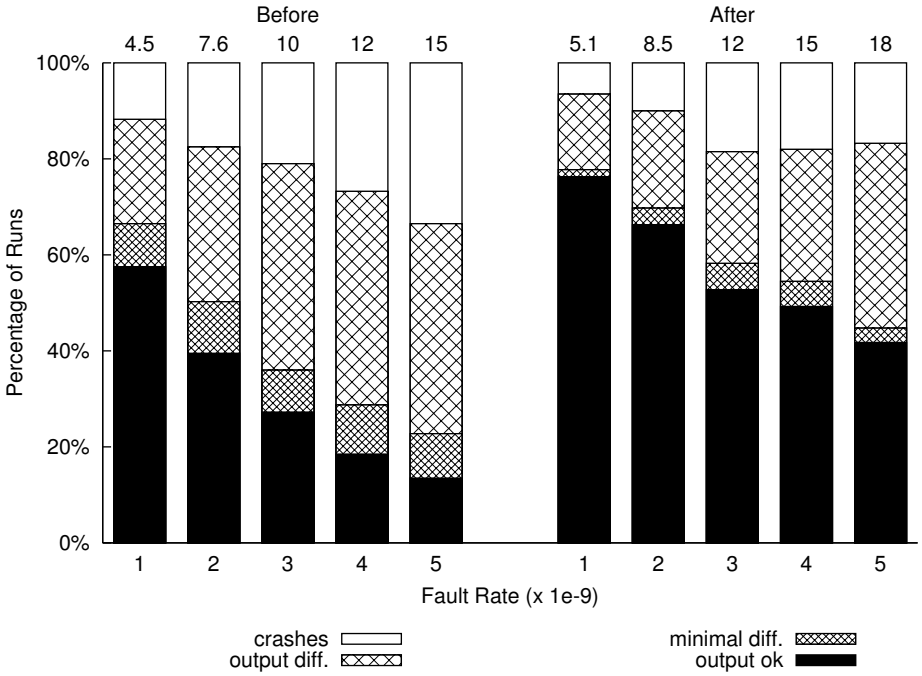

Figure 5.5: Effect of protection against pruning error in Speech Decoding (numbers above bars indicate number of errors injected averaged over completed runs)

This in turns is often caused by a bad score (resulted from an injected error) that is so high that most or all of the active paths are below the pruning threshold. To see if we can reduce the chance of these happening, we modified the algorithm to assume there is an error if the best score and the second best score differs by a large margin and use the second best score for pruning purposes instead. Figure 5.5 shows that this simple tweak improves the percentage of correct runs substantially, confirming that it is possible to greatly improve the algorithm's error resilience with little overhead by safeguarding critical data.

5.5 Summary and Future Work

As we had expected, approximate probabilistic applications are more robust compared to traditional programs. Our fault injection experiments show that these algorithms can naturally recover from most transient errors and produce results that are "correct" within the approximate framework. Even for errors that normally cause the program to crash, most can be masked by substituting erred data with a sensible value or by restarting execution from a well defined location in the program using light-weight checkpointing. We have also looked at how to take advantage of the approximate nature of these algorithms to implement low overhead software level protection that can effectively improve robustness. By identifying the part of the algorithm that is the most vulnerable, a simple software modification to detect error conditions and selectively discard unreliable data is shown to significantly increase the algorithm's resilience against transient errors.

From our experimentation with fault injection on approximate inference algorithms, we found that the application developer has the ability to greatly strengthen the fault tolerance of these algorithms without special hardware support. For transient faults that manifest as data errors, those that are not readily masked by the algorithm tend to occur in pointers or data used in pointer manipulation, and critical data used in irreversible operations. The former suggests that the use of arbitrary pointers should be minimized whenever possible and sanity checks should be used to limit the damage an erroneous pointer can cause. For data structures that require dynamic memory allocation and thus the use of pointers, allocation may be done within a certain address range such that simple base and bound tests can be applied to pointer values. To tolerate errors in critical data that affects irreversible operations, an effective way is to employ the kind of "disaster avoidance" protection similar to the pruning threshold test used in Speech Decoding (Section 5.4.4). For simplicity and

performance, the idea is not to replicate the computation in order to verify but to choose the simplest test that can weed out errors that may cause unrecoverable problems later in the algorithm. The most disruptive faults are those that affect the control flow of the program. We have shown that a minimalist checkpointing mechanism enable successful restarts for most control flow errors with minimal overhead. However, there are cases where restart is never triggered because only a small subset of functions have control flow checks, and there are cases where the algorithm is stuck in an infinite loop of restarts because the source error is not reset or recomputed upon restart. For a more complete protection scheme, active control flow error detection is likely necessary. Unfortunately, block level control flow checking carries high overhead in hardware resources, performance, code size or a combination of the above. A plausible strategy is to maintain a function identifier, which may be stored in a protected register, that is updated on a function call to indicate the function that is being executed. Instead of checking every branch instruction, verification of the identifier may be inserted before critical or irreversible operations, such as system calls, I/O, major modification of a global data structure, etc. This way the protection is targeted at where it is needed most. In a nutshell, the natural error tolerance in approximate inference algorithms enables more efficient error detection and mitigation but it must be tailored to the strength and weakness of the algorithms.

Chapter 6

Conclusion

Probabilistic graphical models have already been applied to a wide array of applications with great results. As scientists and engineers try to model increasingly sophisticated systems and automate more complex processes, where real-world uncertainty cannot be avoided, the probabilistic approach is likely to become more popular. Thus, it is an important class of algorithms for computer architects to consider in the design of future computing systems. Fortunately, these applications seem very well suited for the types of computers that will be available. Little change in either the algorithms or the hardware is likely to be needed because probabilistic algorithms have large amount of data parallelism, with relatively low communication rates, which are the essential features needed to leverage future chip multiprocessors and systems that contain many of these chips.

While these applications have large amounts of data parallelism, they also often have very large working sets that will not fit in future caches. This high effective memory access time means that in order to achieve good performance, some hardware mechanism to hide memory latency will be needed. Our results show that CMP's with multi-context processor

cores work well for these applications. While streaming machines would also be suitable for the algorithms with predictable and regular access patterns, the flexibility of multithreading makes it applicable for a wider range of applications. Multi-context support allows a thread to be run while another thread is waiting for data from memory. Clearly, for this processor to be effective, the memory system must support many outstanding memory references and provide high memory bandwidth. An additional feature that is advantageous for some applications is fine-grain synchronization, which allows implementation options that achieve better parallel efficiency than what would be possible otherwise. On the software side, the principles of constructing a good parallel program are the same in these applications as those in other areas. As the probabilistic approach becomes more popular, parallel implementations of some of the core algorithms will likely be incorporated in software libraries that would relieve application developers from worrying about the low level synchronization and load balancing.

In addition to being suitable for future chip multiprocessors, approximate inference algorithms seem to have an advantage towards achieving robustness in future machines. These algorithms are designed to be tolerant of errors in the model and data use for inference and this makes them inherently more resilient to transient errors as well. Hence, they do not require the same level of hardware protection as most programs. The important thing is to take advantage of the natural recoverability of the algorithm to simplify the error detection and correction schemes. A software centric approach that tailors to the particular vulnerabilities of the algorithm is an effective way to protect these applications with a small overhead and thus making it possible to achieve high level of reliability without hardware support.

In conclusion, probabilistic inference applications match well with the strengths and

challenges of future computing systems: Their abundant data parallelism fits well with CMP architecture and their inherent soft error resilience enables low-cost robust designs.

Bibliography

[1] R. Lindsay, B. Buchanan, E. Feigenbaum, and J. Lederberg, *Applications of Artificial Intelligence for Organic Chemistry – The DENDRAL Project.* McGraw-Hill, 1980.

[2] B. G. Buchanan, *Rule-Based expert systems: The MYCIN experiments of the Stanford heuristic programming project*, E. H. Shortliffe, Ed. Addison-Wesley, 1984.

[3] R. Miller, H. Pople, and J. Myers, "Internist-i, an experimental computer-based diagnostic consultant for general internal medicine," *New England Journal of Medicine*, vol. 307, pp. 468–476, 1982.

[4] S. K. Andersen, K. G. Olesen, F. V. Jensen, and F. Jensen, "Hugin - a shell for building bayesian belief universes for expert systems." in *IJCAI*, 1989, pp. 1080–1085.

[5] D. E. Heckermann, *Probabilistic similarity networks.* MIT Press, 1991.

[6] G. Zweig and S. Russell, "Speech recognition with dynamic bayesian networks," in *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence.* American Association for Artificial Intelligence, 1998, pp. 173–180.

[7] D. Heckerman, "Bayesian networks for data mining," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 79–119, 1997.

[8] J. Sun, N.-N. Zheng, and H.-Y. Shum, "Stereo matching using belief propagation," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 25, no. 7, pp. 787–800, 2003.

[9] S. Thrun, "Probabilistic algorithms in robotics," *AI Magazine*, vol. 21, no. 4, pp. 93–109, 2000. [Online]. Available: citeseer.ist.psu.edu/thrun00probabilistic.html

[10] N. Friedman, "Inferring cellular networks using probabilistic graphical models," *Science*, vol. 303, no. 5659, pp. 799–805, 2004.

[11] E. Segal, B. Taskar, A. Gasch, N. Friedman, and D. Koller, "Rich probabilistic models for gene expression." in *ISMB (Supplement of Bioinformatics)*, 2001, pp. 243–252.

[12] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[13] D. W. Wall, "Limits of instruction-level parallelism," Digital Western Research Laboratory, Palo Alto, CA, Tech. Rep. WRL Research Report 93/6, November 1993.

[14] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in *SPIE: Real Time Signal Processing IV*, vol. 298, January 1981, pp. 241–248.

[15] H. Geffner, *Default Reasoning: Causal and Conditional Theories*. MIT Press, 1992.

[16] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 10, pp. 338–353, 1965.

[17] I. J. Good, "A causal calculus," *British Journal of the Philosophy of Science*, vol. 11, pp. 305–318, 1961.

[18] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., 1988.

[19] C. J. Preston, *gibbs States on Countable Sets*. Cambridge University Press, 1974.

[20] F. Spitzer, *Random Fields and Interacting Particle Systems*. M.A.A. Summer Seminar Notes, 1971.

[21] T. Dean and K. Kanazawa, "A model for reasoning about persistence and causation," *Computational Intelligence*, vol. 5, no. 3, pp. 142–150, 1989.

[22] A. E. Nicholson and J. M. Brady, "The data association problem when monitoring robot vehicles using dynamic belief networks," in *ECAI 92: 10th European Conference on Artificial Intelligence Proceedings*, 1992, pp. 689–693.

[23] U. Kjaerulff, "A computational scheme for reasoning in dynamic probabilistic networks," in *Proceeedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, 1992, pp. 121–129.

[24] A. A. Markov, "An example of statistical investigation in the text of "eugene onegin" illustrating coupling of "tests" in chains," *Bulletin de l'Academie Imperiale Des Sciences de St. Petersburg*, vol. 3, pp. 153–162, 1913.

[25] J. Pearl, "Fusion, propagation, and structuring in belief networks," *Artificial Intelligence*, vol. 29, no. 3, pp. 241–288, 1986.

[26] S. L. Lauritzen and D. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society, Series B*, vol. 50, pp. 157–224, 1988.

[27] G. F. Cooper, "The computational complexity of probabilistic inference using bayesian belief networks (research note)," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 393–405, 1990.

[28] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence*, 1999, pp. 467–475.

[29] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 6, no. 6, pp. 721–741, Nov. 1984.

[30] M. Henrion, "Propagating uncertainty in bayesian networks by probabilistic logic sampling," in *Proceedings of the 2nd Annual Conference on Uncertainty in Artificial Intelligence*, 1986, pp. 149–163.

[31] F. Robert and C. Kuo-Chu, "Weighing and integrating evidence for stochastic simulation in bayesian networks," in *Proceedings of the 5th Annual Conference on Uncertainty in Artificial Intelligence*, 1989, pp. 209–219.

[32] A. Darwiche, "Recursive conditioning," *Artificial Intelligence*, vol. 126, no. 1-2, pp. 5–41, 2001.

[33] D. Allen and A. Darwiche, "Optimal time–space tradeoff in probabilistic inference," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003, pp. 969–975.

[34] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller, "Link prediction in relational data," in *Advances in Neural Information Processing Systems 16*, S. Thrun, L. Saul, and B. Schölkopf, Eds. Cambridge, MA: MIT Press, 2004.

[35] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient belief propagation for early vision." in *Proc. IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR 2004)*, vol. 1, 2004, pp. 261–268.

[36] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 11, pp. 1222–1239, November 2001.

[37] M. Montemerlo, N. Roy, and S. Thrun. (2002) The carnegie mellon robot navigation toolkit (carmen). [Online]. Available: http://www-2.cs.cmu.edu/ carmen/

[38] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *IEEE International Conference on Robotics and Automation (ICRA99)*, May 1999.

[39] The cmu sphinx group open source speech recognition engines. [Online]. Available: http://cmusphinx.sourceforge.net/html/cmusphinx.php

[40] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, April 1967.

[41] J. P. Singh, W.-D. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared-memory," *SIGARCH Computer Architecture News*, vol. 20, no. 1, pp. 5–44, 1992.

[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. ACM Press, 1995, pp. 24–36.

[43] V. Aslot and R. Eigenmann, "Quantitative performance analysis of the spec ompm2001 benchmarks," *Scientific Program.*, vol. 11, no. 2, pp. 105–124, 2003.

[44] A. V. Kozlov and J. P. Singh, "Parallel implementations of probabilistic inference," *Computer*, vol. 29, no. 12, pp. 33–40, 1996.

[45] L. Liu, C. Lai, and H. shan Jiang, "Parallel module network learning on distributed memory multiprocessors," in *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, 2005, pp. 129–134.

[46] V. K. Namasivayam and V. K. Prasanna, "Scalable parallel implementation of exact inference in bayesian networks," in *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS '06)*. IEEE Computer Society, 2006, pp. 143–150.

[47] A. Jaleel, M. Mattina, and B. Jacob, "Last level cache (llc) performance of data mining workloads on a cmp – a case study of parallel bioinformatics workloads," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006, pp. 88–98.

[48] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "Specomp: A new benchmark suite for measuring parallel computer performance," in *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*. London, UK: Springer-Verlag, 2001, pp. 1–10.

[49] (1997) Openmp: A proposed industry standard api for shared memory programming. [Online]. Available: http://www.openmp.org/

[50] N. R. Fredrickson, A. Afsahi, and Y. Qian, "Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2003, pp. 140–149.

[51] H. Saito, G. Gaertner, W. B. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, G. M. van Waveren, and B. Whitney, "Large system performance of spec omp2001 benchmarks," in *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*. London, UK: Springer-Verlag, 2002, pp. 370–379.

[52] Y. Chen, Q. Diao, C. Dulong, C. Lai, W. Hu, E. Li, W. Li, T. Wang, and Y. Zhang, "Performance scalability of data-mining worklaods in bioinformatics," *Intel Technology Journal*, vol. 09, no. 02, pp. 131–142, 2005.

[53] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, 1998, pp. 1–13.

[54] G. Karypis. (1998) Metis: Serial graph/mesh partitioning and sparse matrix ordering. [Online]. Available: http://www-users.cs.umn.edu/ karypis/metis/metis/index.html

[55] K. Schloegel, G. Karypis, and V. Kumar, "Parallel multilevel algorithms for multi-constraint graph partitioning," in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing.* London, UK: Springer-Verlag, 2000, pp. 296–310.

[56] Xtensa software development tools. [Online]. Available: http://www.tensilica.com/

[57] M. Pradhan, G. Provan, B. Middleton, and M. Henrion, "Knowledge engineering for large belief networks," in *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence (UAI-94).* San Francisco, CA: Morgan Kaufmann, 1994, pp. 484–49.

[58] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen, "A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single fpga," in *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays.* New York, NY, USA: ACM Press, 2007, pp. 60–68.

[59] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," in *Proceedings of the 18th AAAI National Conference on Artificial Intelligence.* Menlo Park, CA, USA: AAAI, 2002, pp. 593–598.

[60] B. Schumitsch, S. Thrun, G. Bradski, and K. Olukotun, "The information-form data association filter," in *Advances in Neural Information Processing Systems 18*, Y. Weiss, B. Schölkopf, and J. Platt, Eds. Cambridge, MA: MIT Press, 2006, pp. 1193–1200.

[61] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings 2002 International Conference on Dependable Systems and Networks (DSN '02)*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 389–398.

[62] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline." in *Proc. Int'l Conf. on Dependable Systems and Networks (DNS 2004)*, 2004, p. 61.

[63] N. Wang, M. Fertig, and S. Patel, "Y-branches: When you come to a fork in the road, take it," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*. IEEE Computer Society, 2003, p. 56.

[64] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, 2003, p. 29.

[65] S. Kim and A. K. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*. IEEE Computer Society, 2002, pp. 416–428.

[66] X. Li and D. Yeung, "Exploiting soft computing for increased fault tolerance," in *Proceedings of the 2006 Workshop on Architectural Support for Gigascale Integration*, 2006.

[67] G. Ries, Z. Kalbarczyk, T. Kraljevic, M.-C. Hsueh, and R. K. Iyer, "Depend: a simulation environment for system dependability modeling and evaluation," in *IPDS '96:*

Proc. 2nd Int'l Computer Performance and Dependability Symp. (IPDS '96). Washington, DC, USA: IEEE Computer Society, 1996, p. 54.

[68] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998.

[69] W. Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Trans. Softw. Eng.*, vol. 19, no. 11, pp. 1105–1118, 1993.

[70] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. Lie, D. Mannaru, A. Riska, and D. Milojicic, "JVM susceptibility to memory errors," in *Proc. Usenix JVM Research and Technology Symp.*, 2001, pp. 67–78.

[71] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "A source-to-source compiler for generating dependable software," in *Proc. IEEE Int'l Workshop on Source Code Analysis and Manipulation*, 2001, pp. 33–42.

[72] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[73] N. S. Bowen and D. K. Pradhan, "Processor and memory-based checkpoint and rollback recovery," *Computer*, vol. 26, no. 2, pp. 22–31, 1993.

[74] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[75] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.