Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs

M. Aater Suleman
ECE Department
The University of Texas at Austin
suleman@hps.utexas.edu

Moinuddin K. Qureshi *
IBM Research
T. J. Watson Research Center, New York
mkquresh@us.ibm.com

Yale N. Patt
ECE Department
The University of Texas at Austin
patt@ece.utexas.edu

Abstract

Extracting high-performance from the emerging Chip Multiprocessors (CMPs) requires that the application be divided into multiple threads. Each thread executes on a separate core thereby increasing concurrency and improving performance. As the number of cores on a CMP continues to increase, the performance of some multi-threaded applications will benefit from the increased number of threads, whereas, the performance of other multi-threaded applications will become limited by data-synchronization and off-chip bandwidth. For applications that get limited by data-synchronization, increasing the number of threads significantly degrades performance and increases on-chip power. Similarly, for applications that get limited by off-chip bandwidth, increasing the number of threads increases on-chip power without providing any performance improvement. Furthermore, whether an application gets limited by data-synchronization, or bandwidth, or neither depends not only on the application but also on the input set and the machine configuration. Therefore, controlling the number of threads based on the run-time behavior of the application can significantly improve performance and reduce power.

This paper proposes *Feedback-Driven Threading (FDT)*, a framework to dynamically control the number of threads using run-time information. FDT can be used to implement *Synchronization-Aware Threading (SAT)*, which predicts the optimal number of threads depending on the amount of data-synchronization. Our evaluation shows that SAT can reduce both execution time and power by up to 66% and 78% respectively. Similarly, FDT can be used to implement *Bandwidth-Aware Threading (BAT)*, which predicts the minimum number of threads required to saturate the off-chip bus. Our evaluation shows that BAT reduces on-chip power by up to 78%. When SAT and BAT are combined, the average execution time reduces by 17% and power reduces by 59%. The proposed techniques leverage existing performance counters and require minimal support from the threading library.

Categories and Subject Descriptors:
C.0 [General]: System architectures.
General Terms: Design, Performance.
Keywords: Multi-threaded, CMP, Synchronization, Bandwidth.

* The author was a PhD student at UT Austin during this work.

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

1. Introduction

It has become difficult to build large monolithic processors because of excessive design complexity and high power requirements. Consequently, industry [17] [19] [1] [31] has shifted to Chip-Multiprocessor (CMP) architectures that tile multiple simpler processor cores on a single chip. Industry trends [1] [31] show that the number of cores will increase every process generation. However, because of the power constraints, each core on a CMP is expected to become simpler and power-efficient, and will have lower performance. Therefore, performance of single-threaded applications may not increase with every process generation. To extract high performance from such architectures, the application must be divided into multiple entities called *threads*. Such applications are called multi-threaded applications. In multi-threaded applications, threads operate on different portions of the same problem which increases concurrency of execution. Multi-threaded applications can broadly be classified into two categories [10]. First, when multi-threading is done for the ease of programming and the number of threads is fixed (for example, producer-consumer threads). Second, when multi-threading is done solely for performance (for example, matrix multiply) and changing the number of threads does not impact correctness. Unfortunately, the performance of multi-threaded applications does not always increase with the number of threads because concurrently executing threads compete for shared data (data-synchronization) and shared resources (e.g. off-chip bus). This paper analyzes techniques for choosing the best number of threads for the applications in the second category.

The number of threads for a given application can be set statically using profile information. However, we show that the best number of threads for a given application can change significantly with input set and machine configuration. With CMPs becoming common, general purpose applications are being parallelized. Such applications are expected to perform well across different input sets and on varied machine configurations. Current systems set the number of threads to be equal to the number of available processors [33][2][29], unless informed otherwise. This approach implicitly assumes that increasing the number of threads always improves performance. However, when the performance of an application is limited by the contention for shared data or bus bandwidth, additional threads do not improve performance. In such cases, these threads only waste on-chip power. Furthermore, we show that the increase in contention for shared data due to additional threads can, in fact, increase execution time. Therefore, once the system gets limited by data-synchronization, further increasing the number of threads worsens both power and performance. Thus, for power-efficient and high-performance execution of multi-threaded applications, it is important to choose the right number of threads. However, whether an application gets limited by data-synchronization, or bandwidth, or neither is a function not only of the application but also of the input set and the machine configuration. Therefore, a mechanism that can control the number of threads at run-time, depending on the application behavior, can significantly improve performance and reduce power. This paper proposes *Feedback-Driven Threading (FDT)*, a framework that dynamically controls the number of threads using run-time information. FDT samples a small fraction of the parallelized kernels to estimate the application behavior. Based on this information, it estimates the number of threads at which the performance of the kernel saturates. FDT is a general framework which can handle several performance limiters. While it is desirable to have a scheme that can handle all performance bottlenecks, designing such a scheme may be intractable. Therefore, in this paper, we use the FDT framework to address the two major performance limiters: data-synchronization and bus bandwidth. Possible future work can extend FDT to handle other performance limiters such as contention for on-chip caches and on-chip interconnect.

Shared data in a multi-threaded application is kept synchronized using *critical sections*. The semantics of a critical section dictate that only one thread can be executing it at any given time. When each thread is required to execute the critical section, the total time spent in the critical section increases linearly with the number of threads. The increase in time spent in the critical section can offset the performance benefit obtained from the additional thread. In such cases, further increasing the number of threads worsens both power and performance. In Section 4, we propose an analytical model to analyze the performance of a data-synchronization limited application. We use this model along with the FDT framework to implement *Synchronization-Aware Threading (SAT)*. SAT can estimate the optimal number of threads at run-time depending on the time spent in critical sections. For multi-threaded workloads that are limited by data-synchronization, SAT reduces both execution time and power by up to 66% and 81% respectively.

For data-parallel applications where there is negligible data sharing, the major performance limiter tends to be the off-chip bandwidth. In such applications, demand for the off-chip bandwidth increases linearly with the number of on-chip cores. Unfortunately, off-chip bandwidth is not expected to increase at the same rate as the number of cores because it is limited by the number of I/O pins [12]. Therefore, performance of data-parallel applications is likely to be limited by the off-chip bandwidth. Once the off-chip bandwidth saturates, additional threads do not contribute to performance while still consuming power. In Section 5, we propose an analytical model to analyze the performance of bandwidth-limited applications. We use this model along with the FDT framework to implement *Bandwidth-Aware Threading (BAT)*. BAT can estimate the minimum number of threads required to saturate the off-chip bus. Our evaluation shows that BAT reduces on-chip power by up to 78% without increasing the execution time.

The two techniques, BAT and SAT, can be combined. Our evaluation with 12 multi-threaded applications shows that the combination reduces the average execution time by 17% and the average power by 59%. The proposed techniques leverage existing performance counters and require minimal support from the threading library.

2. Overview

Current systems set the number of threads equal to the number of cores [33][2][29]. While some applications benefit from a large number of threads, others do not. The two major factors that limit the performance of such applications are data-synchronization and off-chip bandwidth. This section describes these two limitations in detail. It also provides an overview of the proposed solution for high-performance and power-efficient execution under these limitations.

2.1 Limitations Due to Data-Synchronization

For multi-threaded applications, programmers ensure ordering of accesses to shared data using *critical sections*. A critical section is implemented such that only one thread can execute it at a given time. Therefore, all executions of a critical section get serialized. When all threads try to execute the critical section, the total time spent in executing the critical sections increases linearly with the number of threads. Furthermore, as the number of threads increase, the fraction of execution time spent in the parallelized portion of the code reduces. Thus, as the number of threads increase, the total time spent in the critical sections increases and the total time spent outside critical sections decreases. Consequently, critical sections begin to dominate the execution time and the overall execution time starts to increase.

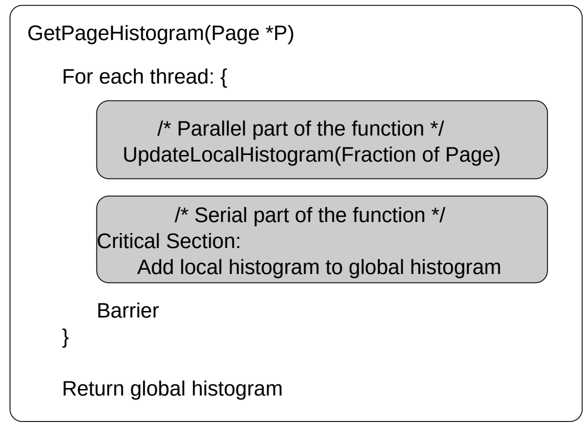

Figure 1. A function from `PageMine` that counts the occurrence of each ASCII character on a page of text

Figure 1 shows a function from `PageMine`[1] that counts the number of times each ASCII character occurs on a page of text. This function divides the work across T threads, each of which gathers the histograms for its portion of the page ($PageSize/T$) and adds it to the global histogram. Updates to the local histogram can execute in parallel without requiring data-synchronization. On the other hand, updates to the global histogram, which is a shared data-structure, are guarded by a critical section. Therefore, one and only one thread can update the global histogram at a given time. As the number of threads increase, the fraction of execution time spent in gathering local histograms decreases because each thread has to process a smaller fraction of the page. Whereas, the number of updates to the global histogram increases, which increases the total time spent in updating the global histogram.

Figure 2 shows the normalized execution time of `PageMine` as the number of threads are increased from 1 to 32 (Section 3 describes our experimental methodology). The execution time decreases until 4 threads and increases substantially beyond 6 threads because the time spent in the critical section begins to dominate the overall execution time. Therefore, having more than six threads worsens both performance and power. A mechanism that can control the number of threads based on the fraction of time spent in critical sections can improve both performance and power-efficiency.

[1] The code for `PageMine` is derived from the data mining benchmark `rsearchk` [26]. This kernel generates a histogram, which is used as a signature to find a page similar to a query page. This kernel is called iteratively until the distance between the signatures of the query page and a page in the document is less than the threshold. In our experiments, we assume a page-size of 5280 characters (66 lines of 80 characters each) and the histograms consists of 128 integers, one for each ASCII character.

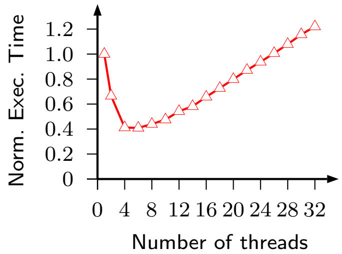

Figure 2. Execution time normalized with respect to one thread for `PageMine`

2.2 Limitations Due to Off-Chip Bus Bandwidth

Data-synchronization does not affect all applications. For example, data-parallel applications where threads operate on separate data require negligible data-synchronization. For such applications, contention for shared resources that do not scale with the number of threads is more likely to limit performance. One such resource is the off-chip bandwidth. For applications with negligible data sharing, the demand for off-chip bandwidth increases linearly with the number of threads. However, the off-chip bandwidth is not expected to increase as the number of cores because it is limited by the number of I/O pins [12]. Therefore, these applications become off-chip bandwidth limited. Once the off-chip bus saturates, no further performance improvement can be achieved by increasing the number of threads. Thus, the performance of an off-chip bus limited system is governed solely by the bus bandwidth and not by the number of threads. However, having more threads than required to saturate the bus only consume on-chip power without contributing to performance.

```
EuclideanDistance(Point A)
        /*Parallel threads compute partial sums*/
        for i = 1 to N
          sum = sum + A[i] * A[i]

    Return sqrt(sum)
```

Figure 3. A function which computes Euclidean Distance (`ED`) of a point from origin in an N-dimensional space

Figure 3 shows a function which computes the Euclidean Distance (`ED`) of a point in an N dimensional space. The function contains a data-parallel loop which can easily be distributed across multiple threads with negligible data-synchronization. As the number of threads increase, more iterations are executed in parallel which reduces the execution time until the off-chip bandwidth utilization is 100%. Once the system is bandwidth limited, no further performance improvement is obtained.

Figure 4a shows the normalized execution time of `ED` (N=100M) as the number of threads increase from 1 to 32. The execution time reduces until 8 threads and then becomes constant. With multiple threads, the demand for off-chip bus increases linearly. Figure 4b shows the bandwidth utilization of this loop as the number of threads increase from 1 to 32. Until 8 threads, the bandwidth utilization increases linearly and then stays at 100% for more threads. Therefore, having more than 8 threads does not reduce execution time. However, more threads increase on-chip power consumption which increases linearly with the number of threads. A threading scheme that is sensitive to bandwidth consumption will avoid such extraneous threads that consume power without improving performance.

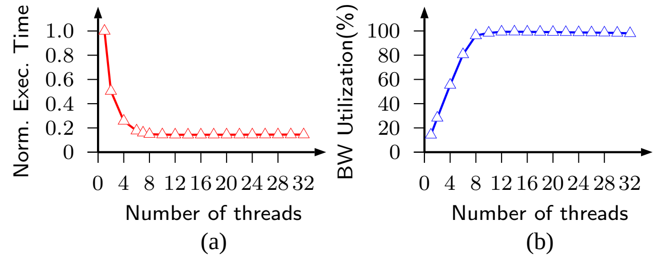

Figure 4. (a) Normalized execution time with respect to one thread for `ED`. (b) Bandwidth utilization of `ED`

2.3 Solution: Feedback-Driven Threading

For power-efficient and high-performance execution of multi-threaded workloads, we propose *Feedback-Driven Threading (FDT)*. Figure 5 shows an overview of the FDT framework. Unlike conventional threading, which statically divides the work into a fixed number of threads, FDT dynamically estimates the number of threads at which the performance saturates. FDT samples some portion of the application to estimate the application behavior. For example, if the system is likely to be bandwidth limited, FDT measures bandwidth utilization during the training phase. After the training, FDT decides the number of threads based on this information. While FDT is a general framework, this paper uses it to address the two major performance limiters: data-synchronization and bus bandwidth. For high performance and power-efficient execution of workloads that are limited by data-synchronization, the FDT framework can be used to implement *Synchronization-Aware Threading* (Section 4). Similarly, for power-efficient execution of bandwidth-limited workloads, the FDT framework can be used to implement *Bandwidth-Aware Threading* (Section 5). We present our experimental methodology before we describe these two techniques.

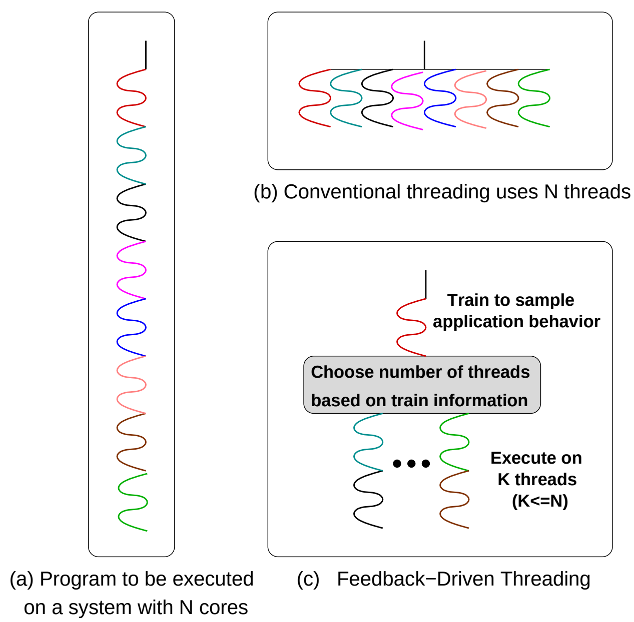

Figure 5. Overview of Feedback-Driven Threading

3. Experimental Methodology

3.1 Configuration

For our experiments we use a cycle-accurate x86 simulator. Configuration of the simulated machine is shown in Table 1. We simulate a CMP system with 32 cores. Each core is two-wide issue, in-order, with private L1 and L2 caches. The L3 cache is 8MB and is shared among all cores. Memory is 200 cycles away and contains 32 banks. The off-chip bus is capable of servicing one cache line every 32 cycles at its peak bandwidth. For power measurements, we count the number of cores that are active in a given cycle and the power is computed as the average of this value over the entire execution time.

System	32-core CMP with shared L3 cache
Core	In-order, 2-wide, 5-stage pipeline, 4-KB Gshare 8KB write-through private I and D cache 64KB, 4-way associative, inclusive private L2 cache
On-chip Interconnect	Bi-directional ring. Separate control and data ring 64-byte wide, 1-cycle hop latency
Coherence	Distributed directory-based MESI
Shared L3 cache	8MB, 8-way associative with 8 banks, 20-cycle, 64-byte cache lines, LRU replacement
Memory Data Bus	4:1 cpu/bus ratio, 64-bit wide, split-transaction pipelined bus, 40 -cycle latency
Memory	32 DRAM banks, approx. 200 cycle bank access, bank conflicts, open/close pages, row buffers modeled

Table 1. Configuration of the simulated machine

3.2 Workloads

We simulate 12 multi-threaded applications from different domains. The applications are divided into three categories. The performance of `PageMine`, `ISort`[3], `GSearch`[9], and `EP`(psuedo-random number generator)[3] is limited by data-synchronization. Whereas, the performance of `ED`, `convert`(a unix utility), `Transpose`[30], and `MTwister`(psuedo-random number generator)[30] is limited by bus bandwidth. Performance of `BT`[3], `MG`[3], `BScholes`[30], and `SConv`[30] is limited neither by data-synchronization nor by off-chip bandwidth. Such applications continue to benefit from more threads. We believe that such applications will drive the number of cores on future CMPs and hence we simulate a 32-core CMP. All applications were parallelized using `OpenMP` [8] directives and compiled using the Intel C Compiler [14]. We execute all applications to completion. Table 2 shows the description and input-set for each application.

Type	Workload	Problem description	Input set
CS limited	PageMine	Data mining kernel	1000 pages
	ISort	Integer sort	n = 64K
	GSearch	Search in directed graphs	10K nodes
	EP	Linear Congruential PRNG	262K numbers
BW limited	ED	Euclidean distance	n = 100M
	Convert	Image processing	320x240 pixels
	Transpose	2D Matrix transpose	512x8192
	MTwister	Mersenne-Twister PRNG	CUDA [30]
Scalable	BT	Fluid Dynamics	12x12x12
	MG	Multi-grid solver	64x64x64
	BScholes	Black-Scholes Pricing	CUDA [30]
	SConv	2D Separable convolution	512x512

Table 2. Details of simulated workloads

4. Synchronization-Aware Threading

Critical sections serialize accesses to shared data in multi-threaded applications. As the number of threads increase, the total time spent in the critical section increases. Therefore, the marginal reduction in the execution time caused by each additional thread must offset the marginal increase in execution time due to the critical section. We explain this phenomenon with an example. Figure 6 shows the execution of a program which spends 20% of its execution time in the critical section and the remaining 80% in the parallel part. The overall execution time with one thread is 10 units.

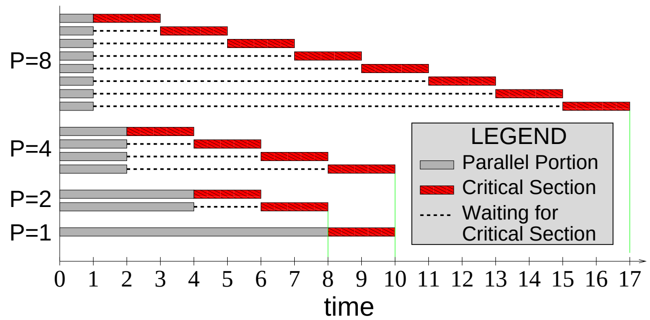

Figure 6. Example for analyzing impact of critical sections

When the program is executed with two threads, the time taken to execute the parallel part is reduced to four units while the total time to execute the critical section increases from two to four units. Therefore, the total execution time reduces from 10 units to 8 units. However, overall execution time reduces with additional threads only when the benefit from reduction in the parallel part is more than the increase in the critical section. For example, increasing the number of threads to four reduces the time for the parallel part from four to two units but increases the time for the critical section from four to eight units. Therefore, increasing the number of threads from two to four increases the overall execution time from 8 units to 10 units. Similarly, increasing the number of threads to eight further increases the overall execution time to 17 units.

4.1 Analytical Model

We can analyze the impact of critical sections on overall execution time using an analytical model. Let T_{CS} be the time spent in the critical section and T_{NoCS} be the time to execute the *parallel part* of the program. Let, (T_P) be the time to execute the critical sections and the parallel part of the program when there are P threads. Then, T_P can be computed as:

$$T_P = \frac{T_{NoCS}}{P} + P \cdot T_{CS} \tag{1}$$

The number of threads (P_{CS}) required to minimize the execution time can be obtained by differentiating Equation 1 with respect to P and equating it to zero.

$$\frac{d}{dP} T_P = -\frac{T_{NoCS}}{P^2} + T_{CS} \tag{2}$$

$$P_{CS} = \sqrt{\frac{T_{NoCS}}{T_{CS}}} \tag{3}$$

Equation 3 shows that (P_{CS}) increases only as the *square-root* of the ratio of time outside the critical section to the time inside the critical section. Therefore, even if the critical section is small, the system can become critical section limited with just a few threads. For example, if the critical section accounts for only 1% of the

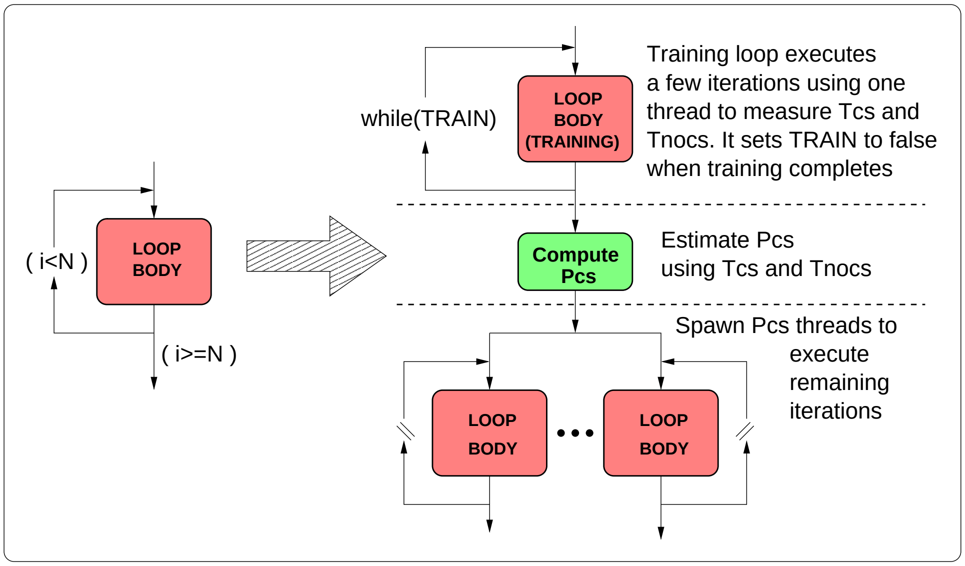

Figure 7. Synchronization-Aware Threading

overall execution time, the system becomes critical section limited with just 10 threads. Moreover, having more than P_{CS} threads, increases both execution time as well as power consumption. Therefore, a mechanism that predicts P_{CS} for such applications can improve both performance and power-efficiency. To that end, we propose Synchronization-Aware Threading (SAT). The next Section describes the implementation of SAT.

4.2 Implementation of SAT

The value of P_{CS} can be computed if T_{CS} and T_{NoCS} are known. Using the FDT framework, SAT samples a small fraction of the application kernel to estimate T_{CS} and T_{NoCS}. We perform SAT only on loop kernels that have been parallelized by the programmer.[2] Figure 7 shows the implementation of the SAT mechanism for a typical kernel. It consists of three parts: training, estimation, and execution.

4.2.1 Training:

The training loop is generated using a method similar to *loop peeling* [18]. The compiler divides the loop into two parts. The first part, which executes only a few iterations, is used for training. To measure T_{CS}, the compiler inserts instructions to read the cycle counter[3] at the entry and exit of the critical section. T_{CS} is computed at runtime by calculating the difference between the two cycle counts. T_{NoCS} can be estimated if the total time to execute each iteration is known. The execution time required for each iteration is also measured by reading the cycle counter at the beginning and end of the loop and taking the difference. T_{NoCS} is computed by subtracting T_{CS} from the total time for one iteration. In our experiments, we perform loop-peeling and instrumentation of the training loop using a source transformation tool.

To simplify the mechanism, the training loop is always executed in single threaded mode. Training loop is terminated if the ratio of T_{CS} to T_{NoCS} is stable (within 5%) for three consecutive iterations. Otherwise, the training continues for a maximum of 1% of the total iterations.

4.2.2 Estimation:

The estimation stage computes P_{CS} using Equation 3 and the values of T_{CS} and T_{NoCS} measured during training. The value of P_{CS} is rounded to the nearest integer. The number of threads is chosen as the minimum of P_{CS} and the number of cores available on the chip.

4.2.3 Execution:

The remaining iterations of the loop are executed using the estimated number of threads. This is performed in our experiments using the `OpenMP` clause *num_threads*, which allows the number of threads to be changed at runtime.

4.3 Results

Figure 8 shows the execution time with SAT for the four applications that are limited by data-synchronization: `PageMine`, `ISort`, `GSearch`, and `EP`. We also show the normalized execution time for the baseline system as the number of threads is varied from 1 to 32. For all cases, the execution time with SAT is within 1% of the minimum execution time.

For `PageMine`, execution time decreases until 4 threads and begins to increase beyond 6 threads. The critical section consumes approximately 2.34% of the total execution time in each iteration of the loop. Therefore, SAT estimates the best number of threads to be 6.53 (rounded to 7). For `ISort`, the execution time is minimized at 7 threads, which is successfully predicted by SAT. The main kernel in `GSearch` has two separate critical sections. After a particular node and its children have been searched, threads remove these nodes from the queue of nodes that are still to be searched. In addition, all nodes visited by the threads are marked to avoid redundant searches. Therefore, the fraction of time within the critical section varies across iterations. On average, 3.84% of time is spent in the critical section. SAT trains for 1% of the iterations and correctly chooses 5 threads. For `EP`, having 4 threads minimizes the execution time while SAT predicts 5 threads.

---

[2] For the applications in our studies, we identify these parallelized kernels with the help of `OpenMP` directive `parallel`. However, SAT is not restricted to applications parallelized with `OpenMP` directives and can easily be extended to other threading primitives

[3] Instructions to read the on-chip cycle counter exist in most modern ISAs.

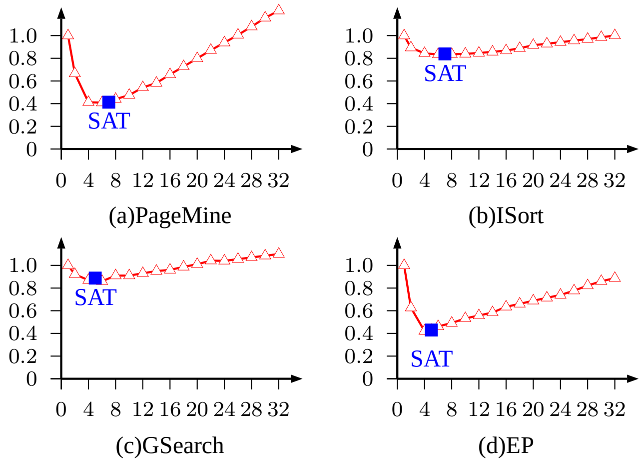

Figure 8. Performance of SAT. Vertical axis shows the normalized execution time and horizontal axis shows the number of threads.

4.4 Adaptation of SAT to Application Behavior

The time spent inside and outside the critical section typically depends on the input set. Therefore, the number of threads that minimize execution time also varies with the input set. Figure 9 shows the number of threads that minimize execution time for `PageMine` as the page-size is varied from 1KB to 25KB (default page-size is 5.2KB). The best number of threads varies widely with the page-size. Therefore, a solution that chooses the best number of threads statically for one page-size will not be optimal for other page sizes. As SAT is a run-time technique, it can adapt to changes in the application behavior.

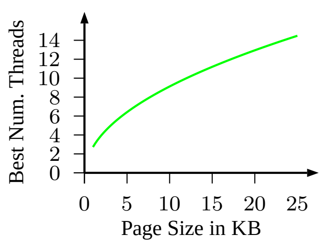

Figure 9. Best number of threads vs page-size for `PageMine`

Figure 10 shows the normalized execution time for `PageMine` for page sizes of 2.5KB and 10KB as the number of threads is varied from 1 to 32. SAT correctly chooses the right number of threads for both the page sizes.

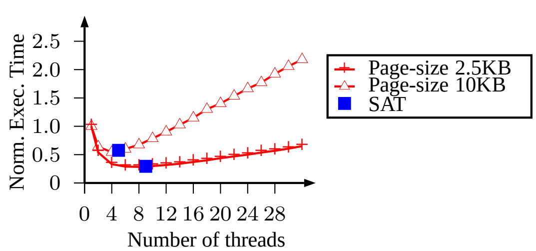

Figure 10. Performance of SAT for 2.5KB and 10KB page-size

5. Bandwidth-Aware Threading

Data-synchronization does not affect all applications. For example, data-parallel applications where threads operate on separate data require negligible synchronization. For such applications, the working set is typically huge and the demand for bandwidth increases linearly with the number of threads.

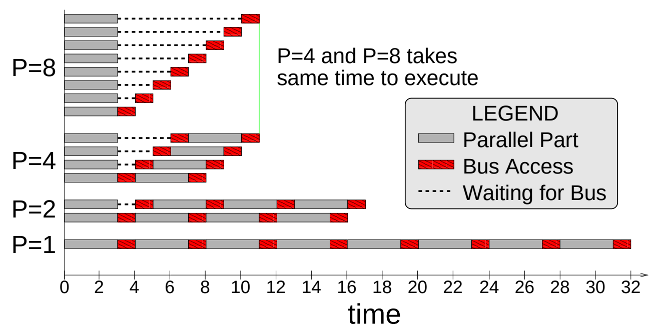

Figure 11. Example for analyzing bandwidth limited systems

Figure 11 demonstrates the bandwidth usage of a typical data parallel application. When a single thread executes, only 25% of execution time is spent transmitting data on the off-chip bus. Therefore, utilization of the off-chip bus is 25%. If the same loop is split across two threads, the execution time reduces and the bus utilization increases to 50%. Similarly, increasing the number of threads to four further reduces execution time while saturating the bus. As the bus is 100% utilized, the system becomes off-chip bus limited and further increasing the number of threads from four to eight does not reduce the execution time.

5.1 Analytical Model

We analyze the impact of off-chip bus bandwidth on overall execution time using an analytical model. Let BU_1 be the percent bus utilization with a single thread. When the working set of the application is large, the bus utilization increases linearly with the number of threads. In such cases, the bus utilization (BU_P) with P threads can be computed as:

$$BU_P = P \cdot BU_1 \tag{4}$$

When BU_P becomes 100%, the system becomes off-chip bandwidth limited. Therefore, the number of threads (P_{BW}) required to saturate the bus can be computed as:

$$P_{BW} = \frac{100}{BU_1} \tag{5}$$

Thus, if a single thread utilizes the off-chip bus for 10% of the time, then the system will become off-chip bandwidth limited for more than 10 threads. Once the number of threads is sufficient to saturate the bus, the performance of the system becomes a function of bus speed rather than the number of threads. If T_1 is the time to execute the *parallel part* of the program with one thread, then the execution time (T_P) with P threads is:

$$T_P = \begin{cases} \frac{T_1}{P} & P \leq P_{BW}, \\ \frac{T_1}{P_{BW}} & P > P_{BW}. \end{cases} \tag{6}$$

Increasing the number of threads beyond P_{BW} does not reduce execution time, however, it does increase the on-chip power. Therefore, a mechanism that can estimate P_{BW} for such applications can reduce the on-chip power. To this end, we propose *Bandwidth-Aware Threading (BAT)*. The next section describes the implementation of BAT.

5.2 Implementation of BAT

The value of P_{BW} can be computed if BU_1 is known. BAT uses the FDT framework to estimate the value of BU_1 and is implemented similar to SAT, except for three differences:

1. Training: The training loop has code to measure the off-chip bus utilization. The number of cycles the off-chip bus is utilized is measured by reading a performance monitoring counter[4] at the start and end of the loop. The difference in the two readings denotes the number of cycles the off-chip bus is busy. The total time to execute each iteration is measured similarly by reading the cycle counter and taking the difference. Bus utilization is computed as the ratio of the bus busy cycles to the total cycles.
2. Termination: Training terminates after at most 1% of the loop iterations are executed. Additionally, after 10000 cycles, if the product of the average bus utilization times the number of cores available on chip is less than 100%, BAT predicts that the system can not become bandwidth limited and training terminates.
3. Estimation: The estimation stage computes P_{BW} using Equation 5 and the value of BU_1. P_{BW} is rounded up to the next integer because a higher number of threads may not hurt performance while a smaller number can. The number of threads is chosen as the minimum of P_{BW} and the number of cores available on the chip.

5.3 Results

Figure 12 shows the execution time with BAT for the four applications that are limited by off-chip bandwidth: `ED`, `convert`, `Transpose`, and `MTwister`. We also show the normalized execution time for the baseline system as the number of threads is varied from 1 to 32. For all cases, the execution time with BAT is within 3% of the minimum execution time.

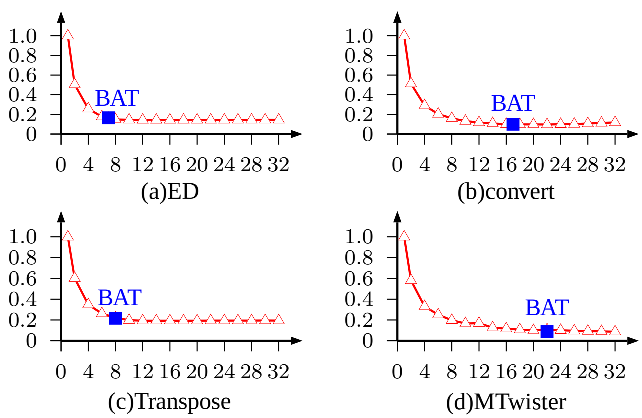

Figure 12. Performance of BAT. Vertical axis denotes normalized execution time and horizontal axis the number of threads.

The kernel in `ED` incurs a miss every 225 cycles on average. The bus utilization with a single thread is approximately 14.3%. The execution time is minimum with 8 threads, however, BAT predicts 7 threads. This occurs because BAT assumes that bandwidth utilization increases linearly with the number of threads. However, the contention for other shared resources like on-chip cache, on-chip interconnect, dram-bank causes the bandwidth utilization to scale slightly sub-linearly. Nevertheless, the execution time of BAT with 7 threads is similar to that with 8.

The kernel in `convert` computes one row of the output image at a time and writes it to a buffer. Both reading and writing the image consumes off-chip bandwidth. As bus utilization with a single thread is approximately 5.8%, BAT predicts 17 threads. The execution time with BAT is similar to the minimum execution time, which occurs with 18 threads.

The data-parallel kernel in `Transpose` computes the transpose of a matrix. Each thread operates on a different column of the matrix and bus utilization is high (12.2% with a single thread). BAT predicts 8 threads which is similar to the minimum number of threads that cause the bus utilization to reach 100%.

`MTwister` includes two data-parallel kernels. The first kernel implements a Mersenne-Twister random number generator [24]. The second kernel applies the Box-Muller transformation [4] on the random numbers generated by the first kernel. The data set does not fit in the L3 cache. The performance of the first kernel continues to scale until 32 threads. However, performance of the second kernel saturates at 12 threads due to bandwidth limitation. Thus, the two kernels require different number of threads and statically choosing a fixed number of threads for the whole program cannot lead to minimum power. BAT correctly predicts the number of threads to be 32 for the first kernel and 12 for the second kernel, reducing the average number of threads to 21. Thus, BAT saves power without impacting the execution time.

The results show that BAT correctly estimates the minimum number of threads required to saturate the off-chip bandwidth. BAT can significantly reduce on-chip power for such applications because on-chip power consumed in cores is directly proportional to the number of active cores. Compared to the case where as many threads are used as the number of on-chip cores, BAT reduces the power consumed in the cores by 78% for `ED`, 47% for `convert`, 75% for `Transpose`, and 31% for `MTwister`. Additionally, as BAT does not impact execution time significantly, the savings in power can be directly interpreted as savings in energy.

5.4 Adaptation of BAT to Machine Configuration

The minimum number of threads required to saturate the bus is dependent on the system bus bandwidth. Figure 13 shows the normalized execution time for `convert` as the number of threads is varied from 1 to 32 for two systems: first with one-half the bandwidth of the baseline machine and the second with double the bandwidth. For the first system, the execution time saturates at 8 threads, whereas, for the second system it continues to decrease. Therefore, a solution that statically chooses the number of threads required to saturate the bus bandwidth of one system can, in fact, hurt performance for another system. For example, using 8 threads for the second system doubles its execution time. BAT computes the number of threads required to saturate the bus at runtime, therefore, it is robust to changes in the machine configuration. For the two systems, BAT correctly predicts the number of threads as 8 and 32.

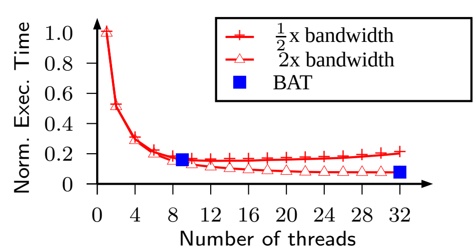

Figure 13. Performance of BAT as off-chip bandwidth is varied

[4] Counters to track the number of cycles the off-chip bus is busy already exist in some of the current processors. For example, the `BUS_DRDY_CLOCKS` counter in the Intel Core2Duo [23] processor and the `BUS_DATA_CYCLE` counter in Intel Itanium2 [16] processor . If such a counter does not currently exist in the system, then the performance monitoring framework can easily be extended to report this information.

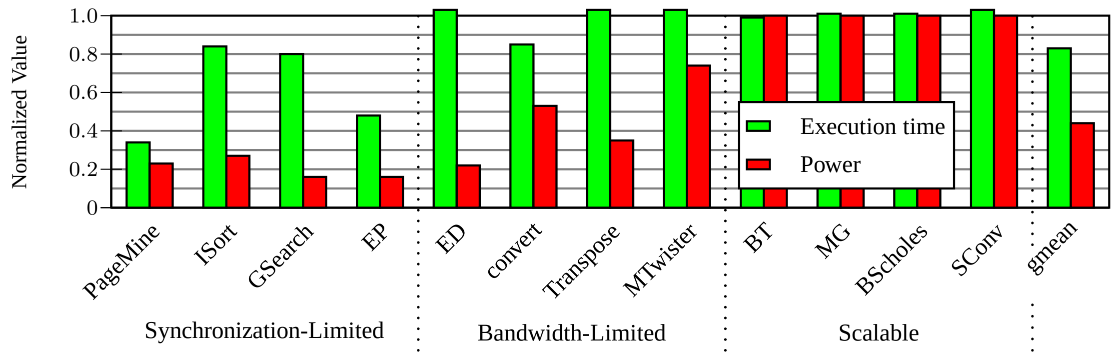

Figure 14. Execution time and power of (SAT+BAT) normalized to 32 threads

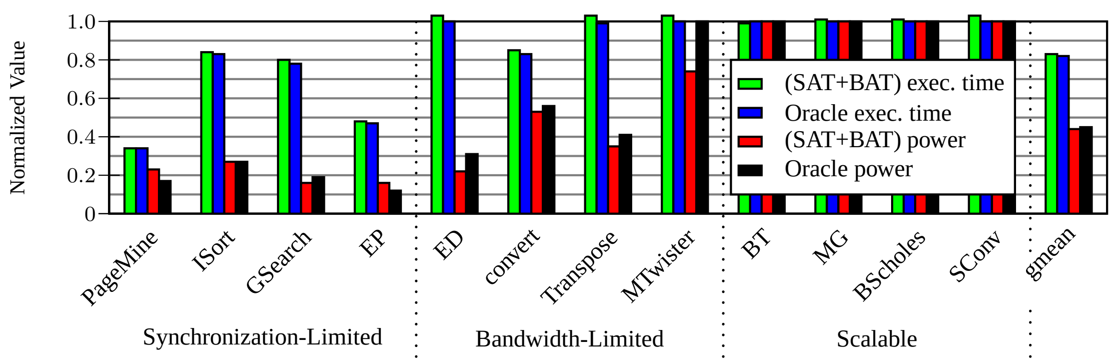

Figure 15. Execution time and power for (SAT+BAT) and an oracle scheme. The values are normalized to 32 threads.

6. Combining SAT and BAT using FDT

SAT reduces execution time and power consumption for data-synchronization limited workloads while BAT saves power for bandwidth-limited workloads. Both techniques can be combined within the FDT framework. The best number of threads computed by BAT and SAT for a given application can be different. In such cases, we choose the minimum of the two values. It can be proved that choosing the minimum of the two values minimizes overall execution time. The proof is included in the Appendix.

6.1 Implementation

To combine SAT and BAT the values of P_{CS} and P_{BW} are computed at runtime using the FDT framework. The implementation remains the same as explained in Section 4.2 and Section 5.2, except for three differences. First, the training loop measures the inputs required for both SAT and BAT. Second, the training loop does not terminate until the training for both SAT and BAT is complete. Third, the number of threads P_{FDT} is computed using equation 7.

$$P_{FDT} = MIN(P_{BW}, P_{CS}, num\_available\_cores) \quad (7)$$

6.2 Results

Figure 14 shows the execution time and power with the combination (SAT+BAT) normalized to the case of conventional threading which spawns as many threads as available cores (32 in our study). The applications are grouped depending on whether they are limited by data-synchronization (`PageMine`, `ISort`, `GSearch`, `EP`), or bandwidth (`ED`, `convert`, `Transpose`, `MTwister`), or neither (`BT`, `MG`, `BScholes`, `SConv`). The bar labeled *gmean* shows the geometric mean measured over all the 12 applications.

As expected, (SAT+BAT) combines the performance and power benefits of the two schemes. For all four data-synchronization limited applications, significant reduction in both execution time and power is achieved. For all four bandwidth limited applications, a significant power reduction is achieved. For `convert`, increasing the number of threads increases the L3 cache misses for each of the individual threads. Therefore, curtailing the number of threads to 17 reduces both power as well as execution time. For the four applications limited neither by data-synchronization nor off-chip bandwidth, FDT retains the performance benefits of more threads by always choosing 32 threads. Therefore, it affects neither the execution time nor the power consumption. On average, (SAT+BAT) reduces the execution time by 17% and power by 59%.

6.3 Comparison with Best Static Policy

We also compare (SAT+BAT) to an oracle scheme that statically sets the number of threads using off-line information. We implemented the oracle scheme by simulating the application for all possible number of threads and selecting the fewest number of threads required to be within 1% of the minimum execution time. Figure 15 shows the execution time and power for (SAT+BAT) and the oracle scheme, normalized to the case when there are 32 threads. For `MTwister`, (SAT+BAT) reduces power by 31% compared to the oracle scheme. `MTwister` contains two kernels. For the first kernel, the best number of threads is 32 and for the second kernel, the best number of threads is 12. The oracle scheme chooses 32 threads for the whole program, whereas, (SAT+BAT) chooses 32 threads for the first kernel and 12 threads for the second kernel. For all other applications, the execution time and power with (SAT+BAT) is similar to the oracle scheme. However, (SAT+BAT) has the advantage that it does not require any prior information about the application and is robust to changes in the input set (Section 4.4) and machine configuration (Section 5.4).

7. Related Work

With multi-core architectures becoming mainstream, industry has started to focus on multi-threaded applications. Several tools have been released in the recent past for improving the performance of such applications. For example, the Intel Vtune performance analyzer [15] enables the programmers to analyze and tune multi-threaded applications. Published guidelines [11] [13] that accompany such tools encourage programmers to carefully choose the number of threads taking thread-spawning overhead and synchronization overhead into account. OpenMP [8], a popular parallel programming paradigm, includes an option `OMP_DYNAMIC`, which allows the runtime library to dynamically adjust the number of threads. The Sun OpenMP compiler [33] uses this option to restrict the number of threads to the "number of physical CPUs on the machine". We are not aware of any OpenMP library in either industry or academia that uses run-time information to dynamically control the number of threads.

We discussed that increasing the number of threads may saturate or worsen performance. Other researchers have made similar observations. For example, Nieplosha et al. [28] studied the performance of scientific workloads on Cray MTA-2 and Sun Niagara. They show that some benchmarks get bandwidth limited on Niagara with as few as 8 threads. Furthermore, they also show that performance of some irregular scientific applications can decrease for more than 8 threads. Similar observations were made by Saini et al. [32] for a different machine.

The studies in [20] describe a compile time technique that takes communication overhead into account in order to improve thread scheduling in SMP systems. Nguyen et al. [27] and Corbalan et al. [6][7] investigate a system that measures the efficiency at different allocations and adjusts the job allocation in SMPs. However, the trade-offs in SMPs and CMPs are different. For example, the inter-processor communication delays are significantly more in case of SMPs compared to CMPs. Similarly, the off-chip bandwidth increases with the number of processors in SMP but may not in case of CMPs. Furthermore, the training time overhead for [27][6][7] increases with the number of possible processor allocations which causes the speedup to decrease as more processors are added to the system. Whereas, our technique requires only a single training loop to estimate the speedup for all possible number of threads, which substantially reduces training overhead.

McCann et al. [25] propose a scheduling mechanism to maximize throughput when the information about speedup versus number of cores for the competing application is known apriori. However, such information is dependent on input set and machine configuration. Our work does not assume such apriori knowledge. Brecht et al. [5] present a theoretical analysis to show that using job characteristics in making scheduling decisions is useful. However, both [25] and [5] have fundamentally different objective than ours in that our aim is to improve the performance of a *single* multi-threaded application.

FDT samples the application at runtime to estimate application behavior. Other researchers have also used temporal sampling for power and performance optimizations in CMPs. For example, Kumar et al. sample a given single-threaded application on different cores of a heterogeneous CMP [21][22]. However, they did these studies for multi-programmed workloads that were composed of different applications, had a predetermined number of applications (threads), and had no data-sharing. The objective of our study is to improve power and performance of multi-threaded applications for which the number of threads can be varied and can contain significant amount of shared data. Furthermore, studies [21] and [22] are restricted to heterogeneous CMPs and do not provide any power or performance benefits for homogeneous CMPs (used in our study).

8. Conclusion

Multi-threading enables applications to achieve high performance on chip-multiprocessors. However, the number of threads must be picked carefully to ensure high performance and low power. In this paper, we analyze the two major performance limiters of multi-threaded applications: data-synchronization and off-chip bandwidth. For applications that get limited by data-synchronization, increasing the number of threads significantly increases execution time as well as power. Similarly, for applications that get limited by off-chip bandwidth, increasing the number of threads increases on-chip power without providing any performance improvement. A mechanism that can control the number of threads based on the application behavior can reduce both execution time and power. To this end, we make the following contributions:

1. We propose *Feedback-Driven Threading (FDT)*, a dynamic technique to control the number of threads at runtime based on the application behavior.
2. We propose a simple analytical model that captures the impact of data-synchronization on execution time. Based on this model and the FDT framework, we develop *Synchronization-Aware Threading (SAT)*.
3. We propose a simple analytical model that estimates the minimum number of threads required to saturate the bus. Based on this model and the FDT framework, we develop *Bandwidth-Aware Threading (BAT)*.
4. We combine SAT and BAT within the FDT framework. Our evaluation, with 12 multi-threaded applications, shows that the combination reduces the average execution time by 17% and power by 59%.

The proposed techniques leverage existing performance counters and require minimal support from the threading library. Moreover, these techniques do not require any prior information about the application and are robust to variation in input set and machine configuration.

9. Future Work

We assumed that only one thread executes per core assuming no SMT on individual cores. However, the conclusions derived in this paper are also applicable to CMP systems with SMT-enabled cores. Our model for bandwidth utilization assumes that bandwidth requirement increases linearly with the number of threads, which ignores cache contention and data-sharing. More comprehensive models that take these effects into account can be developed. For non-iterative kernels, the compiler can generate a specialized training loop for estimating application behavior. Although this paper uses the FDT framework to implement only SAT and BAT, FDT is a generalized framework which can be used to handle other performance limiters such as contention for on-chip interconnect, cache, or DRAM banks.

Acknowledgments

Thanks to Eric Sprangle and Anwar Rohillah for discussions and insights on scalability of multi-threaded workloads. Aamer Jaleel and Hyesoon Kim provided valuable feedback which improved the quality of this paper. We thank the anonymous reviewers and members of the HPS research group for their suggestions. This research was supported by gifts from the Cockrell Foundation, Intel, and IBM. We also acknowledge the Texas Advanced Computing Center for providing computing resources.

References

[1] Advanced Micro Devices, Inc. White Paper: Multi-Core Processors – The next evolution in computing. 2005.

[2] D. an Mey et al. The RWTH Aachen SMP-Cluster User's Guide Version 6.2, May 2007.

[3] D. Bailey et al. NAS parallel benchmarks. Technical report, NASA, 1994.

[4] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 1958.

[5] T. Brecht and K. Guha. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27/28(4), 1996.

[6] J. Corbalan et al. Dynamic speedup calculation through self-analysis. Technical Report UPC-DAC-1999-43, UPC, 1999.

[7] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. *IEEE Trans. Parallel Distrib. Syst.*, 2005.

[8] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 1998.

[9] A. J. Dorta et al. The openmp source code repository. In *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2005.

[10] R. Ennals. Efficient Software Transactional Memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.

[11] M. Gillespie and C. Breshears(Intel Corp.). Achieving Threading Success. www.intel.com/cd/ids/developer/asmo-na/eng/212806.htm, 2005.

[12] J. Huh et al. Exploring design space of future CMPs. In *PACT '01*.

[13] Intel. Developing multithreaded applications: A platform consistent approach. www.intel.com/cd/ids/developer/asmo-na/eng/53797.htm, 2003.

[14] Intel. ICC 9.1 for Linux. http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm.

[15] Intel. Threading methodology: Principles and practices. www.intel.-com/cd/ids/developer/asmona/eng/219349.htm, 2003.

[16] Intel. Intel Itanium 2 Processor Reference Manual , 2004.

[17] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[18] K. Kennedy et al. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002.

[19] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[20] R. Kumar et al. Compiling several classes of communication patterns on a multithreaded architecture. In *IPDPS '02*, 2002.

[21] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36*, 2003.

[22] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA 31*, 2004.

[23] D. Levinthal. Introduction to Performance Analysis on Intel CORE 2 Duo Processors. "http://assets.devx.com/goparallel/17775.pdf", 2006.

[24] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 1998.

[25] C. McCann et al. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *Trans. Comp. Sys.*, 1993.

[26] R. Narayanan et al. MineBench: A Benchmark Suite for Data Mining Workloads. In *IISWC*, 2006.

[27] T. D. Nguyen et al. Maximizing speedup through self-tuning of processor allocation. In *Intn'l Parallel Processing Symposium*, 1996.

[28] J. Nieplocha et al. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Computing frontiers*, 2007.

[29] Y. Nishitani, K. Negishi, H. Ohta, and E. Nunohiro. Implementation and Evaluation of OpenMP for Hitachi SR8000. In *ISHPC 3*, 2000.

[30] Nvidia. CUDA SDK Code Samples. http://developer.download.-nvidia.com/compute/cuda/sdk/website/samples.html, 2007.

[31] R. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. *Technology@Intel Magazine*, Dec 2006.

[32] S. Saini et al. A Scalability Study of Columbia using the NAS Parallel Benchmarks. *Journal of Comput. Methods in Sci. and Engr.*, 2006.

[33] R. van der Pas et al. OMPlab on Sun Systems. Presentation at the International Workshop on OpenMP, 2005.

Appendix

When BAT and SAT are combined, the best number of threads for SAT (P_{CS}) and BAT (P_{BW}) can be different. In such case, choosing the minimum of the two values minimizes overall execution time. This can be proved as follows:

There are two cases:

1. $P_{CS} < P_{BW}$. This case is shown in Figure 16. The execution time decreases while the number of threads (P) is less than P_{CS} and then it starts to increase. When P is greater than P_{BW}, the execution time spent outside the critical sections becomes constant instead of reducing. Therefore, the overall execution time increases linearly with the number of threads. Thus, selecting P_{CS} minimizes the overall execution time.

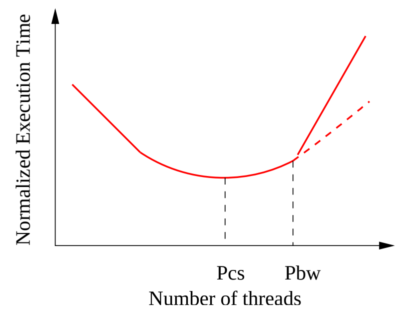

Figure 16. Overall execution time when $P_{CS} < P_{BW}$

2. $P_{BW} < P_{CS}$. This case is shown in Figure 17. The execution time decreases while P is less than P_{BW}. When P is greater than P_{BW}, the execution time spent outside the critical sections ceases to reduce, which means that the system becomes limited by critical sections sooner and effective P_{CS} shifts to P_{BW}. Therefore, after P_{BW}, overall execution time increases linearly with the number of threads. Thus, selecting P_{BW} minimizes the overall execution time.

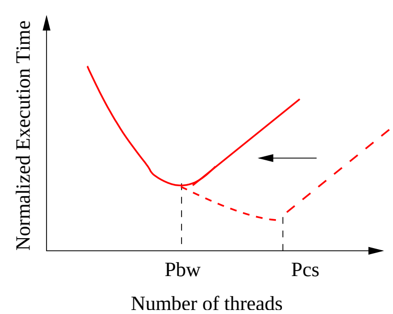

Figure 17. Overall execution time when $P_{BW} < P_{CS}$