BALANCING DATA LOCALITY AND PARALLELISM FOR IMPROVED APPLICATION PERFORMANCE ON MULTI-CORE PLATFORMS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Michael Jason Cade

San Marcos, Texas
May 2009

For my dad, who knew how everything worked but could not explain anything without drawing a picture.

ACKNOWLEDGEMENTS

Without the support and encouragement of my wife Tina, none of this would have ever happened. She taught me how to be a student.

I owe much gratitude to my committee chair Dr. Qasem, who possesses the rare combination of patience and wisdom that was required to guide me toward the answers without giving them all away. I have also had the exceptional good fortune of having two of the best teachers I've ever met, Dr. Guirguis and Dr. Hazlewood, on my committee.

Finally, Josh Magee deserves credit for sharing his knowledge of super pages as well as some of his code. Getting Josh's super page know-how provided the seeds for the biggest of the big 'aha' discoveries.

This manuscript was submitted on December 15, 2008.

TABLE OF CONTENTS

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1

INTRODUCTION

In the past, chip manufacturers have realized performance increases by packing more transistors on smaller chips. As modern designs approach the lower bounds on transistor size, as well as the upper bounds on cooling capacity, performance increases through chip area and transistor size reduction are becoming more difficult and more expensive to achieve. In response, manufacturers have turned their efforts toward integrating multiple simplified processing cores onto a single chip. By focusing development toward the duplication of relatively small and simple design units on a single chip, away from the larger, more complex, wide-issue, superscalar chip designs, advances in performance can again be realized. At the same time reductions in power consumption and design overhead can be achieved [1–3]. By adequately capitalizing on opportunities for parallelism, a chip multiprocessor (CMP) with two cores can achieve the same throughput of a single core processor while running at nearly half the frequency. This is of enormous importance in the current power and heat limited environment because lower frequencies imply lower power consumption and less heat [4–8].

Although CMP architectures promise large theoretical gains in performance potential, these improvements can not be attained by hardware alone. In order to realize the full potential of CMP systems much of the responsibility to find and exploit

opportunities for parallelism is now placed on software and programmers [9]. In many cases, the state of the art in performance enhancing tools lacks the sophistication required to make use of the full throughput and energy savings potential in modern CMP systems. The problem of finding and exploiting opportunities for parallelism in software is a difficult one and will require a great deal of effort if CMPs are to deliver at their performance and power capacities.

Further complicating the potential payoffs from the shift toward CMPs is the fact that at some level, memory resources will be shared among different processing cores. On many CMP systems, one or more levels of cache are shared among processing units [10]. When cache resources are shared among processors, data locality and parallelism become related. Consider the data-parallel execution model. Given some data and a task to be performed on the data, if multiple processing units are available, each unit can be assigned to perform the task on a subset of the data in parallel with the other processing units. As parallelism is increased by the addition of more cores or threads there is more contention for the shared memory resources. The benefit of increased parallelism is then gained with an associated cost of reduced data locality. On the other hand, if too much attention is given to data locality and execution threads are delayed or shut down to avoid cache eviction, the improvement in locality comes with the cost of unexploited parallelism. Since the relationship between locality and parallelism is sometimes contentious, care must be taken to find an appropriate balance between the two in order to optimize performance. Additionally, because the bandwidth to memory is shared by multiple processing units, underexploited data locality will force unnecessary memory accesses which equate to needless consumption of power. Thus, in CMP systems power efficiency has become tied to efficient use of the

memory hierarchy [11, 12].

When considering approaches to parallelization, there are essentially three models. Data parallelism is described above. Another model for parallelism is task parallelism. A task-parallel scheme divides the work to be done among processing units based on the tasks that need to be done to solve a given problem. For example, given a problem whose solution requires the solution of two subproblems, two processing units might be dispatched to solve the subproblems in parallel, reducing the overall time to solve the main problem. Finally, there is the pipelined parallel model. In pipelined parallelism, the problem is decomposed into a chain of interdependent stages. Processing units are dispatched to handle the stages in parallel. Each stage is then related to its temporal neighbors in a producer-consumer fashion. In other words, each stage consumes output from a previous stage and provides input to future stages.

In the past, pipelined parallelism has not received nearly as much research attention as data or task parallel models. However, this model of parallelism is likely to play a more significant role in parallelizing applications on multi-core systems because several factors make pipeline parallelism more relevant for CMPs. First, unlike data parallel models, pipeline parallelism can be effective in exploiting locality at the cache level since data is shared among the pipeline stages. This will be particularly useful for multi-core architectures with one or more levels of shared cache. Second, pipeline parallelism can be used to parallelize important classes of applications that exhibit producer-consumer behavior. Applications that fall within this domain include streaming multimedia such as MPEG decoders, iterative stencil computations, differential equation solvers and computational fluid dynamics code such as mgrid and swim from the SPEC benchmark suite. For many of these applications a data or task

parallel model is either difficult to construct or inefficient in practice. Finally, with the shift toward CMPs, parallelism has become more mainstream. Hence, the types of applications we want to parallelize are also changing. The pipeline parallelism model can be used to parallelize many applications that may otherwise appear to be completely sequential. In particular, the pipeline parallelism model could be generalized to parallelize loops with carried dependencies.

This thesis presents a model that captures the interaction between data locality and parallelism in the context of pipeline parallelism. To facilitate exploration of the relationship between parallelism and data locality, as well as the development of the parallelism-locality cache-reuse model, a synthetic benchmark has been constructed. The benchmark embodies the memory reuse patterns and exploitable parallelism characteristics of several applications that exhibit the general producer-consumer behavior at various stages of computation.

Experimental results suggest that consideration of the synchronization window, or the amount of work individual threads can be allowed to do between synchronizations, allows for parallelism- and locality-aware performance optimizations. The optimum synchronization window is a function of the number of threads and the size and configuration of the last-level of cache that is shared among processing units. By considering these two factors, the calculation of the optimum synchronization window incorporates parallelism and data locality issues for maximum performance.

CHAPTER 2

RELATED WORK

The related work is divided into three sections. First, work related to exploiting parallelism on CMPs is presented. Next we explore research that pertains to optimizing locality on CMPs. In the third section we will look at articles where optimizations for locality and the exploitation of parallelism are considered together.

2.1 Exploiting Parallelism on CMPs

That the industry would turn its efforts toward CMP architectures was predicted as early as 1989 [13]. In the last decade, the CMP platform has become a reality as all major manufacturers now produce some form of CMP architecture [2].

The exploitation of parallelism on these new architectures must become a key focus if the performance gains promised by CMPs are to be realized. In the past, legacy software has benefited from the hardware industry's ability to increase performance without changing the computational paradigm. Now, as the upper bounds of superscalar performance appear to be on the horizon and manufacturers focus on CMP architectures, software must adjust to a new parallel paradigm of computation if it is to realize advertised hardware performance increases [9]. Applications that take no measures to exploit the parallelism offered by CMPs may even see performance decrease on CMP platforms in spite of the architectures' touted performance

improvements. On the other hand, applications that exploit opportunities for thread-level parallelism may perform 50-100 percent better on multi-core systems than on their superscalar predecessors [3].

A new way of thinking in computational sciences presents a problem of daunting scale to those who will be tasked with updating and maintaining legacy code. Thies *et al.* provide a set of tools to facilitate the transformation of legacy code toward a more appropriate parallelism for CMP architectures. Here the authors have developed a set of parallel programming primitives to support pipeline parallelism [14]. While Thies's work concentrates on updating legacy code for the modern parallel paradigm, the primitives described here could easily be adopted by authors of new applications that are being written for CMP architectures.

Because of properties that are common to current CMP designs, parallel algorithms should be fine-grained and as asynchronous as possible if they are to take full advantage of multicore performance. Fine granularity is essential because cores in CMPs are associated with relatively small local memories. Therefore, the amount of data that a task or thread operates on must be small to reduce bus traffic and improve data locality. Asynchronicity will hide the latency of memory accesses as well as reduce the overhead incurred by synchronization points [15]. In [15] Buttari *et al.* present algorithms for the Cholesky, LU and QR factorizations where operations are represented as sequences of small tasks that can operate in parallel on square blocks of data.

In some cases, adding more execution threads is not beneficial or is prohibitively difficult to implement. Padopoulos *et al.* provide the database application domain as an example [16]. Here, the authors discuss two scenarios where more threads or cores are not beneficial to the overall performance of database management systems. The

first scenario involves a database system that is able to issue queries in parallel. In this case, parallelization will not be beneficial when the number of execution threads increases beyond a certain point. After some number of threads have been deployed, the synchronization required to keep shared structures within the database system consistent would prohibit performance gains. The second example involves simpler database management systems that do not easily support parallel queries. In these systems, implementing parallel threads of execution is prohibitively difficult. In either case, the authors propose to exploit additional cores by using them as intelligent prefetching agents that they call Helper Cores. Papadopoulos *et al.* claim that their methodologies show improvements even on architectures that employ hardware prefetching because of the expanded variety of prefetch patterns that are possible using the helper core methodology [16].

2.2 Data Locality Optimizations for CMPs

The body of research covering locality issues spans four decades. As long as the gap between memory and compute performance exists, data locality will continue to be an important consideration for CMP architectures.

The widening gap between memory and compute speed was an issue well before multicore architectures began to appear in the marketplace. The concept of machine balance involves balancing an application's ratio of memory accesses to compute operations per cycle with the ratio that the architecture can physically perform in a cycle. In 1994, Carr and Kennedy pointed out that programmers had begun making high-level code transformations to achieve better machine balance within loop nests. The authors claim that this introduces an undesirable level of machine-dependence in

high-level code, which decreases the portability. They also contend that readability and maintainability is adversely affected by such high-level code transformations. Carr and Kennedy further claim that these machine-balance optimizations should be the responsibility of compiler tools and that by moving these transformations into compilers, high-level code becomes easier to write and more portable [17].

The machine balance issue becomes even more difficult in current multicore architectures because of the larger disproportion between the bus bandwidth and the compute power provided by multiple cores. Consider the Intel Clovertown processor. Each Clovertown chip contains four cores, each core capable of 10.64 GFlops. In total, one Clovertown chip has a theoretical peak of 42.56 GFlops. However, the bus bandwidth tops out at 10.64 GB/s which could provide at most 1.33 GWords/s (where a word is a 64 bit double). Since one core that is executing a workload heavily biased toward floating-point operations is more than enough to saturate the memory bus, adding additional cores to execute similarly biased workloads without making an attempt to exploit data locality would provide no significant benefit [15]. Buttari *et al.* describe such an effort to exploit data locality in linear algebra algorithms by representing operations as sequences of small tasks and dynamically scheduling them. Part of Buttari's effort relies on reorganizing matrices into a block data layout rather than the column major format found in FORTRAN arrays or the row major format found in C arrays [15].

Data locality issues incur an additional layer of complexity within the CMP domain since in many cases, cores on a CMP share some of the cache facilities in addition to sharing the memory interface [18]. Future architectures are likely to continue to share some cache facilities among cores on a die. Research has shown performance increases

of several orders of magnitude in data-intensive workloads on shared last level cache arrangements versus CMPs whose cores have private last level caches. Jaleel *et al.* consider bionformatics workloads specifically. In bionformatics large amounts of genetic data are mined to discover knowledge. Jaleel asserts that if multi-threaded applications tended to be data-independent, exhibiting little or no sharing, then unique cache facilities for each core would be the most appropriate approach. However, if applications tend toward sharing some amount of data (as they do within the bionformatics domain), sharing the last-level cache is a more appropriate approach. This is simply because if each core had it's own unique cache hierarchy, memory blocks shared among cores would need to be duplicated in each cache which reduces overall cache capacity. Jaleel *et al.* report a reduction in memory bandwidth demands by factors of 3-625 when the last-level cache is shared versus private last-level caches [19].

While on the surface it may seem that if cache resources are to be shared by multiple processing cores the size of the caches should grow in a linear fashion corresponding to the number of processors. However, it has been shown that, given an appropriate parallelization schedule, the number of cache misses will go down in a CMP system with a shared cache that is only additively larger than a single core chip's cache running the same workload [20].

As more cores become available in CMP architectures, there are more compute resources available to running processes. At the same time, more cores mean more contention for shared memory resources and interfaces [21, 22]. Thus, contention for these shared on-chip memory resources and interfaces becomes one of the key issues in CMP performance optimization. Much work has been devoted to ascertaining and exploiting the opportunities for data locality within individual loop nests [23–25]. Li

and Kandemir offer a more global loop-based locality approach that they apply to heterogeneous multi-core architectures. In their work, Li and Kandemir propose a system for considering all of the loop nests in an application simultaneously, thereby accounting for the interactions among different loop nests. They claim that their approach provides significant benefit in power and performance over the conventional loop based approach [26].

Nikolopoulos proposes a methodology for dynamically partitioning a shared cache among threads in simultaneous multithreaded architectures. The proposed methodology involves using two tile sizes, one that occupies the entire cache and another that occupies a fraction of the cache inversely proportional to the number of threads sharing the cache. By switching between the two tile sizes dynamically at run-time Nikolopoulos' methodology reduces unnecessary conflict misses that would otherwise occur when two or more threads of execution attempt to utilize loop tile sizes that would occupy the entire shared cache. Nikolopoulos notes that this dynamic tiling implementation would benefit all processors that make use of a shared data cache, including CMPs [27].

By exploiting the reuse of data that is already in cache wherever possible, two issues can be addressed. First, execution time is decreased because memory latencies are not incurred a second time when data that has already been fetched can be reused rather than being cast out and fetched again later. Second, pressure on memory bandwidth is reduced because data does not have to be fetched multiple times. There are techniques that address the latency problem but do not address the bandwidth problem. For example prefetching reduces the effects of memory latency by performing loads before data is needed but prefetching does not address the bandwidth issue [28].

Ding and Kennedy present a two-step approach to reduce memory bandwidth requirements within a workload by exploiting data locality. The first step involves fusing computations on the same data to increase the amount of temporal locality. The second step involves reorganizing the data layout to group data used by the same computation to increase spatial locality [29].

Since the amount of activity on the memory bus can be correlated to power consumption [11, 12, 30], reducing pressure on the memory interface by exploiting data reuse has a positive effect on overall power consumption. Daylight *et al.* introduce data structure transformations that allow traversal through large data structures with fewer memory accesses and thereby decrease power consumption in embedded multimedia software [31].

2.3 Integrating Parallelism and Locality Optimizations for CMPs

Within the context of CMP architectures, the literature shows that software compiled with special attention given to parallelism as well as data locality issues run with significant power and performance benefits over software that does not [26].

One approach that incorporates parallelism with consideration for data locality is the stream programming model. Stream programming is a different programming paradigm altogether. Rather than the von Neumann model, stream programming uses a data-flow model. In this data-flow model, some quantity of data is loaded into local memory as a bulk data load, operations are performed in parallel on the loaded data and then the resulting data is stored back as a bulk data store. Streamware is a flexible software system proposed by Gummaraju *et al.* that can be used to map stream programs onto a variety of multicore systems. Using the stream programming model,

the authors are able to leverage the parallelism inherent in multicore systems while still accounting for data locality issues. They also claim that their tools and methodologies are not restricted to traditionally streamed applications like media and image processing software but can be extended to general-purpose data-intensive applications. [32].

Vadlamani and Jenks' *Synchronized Pipelined Parallelism Model* (SPPM) is another effort to incorporate parallelism with data locality in a way that is appropriate for CMPs [18]. SPPM applies to an important class of workloads where the problem can be seen as a sequence of interdependent stages. The computation performed at each stage is dependant on output from a previous stage. When computation is completed at the current stage, results are passed on as input to some future stage. In other words, the type of problems that SPPM is successfully applied to can be thought of as chains of computational stages, each stage having a producer-consumer relationship with its temporal neighbors. The heart of the SPPM algorithm involves processing units working simultaneously through the dataset in a temporally staggered arrangement. This allows processors that are ahead in the dataset to act as prefetch engines as well as producers of data for the processor units that follow. Research shows that a workload employing the SPPM model of parallelization incurs an overall cache miss rate and memory bus utilization that is similar to the cache misses and bus utilization if it were to be run strictly sequentially while realizing the performance increases that come with parallelization. This is an improvement over the standard spatial decomposition model of parallelization, which suffers from inefficient use of shared caches and increased pressure on the memory bus by failing to consider data locality [18].

2.4 Limitations of the current body of literature

A great deal of research effort has been directed toward finding and exploiting opportunities for parallelism. Methodologies and models for parallelism have been studied on virtually every architectural model which is capable of concurrent processing. There is also a wealth of research regarding data locality and methods for exploiting data locality to improve both performance and power consumption. Again, issues surrounding locality have been well studied on many different architectural models.

The current body of literature is somewhat sparse when these two issues are considered together. The relationship between locality and parallelism is of considerable importance in light of modern CMP architectures, especially on CMP architectures where there is some sharing of cache. If processing units within a CMP system share cache at some level, parallelism and data locality become inextricably intertwined. There must be more research directed toward developing models to sufficiently explain the behavior surrounding this relationship so that workloads may be more easily optimized for these new CMP architectures.

CHAPTER 3

A MODEL FOR RELATING PARALLELISM AND LOCALITY

The parallelism-locality model presented here has been developed through empirical study of a synthetic benchmark. The benchmark encapsulates the memory reuse patterns and parallelism characteristics of many workloads that exhibit temporal producer-consumer behavior at some point during execution.

It is important to note that all of the calculations and models here presuppose a contiguous allocation of memory for the dataset. Also, when discussing cache, this work refers to the last level of the cache hierarchy.

The major computational component of the synthetic benchmark involves a large dataset that can be thought of as a three-dimensional physical space. During execution, this dataset is updated iteratively. During each iteration, each member of the dataset is updated as a function of its adjacent neighbors and it's current value, such that the data at $time_{n+1}$ is a function of the data at $time_n$.

Since the data in position x at $time_{n+1}$ is dependant on the data at position x and all of x's neighbors at $time_n$, the data at x cannot be updated to its $time_{n+1}$ value in place without storing the $time_n$ value for use by its neighbor's update functions. One approach to this type of iterative, time-step update is to create a duplicate of the dataset. One dataset represents the current state, or $time_n$, and the other represents the state of the data at $time_{n+1}$, or the next state. Let the current state dataset be

called D_n and the next state dataset be D_{n+1}. Data is read from the D_n dataset, the update function is applied, and the result is written to the corresponding element in the D_{n+1} dataset. When the entire D_{n+1} dataset has been populated it represents the new current state of the data. Thus, D_{n+1} becomes D_n and vice-versa. The next iteration repeats the behavior of the previous iteration, reading from the new D_n (formerly D_{n+1}) and writing to the new D_{n+1} (formerly D_n). This execution model fits well within the pipeline parallelism model, where one stage of execution produces data for future stages.

A wholly sequential approach involves one thread of execution iterating through the entire D_n dataset, calculating the next state values for each element, and writing those values to the D_{n+1} set. Once the entire next-state set has been populated, the D_n set and the D_{n+1} set are swapped, and the process repeats itself to calculate the next state. This continues until the desired number of iterations are completed and the data is in its final state, D_m.

At the other extreme, up to m threads of execution may be deployed to update the data in parallel so long as care is taken to ensure that threads operating on future updates do not collide with threads working on past updates. In this parallel approach, the two datasets can be thought of as the even iteration data and the odd iteration data. For example, $thread_0$ would be deployed, reading from the D_{2n} set and writing to the D_{2n+1} set. After $thread_0$ had read enough of the D_{2n} set and written enough of the D_{2n+1} set, $thread_1$ could begin reading from the D_{2n+1} set and writing to the D_{2n} set. At some point, $thread_1$ would be far enough along to allow $thread_2$ to begin reading from the D_{2n} set and writing to the D_{2n+1} set. This would continue until either m threads had been deployed or, $thread_0$ reached the end of the D_{2n} set and could begin

again at the top of D_{2n} making the update for the next necessary state (Figure 3.1).

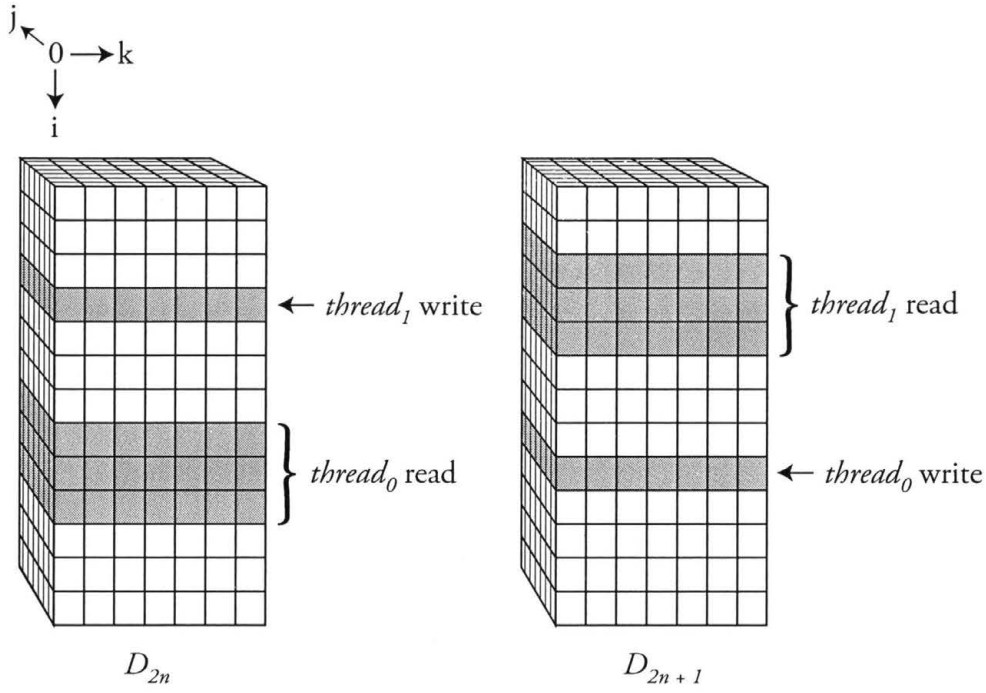

Figure 3.1: Multiple threads may operate simultaneously on the data so long as they do not collide. Here, $thread_0$ reads from D_{2n} and writes to D_{2n+1}. A second thread, $thread_1$, may be deployed that reads from D_{2n+1} and writes to D_{2n} so long as mechanisms are in place to prevent $thread_1$ from overwriting data that $thread_0$ has not yet consumed.

In order to preserve data dependencies, the threads must be prevented from colliding. If a thread that is making an update for $time_{t+1}$ gets too close, in terms of data, to the thread that is making the update for $time_t$ the data necessary for the $time_t$ update will be overwritten by the update for $time_{t+1}$ and the final result will be corrupted. Thus, some synchronization device must be implemented to keep the threads sufficiently far apart.

In the synthetic benchmark presented here, a second synchronization limit is put on threads operating in parallel. In addition to preserving data dependence by preventing threads from getting too close together, an attempt is made to exploit data locality by preventing threads from getting too far apart. This introduces the concept

of an execution window. Synchronization for the synthetic benchmark's threads is performed such that the threads of execution operate within a minimum and maximum window size (W_{min} and W_{max}, respectively), measured by data distance on the innermost index of the dataset (Figure 3.2).

The execution window allows for the concept of a synchronization granularity. The synchronization granularity is the number of iterations that either thread may safely execute and still maintain the execution window constraints. If $thread_0$ and $thread_1$ are the ideal distance from one another, then either thread may execute no more than $1/2(W_{max} - W_{min} - 1)$ iterations on the innermost index of the dataset without regard for the other thread's progress. The synchronization granularity, W_i, is expressed in the following equation.

$$W_i = \frac{W_{max} - W_{min} - 1}{2} \tag{3.1}$$

After each W_i iterations on the innermost dimension of the dataset, there is a barrier that forces the threads to return to the ideal separation distance. As W_i grows, so does the interval between thread synchronizations. Consequently, as the synchronization interval size grows, thread synchronization overhead decreases.

The optimal synchronization interval would maximize the exploitation of data locality while incurring the least amount of synchronization overhead. Synchronization overhead decreases as the synchronization interval grows, and the synchronization interval grows as W_{max} and thus W_i increase. For these two reasons, the synchronization interval should be as large as possible. However, if the synchronization interval between $thread_0$ and the last thread ($thread_n$) is too large, threads coming

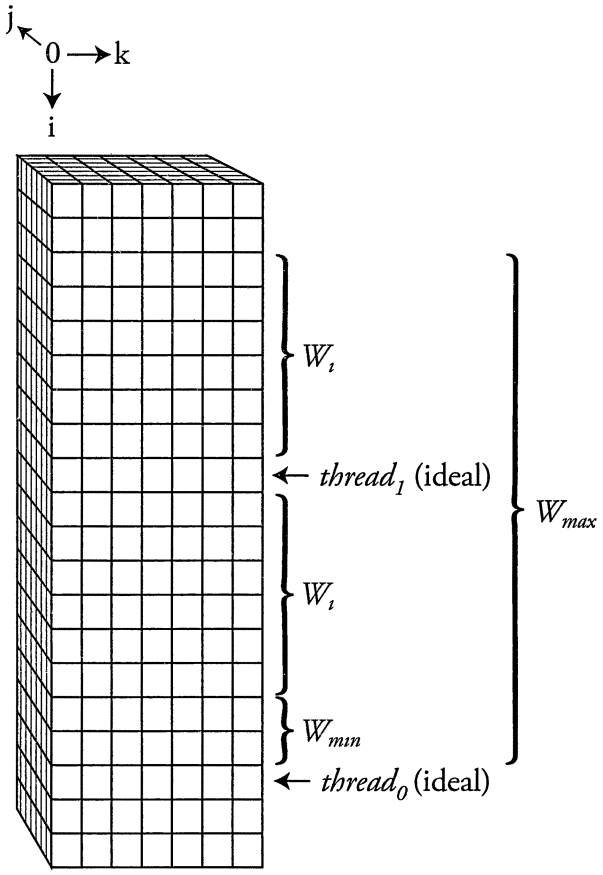

Figure 3.2: Here, W_i is the synchronization granularity, W_{min} is the minimum window size and W_{max} is the maximum window size $(2W_i + 1 + W_{min})$.

after $thread_0$ will compete with $thread_0$ for cache rather than being able to capitalize on the data that is already in cache from $thread_0$'s previous accesses. For this reason, the distance between $thread_0$ and $thread_n$, in terms of the amount of data between them, should be within the size of the cache. Ultimately then, the optimal synchronization interval is the largest interval such that the data between $thread_0$ and $thread_n$ fits within cache. In other words, the ideal situation has $thread_n$ accessing the oldest data in cache while $thread_0$ accesses the newest data in cache. The fraction of the cache between $thread_0$ and $thread_n$ can be expressed by the following equation:

$$U = \frac{\alpha}{C} \tag{3.2}$$

where U is the amount of cache consumed by data that $thread_0$ has accessed but that $thread_n$ has not yet accessed, α is the cost, in terms of cache, of updating the elements that $thread_n$ has not yet updated in this pass but $thread_0$ has finished updating for this pass through the dataset, and C is the total size of the cache.

The cache cost factor can be expressed as:

$$\alpha = (T-1) \times W_i \times \beta \times \gamma \tag{3.3}$$

where T is the total number of threads deployed to update the dataset, β is the number of elements in the dataset for each unit of W_i and γ is a measure of the cost, in terms of cache, of evaluating the next state for one element in the dataset. Here, T must be greater than or equal to two, since if there are fewer than two threads there can be no cache consumption cost between the first thread and the last thread. Also worth noting here is that because data moves in and out of cache on the granularity of cache lines rather than bytes, γ must be considered in terms of cache lines and not bytes.

The γ term may be expressed in the following way:

$$\gamma = \frac{l}{S/d} \tag{3.4}$$

where l is the number of cache lines accessed when calculating the next state for one element of the dataset, S is the size of one cache line and d is the size of one element of the dataset.

Finally, U may be explicitly defined as:

$$U = (T-1)\frac{W_i \times \beta \times \frac{l}{S/d}}{L}$$

$$= (T-1)\frac{W_i \times \beta \times l}{S/d \times L}$$

$$= (T-1)\frac{W_i \times \beta \times l \times d}{S \times L} \tag{3.5}$$

where L is the total number of lines in cache. L replaces C in equation 3.2 since here cache costs are defined in terms of cache lines and not bytes. However, since L can be rewritten as C/S, where C is the total last-level cache size in bytes, we can simplify equation 3.5 to:

$$U = (T-1) \times \frac{W_i \times \beta \times l \times d}{C} \tag{3.6}$$

For the synthetic benchmark described in this work, the model may be further refined by replacing β with the second and third dimensions of the dataset because here W_i is measured on the first dimension of the dataset. Therefore, for this benchmark, U is defined by the equation below.

$$U = (T-1) \times \frac{W_i \times j \times k \times l \times d}{C} \tag{3.7}$$

From equation 3.7 the following can be derived to calculate the optimal synchronization granularity (W_i).

$$\frac{1}{W_i} = \frac{1}{U} \times (T-1) \times \frac{j \times k \times l \times d}{C}$$

$$W_i = U \times \frac{C}{(T-1) \times j \times k \times l \times d} \tag{3.8}$$

The ultimate goal, as stated earlier, is to maximize U without exceeding the capacity of the cache. By maximizing U, the synchronization overhead is reduced to the minimum value that still allows exploitation of the data locality inherent in the pipeline parallel execution model. Thus, the target for U is one, yielding the following equation to calculate the ideal W_i.

$$W_i = \frac{C}{(T-1) \times j \times k \times l \times d} \tag{3.9}$$

For example, consider a three dimensional iterative stencil algorithm applied to a dataset has a second and third dimension of 64 elements, each element being 8 bytes. Using the cache-use model above with a four mega-byte last-level cache shared between two threads and 64 byte cache lines, the ideal synchronization interval is calculated to be 14.2 units on the innermost dimension of the dataset.

$$W_i = \frac{2^{22}}{(2-1) \times 2^6 \times 2^6 \times 9 \times 2^3}$$

$$W_i = 14.2 \tag{3.10}$$

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 Testing Environment

Experimental results were collected by running the synthetic benchmark on a 2.33GHz Intel® Core™2 Duo with 4MB of L2 cache shared between the two cores. The benchmark tests were run under Linux 2.6.24 with the perfctr module installed. Cache and TLB data were collected with PAPI. PAPI is a platform-independent specification for an interface to hardware performance counters. These counters exist on modern microprocessors as a small set of registers that can be used to count the occurrence of specific events related to a processor's function [33]. For the synthetic benchmark, there are two important timing measurements $time_{wall}$ and $time_{wait}$. $time_{wall}$ is simply the amount of wall-clock time spent in the function that performs the iterative stencil algorithm. This is measured with two calls to gettimeofday, one at the function's entry point and another at the function's exit point. $time_{wait}$ is a measure of the length of time threads spend waiting at synchronization barriers. It is also measured with two calls to gettimeofday, one immediately before entering a barrier and another immediately after exiting a barrier. Synchronization counts were collected with counters in the benchmark code itself.

A dataset of substantial size is necessary to smooth the data collected and to emphasize differences in measurements under different testing conditions. These goals

can also be supported by iterating over the dataset a large number of times. In contrast, the dataset and iteration count should be small enough to allow data to be collected in a reasonable amount of running time. For each test case presented here, the dataset is a $512 \times 64 \times 64$ array of doubles. The array is updated using a three-dimensional iterative stencil algorithm where the next state of each element is dependent on the current state of the element and the current state of all of its neighbors. The entire dataset is updated iteratively in this way 400 times for each test. These values represent a good compromise between a sample large enough to reduce the noise in the data and a sample size small enough to allow data to be collected in a reasonable amount of time.

Multi-threaded test cases involve two parallel threads of execution. Each thread is bound to a unique core using calls to pthread_setaffinity_np so that each thread is explicitly scheduled on one and only one processing unit. The minimum window size is fixed at the smallest value that preserves data dependencies. For the synthetic benchmark described here, the minimum window size is fixed at two. Since the minimum window size is fixed, the independent variable is the maximum window size.

Synchronization granularity, or W_i, is directly related to the maximum window size; increasing W_{max} increases the synchronization granularity. Equation 3.1 shows the relationship between W_{max} and W_i. To evaluate the model presented in equation 3.9, the benchmark is repeated over a range of maximum window sizes. The results presented are the averaged results of five different test runs under the same testing conditions.

Table 4.1: Performance of sequential versus parallel execution on four-kilobyte pages.

	Sequential	Parallel ($W_{max} = 31$)		
		$thread_0$	$thread_1$	sum
Total Cycles	1.20×10^{11}	5.81×10^{10}	5.64×10^{10}	1.14×10^{11}
Wall Clock Time	51.61 S	25.58 S	25.58 S	25.75 S
TLB Misses	3.56×10^{6}	1.84×10^{6}	1.82×10^{6}	3.66×10^{6}
Data Miss Rate	0.003	0.0031	0.0007	0.0019

4.2 Four-Kilobyte Pages

Initially, a series of tests was conducted with standard four-kilobyte pages. The parallelized code showed a performance improvement over strictly sequential code (see table 4.1). Although performance increased over sequential execution, the parallelized code did not exhibit the expected optima when manipulating the synchronization window sizes.

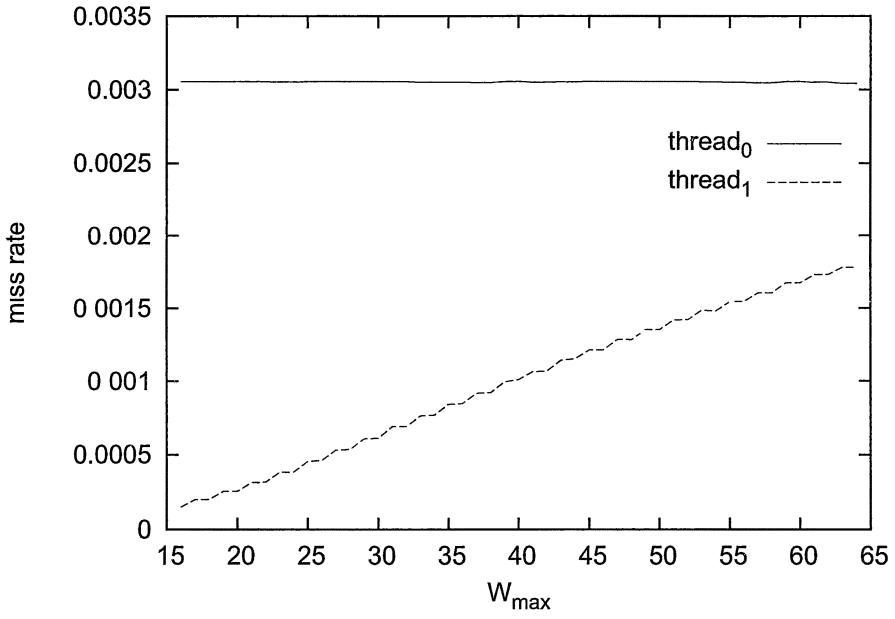

Figure 4.1: Data miss rate for synthetic benchmark with four-kilobyte pages.

Intuitively, there should be an optimal synchronization window size, where the

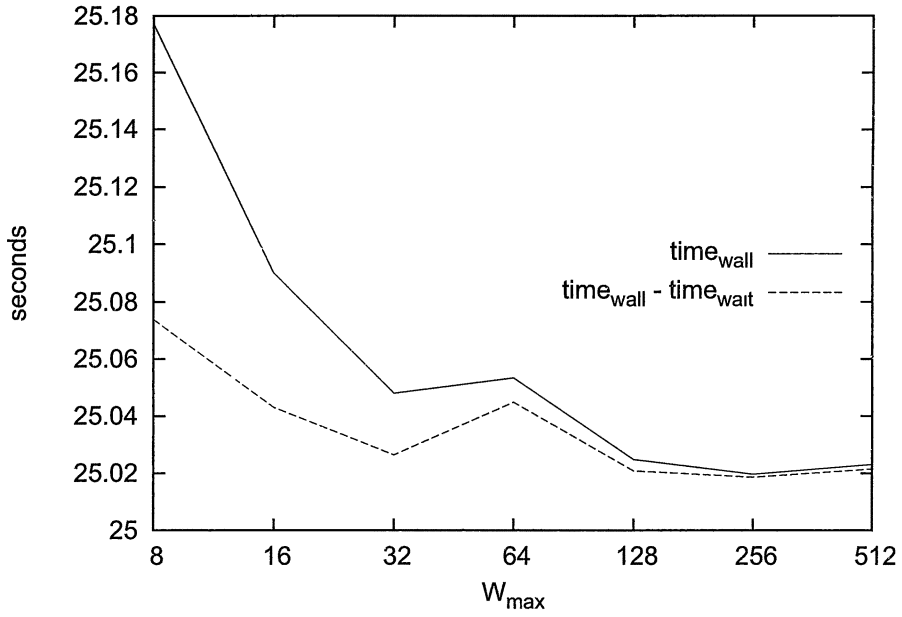

Figure 4.2: $thread_0$ time for synthetic benchmark with four-kilobyte pages.

maximum amount of data locality is exploited while incurring the minimum amount of synchronization overhead. Based on equations 3.9 and 3.1, that optimal point should be at a W_{max} of 31. However, in the tests where the standard four-kilobyte pages were used, the increase in miss rates is linear relative to the increase in W_{max} near the theoretical optimal synchronization window size (see Figure 4.1).

Table 4.2: Performance of 4KB versus 16MB pages ($W_{max} = 31$).

		Total Cycles	Wall Clock Time	TLB Misses	Data Miss Rate
4KB pages	$thread_0$	5.81×10^{10}	25.58 S	1.84×10^6	0.0031
	$thread_1$	5.64×10^{10}	25.58 S	1.82×10^6	0.0007
	sum	1.15×10^{11}	25.75 S	3.66×10^6	0.0019
16MB pages	$thread_0$	5.78×10^{10}	24.96 S	5.31×10^4	0.0031
	$thread_1$	5.62×10^{10}	24.96 S	7.75×10^4	0.0002
	sum	1.14×10^{11}	25.13 S	1.31×10^5	0.0017

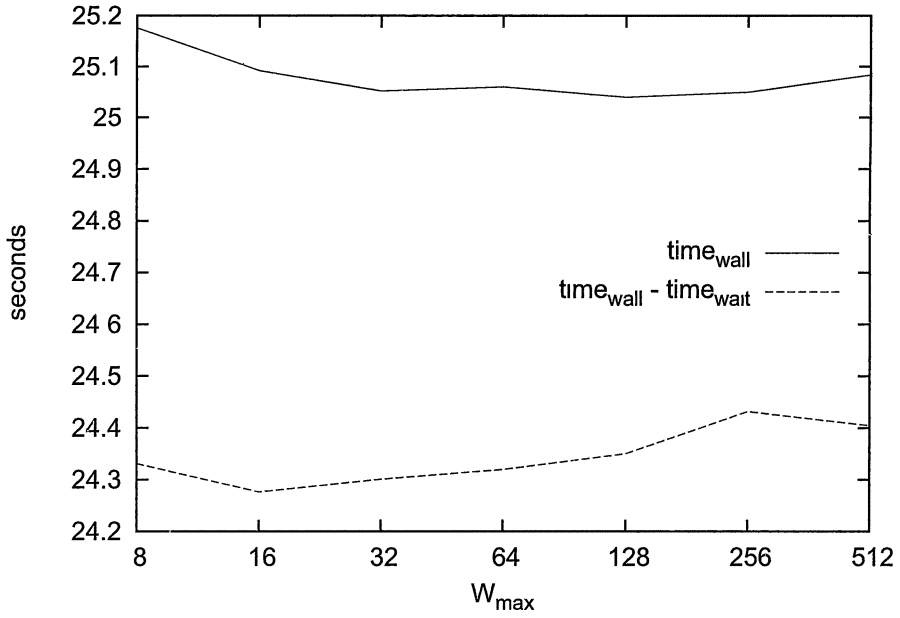

Figure 4.3: $thread_1$ time for synthetic benchmark with four-kilobyte pages.

The wall-clock time results with four-kilobyte pages were similar. While figures 4.2 and 4.3 show that the wait time does increase on $thread_0$ at window sizes smaller than the theoretical optimum, there appears to be no minima on $thread_1$, where it is expected to be, which would clearly point to an optimal synchronization window size.

The inconclusive behavior of the benchmark running with four-kilobyte pages is attributed to the small page size. At $512 \times 64 \times 64$ doubles, the dataset occupies 2^{12} four-kilobyte pages. Since there are two copies of the dataset, one for the current state and one for the next state, the entire dataset covers 2^{13} four-kilobyte pages. Because the data covers more than 8000 pages, and these pages are scattered around the real address space, the access pattern of real addresses is likely to be somewhat disorderly when compared to the access pattern of a contiguously mapped dataset. If the access pattern of real addresses is disorderly data blocks are less likely to be mapped into cache sets in an orderly way. Therefore, the lack of a strictly linear access pattern, in terms of real addresses, results in utilization of cache that is suboptimal because it can

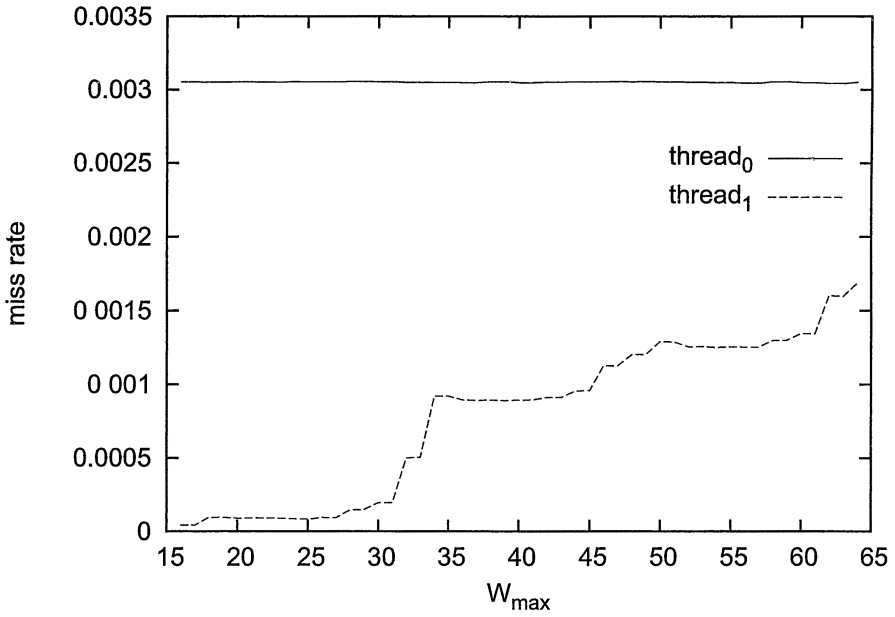

Figure 4.4: Data miss rate for synthetic benchmark with 16-megabyte pages.

cause some cache sets to be more heavily utilized than others. When higher demand is placed on one cache set, the rate of conflict misses will be inflated for that set and the overall miss rate is then increased.

In addition to utilizing cache inefficiently, spreading the dataset across multiple pages caused an increase in TLB misses (Table 4.2). It was hypothesized that the time cost to service TLB misses was dominating the overall execution time and thus responsible for the inconclusive results in terms of execution time.

4.3 16-Megabyte Pages

After getting poor results with four-kilobyte pages an attempt was made to reduce TLB misses and perhaps improve performance in terms of time by allocating the dataset on 16-megabyte pages. With the larger pages, TLB misses are reduced by more than 96 percent. The cache miss rate for $thread_1$ is also reduced by more than 70 percent with 16-megabyte pages versus four-kilobyte pages which results in a ten percent decrease in

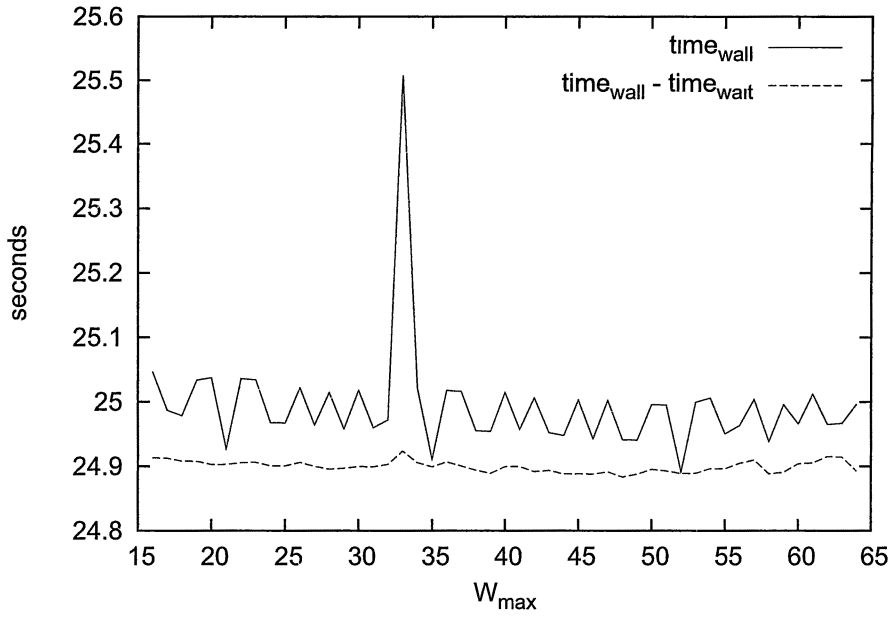

Figure 4.5: $thread_0$ time for synthetic benchmark on 16-megabyte pages.

the overall cache miss rate (Table 4.2). It is worth noting that the miss rate for $thread_0$ is the same for both page sizes. This is due to the fact that the first thread will encounter the same number of compulsory misses for data accesses, regardless of the way the data maps into cache sets.

The reason for the decrease in TLB misses is that the entire dataset fits on one 16-megabyte page. Because the TLB on the Core™ 2 Duo has 256 entries, there would be a considerable amount of thrashing in the TLB if the dataset were spread across a large number of pages, as it is with four-kilobyte pages. However, since one copy of the dataset is fully contained within one 16-megabyte page, there is far less contention for TLB entries and therefore far fewer TLB misses when the data is mapped onto 16-megabyte pages.

The reduced miss rate for $thread_1$ on 16-megabyte pages is attributable to the fact that a 16-megabyte page is mapped into one contiguous block of memory. When the dataset is mapped into memory on a 16-megabyte page, all of the data accesses for the

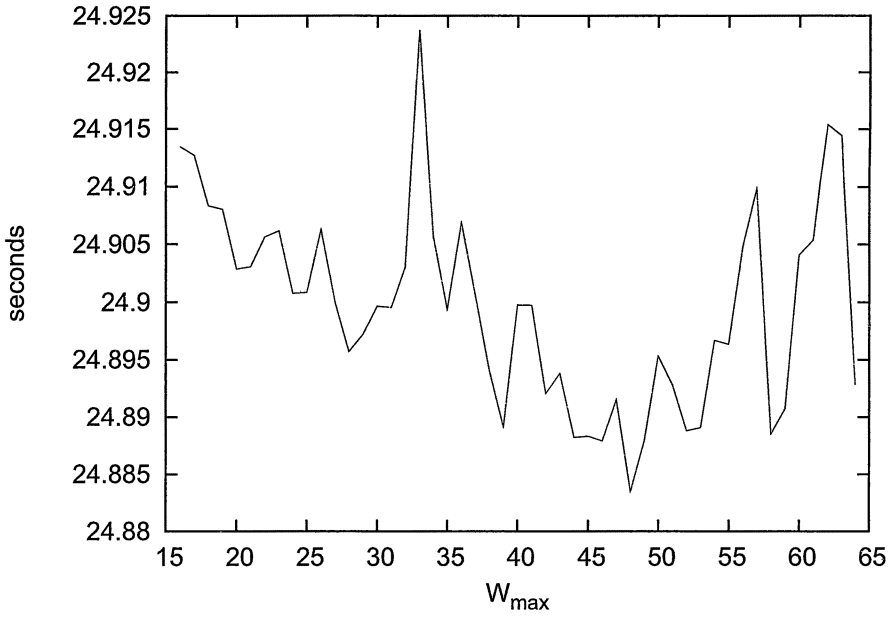

Figure 4.6: $thread_0$ time minus $thread_0$ wait time for synthetic benchmark on 16-megabyte pages.

stencil algorithm happen in a linearly ordered way in terms of real addresses. When a large block of real addresses are accessed linearly, the load is spread evenly across all of the sets in cache. This allows full utilization of the cache's capacity. Because maximal utilization of cache capacity is required for the model presented in equation 3.9, it is only with 16-megabyte pages that we see the model supported in the test results.

Figure 4.4 illustrates the theoretical optima at or near a W_{max} of 31. In test cases with 16 megabyte pages, W_{max} of 31 allows for the largest synchronization interval that also exploits the data locality inherent in the two-threaded, pipeline parallel execution of the iterative stencil algorithm.

Figures 4.5 and 4.7 show the difference between the wait time and the wall-clock time on each thread. The lower line in each plot is the total wall-clock time minus the total wait time on the thread. This can be thought of as the amount of time that the thread is actively doing meaningful work regarding the synthetic benchmark. It is

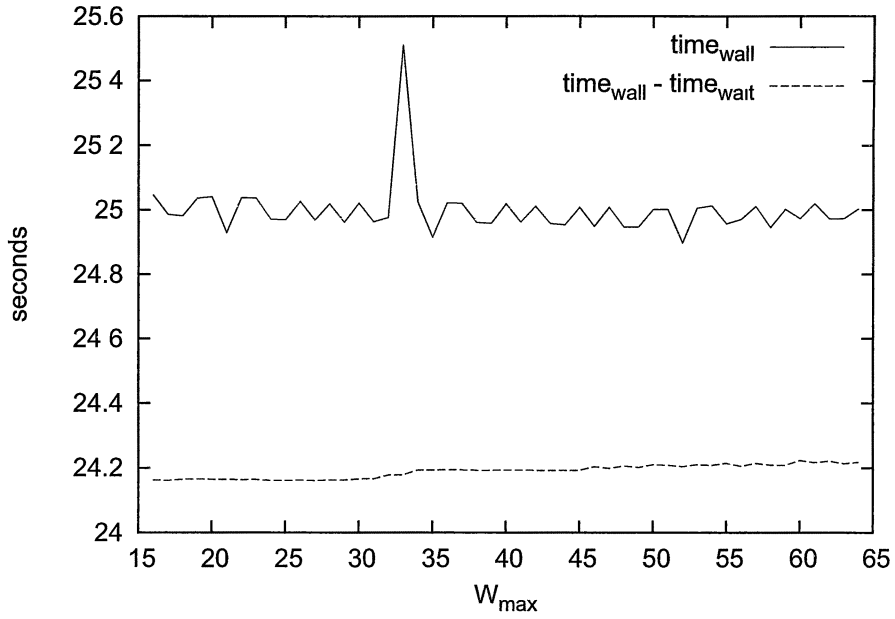

Figure 4.7: $thread_1$ time for synthetic benchmark on 16-megabyte pages.

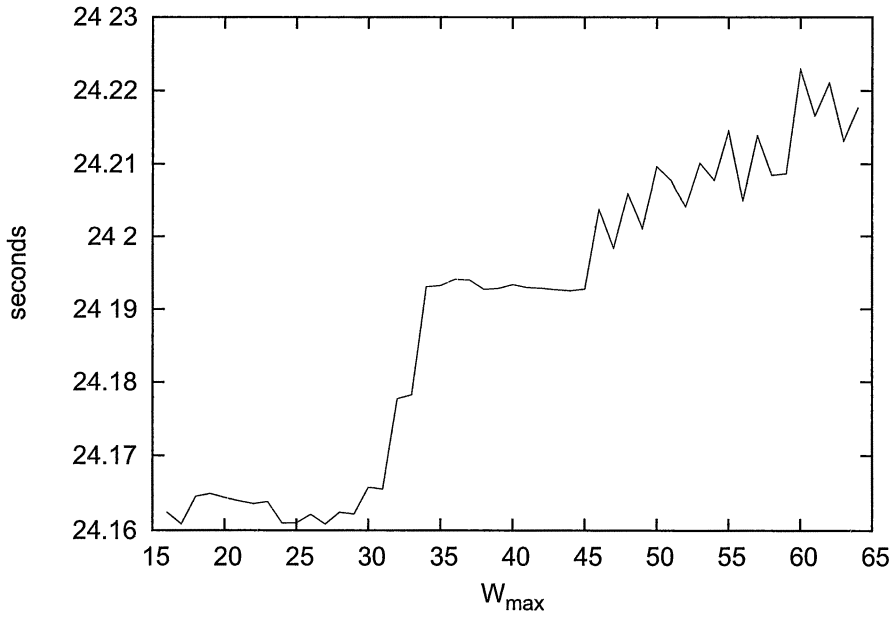

Figure 4.8: $thread_1$ time minus $thread_1$ wait time for synthetic benchmark on 16-megabyte pages.

expected that the distance between the two plots (which corresponds to the amount of time spent waiting) would be much greater for $thread_1$ than for $thread_0$. This is because, $thread_1$ encounters far fewer cache misses since it is capitalizing on the data cached by $thread_0$. Figures 4.6 and 4.8 have the total wall-clock time removed from the plot so that the y-axis scaling is increased for the wall-clock minus wait time data. These two figures again support the theoretical model, showing W_{max} near 31 to correspond to the optimal synchronization interval.

CHAPTER 5

CONCLUSIONS

Pipeline parallelism provides a locality conscious approach to parallelism that is appropriate for CMP platforms. When implementing pipeline parallelism, an important consideration is the synchronization interval. The optimal synchronization interval is as large as possible, so synchronization overhead is minimized, but small enough to allow for maximum reuse of in-cache data.

This research provides a model for predicting a profitable synchronization window for pipeline parallelism given the target platform's cache size and configuration as well as some properties of the workload to be parallelized (equation 3.9). The model performs as expected under test but only when the dataset is mapped into real address space in a contiguous block. This limitation is due to the fact that the model assumes the data will be spread evenly across all cache sets. Ideally, the dataset will be mapped into real address space on one large page. This type of mapping guarantees a contiguous mapping.

Tests performed with the ideal synchronization window size encountered a 5.5 percent lower overall miss rate than tests performed with a window size that was one iteration greater than ideal. The miss rate for the ideal window size was 15 percent lower when compared to tests run with a synchronization interval that was three iterations larger than ideal (see Figure 4.1).

CHAPTER 6

FUTURE WORK

More testing on a wide variety of platforms is necessary to verify the validity and generality of the model presented here. One limitation of the architecture under test in the results presented is that it does not support simultaneous multi-threading (SMT). On one hand, SMT may present additional complexities that will necessitate modification of the model. On the other hand, SMT may provide a way to capitalize on some of the time threads spend waiting at synchronization points. Also, the model presented here was developed and tested on a two core machine. More work is needed to determine how well this model scales and to generalize the model so that it can be applied to architectures with n-cores and m-levels of shared cache.

Further research should be performed with the ultimate goal of translating the model into a generalized set of heuristics that could be implemented in a compiler tool-chain. The appropriate place to make architecturally dependant optimizations, like the synchronization window optimization presented in this work, is at compile-time.

BIBLIOGRAPHY

[1] J. Huh, D. Burger, and S. Keckler, "Exploring the design space of future CMPs," *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 199–210, 2001.

[2] D. Greer, "Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *SIGPLAN Not.*, vol. 31, no. 9, pp. 2–11, 1996.

[4] L. A. Barroso, "The price of performance," *Queue*, vol. 3, no. 7, pp. 48–53, 2005.

[5] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *Micro, IEEE*, vol. 23, no. 2, pp. 22–28, 2003.

[6] J. Laudon, "Performance/watt: the new server focus," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 5–13, 2005.

[7] R. Sasanka, S. Adve, Y. Chen, and E. Debes, "The energy efficiency of CMP vs. SMT for multimedia workloads," *Proceedings of the 18th annual international conference on Supercomputing*, pp. 196–206, 2004.

[8] T. Mudge, "Power: a first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[9] J. Dongarra, D. Gannon, G. Fox, and K. Kenned, "The impact of multicore on computational science software," *CTWatch Quarterly*, vol. 3, pp. 3–10, 2007.

[10] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 505–522, 2005.

[11] M. Stan and W. Burleson, "Bus-invert coding for low-power I/O," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 3, no. 1, pp. 49–58, 1995.

[12] S. Wuytack, F. Catthoor, L. Nachtergaele, H. De Man, and L. IMEC, "Power exploration for data dominated video applications," *Low Power Electronics and Design, 1996., International Symposium on*, pp. 359–364, 1996.

[13] P. Gelsinger, P. Gargini, G. Parker, and A. Yu, "Microprocessors circa 2000," *Spectrum, IEEE*, vol. 26, no. 10, pp. 43–47, 1989.

[14] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *International Symposium on Microarchitecture*, 2007.

[15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel Tiled QR Factorization for Multicore Architectures," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4967, p. 639, 2008.

[16] K. Papadopoulos, K. Stavrou, and P. Trancoso, "Helpercore_db: Exploiting multicore technology for databases," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, (Washington, DC, USA), p. 420, IEEE Computer Society, 2007.

[17] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1768–1810, 1994.

[18] S. N. Vadlamani and S. F. Jenks, "The synchronized pipelined parallelism model," The 16th IASTED International Conference on Parallel and Distributed Computing and Systems, 2004.

[19] 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 *HPCA '07: Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA 2007)*, 2007.

[20] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads," in *SPAA* (P. B. Gibbons and M. Adler, eds.), pp. 235–244, ACM, 2004.

[21] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "Cachescouts: Fine-grain monitoring of shared caches in cmp platforms," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, (Washington, DC, USA), pp. 339–352, IEEE Computer Society, 2007.

[22] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, p. 176, IEEE Computer Society Washington, DC, USA, 2004.

[23] M. Kandemir, "Data locality enhancement for cmps," in *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, (Piscataway, NJ, USA), pp. 155–159, IEEE Press, 2007.

[24] X. Vera, J. Abella, J. Llosa, and A. González, "An accurate cost model for guiding data locality transformations," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, pp. 946–987, 2005.

[25] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 39, no. 4, pp. 442–459, 2004.

[26] F. Li and M. Kandemir, "Locality-conscious workload assignment for array-based computations in mpsoc architectures," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*, (New York, NY, USA), pp. 95–100, ACM, 2005.

[27] D. Nikolopoulos, "Dynamic tiling for effective use of shared caches on multithreaded processors," *International Journal of High Performance Computing and Networking*, vol. 2, no. 1, pp. 22–35, 2004.

[28] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 40–52, ACM, 1991.

[29] C. Ding and K. Kennedy, "Improving effective bandwidth through compiler enhancement of global cache reuse," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 10038b, 2001.

[30] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim: a memory system simulator," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 100–107, 2005.

[31] E. Daylight, D. Atienza, A. Vandecappelle, F. Catthoor, and J. Mendias, "Memory-access-aware data structure transformations for embedded software with dynamic data accesses," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, pp. 269–280, March 2004.

[32] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum, "Streamware: programming general-purpose multicore processors using streams," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 297–307, ACM, 2008.

[33] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using PAPI for hardware performance monitoring on Linux systems," in *Conference on Linux Clusters: The HPC Revolution*, 2001.

VITA

Michael Jason Cade was born in New Orleans, Louisiana, on March 25, 1972, the son of Lyn V. and Michael Allen Cade. After completing his work at Olathe South High School, Olathe Kansas, in 1990, he entered Kansas State University at Manhattan Kansas. He received a Bachelor of Fine Arts in Graphic Design from Kansas State University in 1995. From 1995 through 2005 he was employed as a commercial artist specializing in design for print media. In the summer of 2005 he entered the Graduate College of Texas State University-San Marcos to pursue a Master's degree in Computer Science.

Permanent Address: 906 Marlton
San Marcos, Texas 78666

This thesis was typed by Michael Jason Cade.