

# Balancing Locality and Parallelism on Shared-cache Multi-core Systems

Michael Jason Cade  
Texas State University  
San Marcos, TX  
mc1235@txstate.edu

Apan Qasem  
Texas State University  
San Marcos, TX  
apan@txstate.edu

**Abstract**—The emergence of multi-core systems opens new opportunities for thread-level parallelism and dramatically increases the performance potential of applications running on these systems. However, the state of the art in performance enhancing software is far from adequate in regards to the exploitation of hardware features on this complex new architecture. As a result, much of the performance capabilities of multi-core systems are yet to be realized. This research addresses one facet of this problem by exploring the relationship between data-locality and parallelism in the context of multi-core architectures where one or more levels of cache are shared among the different cores. A model is presented for determining a profitable synchronization interval for concurrent threads that interact in a producer-consumer relationship.

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, data reuse patterns within the workload, and the size and configuration of the last-level of cache that is shared among processing units. By considering these factors, the calculation of the optimum synchronization window incorporates parallelism and data locality issues for maximum performance.

**Index Terms**—shared-cache, parallelism, performance tuning, memory hierarchy optimization

## I. 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 increased design complexity 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 [8]. By focusing development toward the duplication of relatively small and simple design units on a single chip, profitable advances in performance can again be realized. At the same time reductions in power consumption and design overhead can be achieved [10]. By adequately capitalizing on opportunities for parallelism, a chip multiprocessor (CMP) with two cores can achieve the throughput of a single core processor that is running at nearly twice 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 [1], [14].

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 [6]. 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 [18]. When cache resources are shared among processors, data locality and parallelism become related. Consider the data-parallel execution model. Given a large data set and a task to be performed on the data, multiple processing units can be assigned to perform the task on subsets of the data in parallel. As parallelism is increased by the addition of more processing units there is more contention for 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 can be 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 [19], [23].

When considering approaches to parallelization, there are essentially three models, data, task and pipelined parallelism. In the past, pipelined parallelism has not received as much research attention as data or task parallel models. However,

due to several factors that make pipeline parallelism more relevant for CMPs, this model of parallelism is likely to play a significant role in parallelizing applications on multi-core systems. Unlike data parallel models, pipeline parallelism can effectively exploit locality within a shared cache since data is shared among the pipeline stages. Furthermore, 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, parallelization of workloads is essential to performance. 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 allows for parallelism- and locality-aware performance optimizations. The optimum synchronization window is a function of the number of threads, data reuse patterns within the workload, and the size and configuration of the last-level of cache that is shared among processing units.

## II. RELATED WORK

The related work is divided into three sections. First, work related to exploiting parallelism on CMPs is presented. Next research that pertains to optimizing locality on CMPs is explored. The third section addresses research where optimizations for locality and the exploitation of parallelism are considered together.

### A. Exploiting Parallelism on CMPs

The exploitation of parallelism on CMP architectures must become a key focus if the performance gains promised by this architecture 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, to realize the full potential of CMPs, software must adjust to a new parallel paradigm of computation [6]. Applications that take no measures to exploit the parallelism offered by CMPs may even see performance decrease on CMP platforms. 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 [17].

### B. Data Locality Optimizations for CMPs

The issue of machine balance is particularly 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 (64-bit words). 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 [2]. 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 or row major layouts found in FORTRAN and C [2].

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 [20]. 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.* assert that if multi-threaded applications tended to be data-independent, unique cache facilities for each core would be the most appropriate approach. However, if applications tend toward sharing some amount of data, sharing the last-level cache is a more appropriate approach. Jaleel reports a reduction in memory bandwidth demands by factors of 3 to 625 when the last-level cache is shared versus private last-level caches [11].

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 [16], [24]. 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 discovering and exploiting the opportunities for data locality within individual loop nests [12], [13], [21]. 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 [15].

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. 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 [5].

Since the amount of activity on the memory bus can be correlated to power consumption [22], 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 [4].

### C. 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 and data locality issues runs with significant power and performance benefits [15].

One approach that incorporates parallelism with consideration for data locality is the stream programming model. In contrast to the von Neumann model, stream programming uses a data-flow model. In the data-flow model, data is loaded into local memory as a bulk load, operations are performed in parallel on the loaded data and the results are stored as a bulk 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 [9].

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 [20]. SPPM applies to an important class of workloads where the problem can be seen as a sequence of interdependent stages. In other words, these workloads can be modeled 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 processing 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 [20].

### D. Limitations of the current body of literature

The current body of literature is somewhat sparse where data locality and parallelism 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. In shared cache systems, parallelism and data locality become inextricably intertwined. More research should be directed toward modelling the behaviors surrounding this relationship so that workloads may be optimized for these new CMP architectures.

## III. A MODEL FOR RELATING PARALLELISM AND DATA 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 its current value, such that the data at  $time_{n+1}$  is a function of the data at  $time_n$ .

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  $D_{n+1}$ .

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

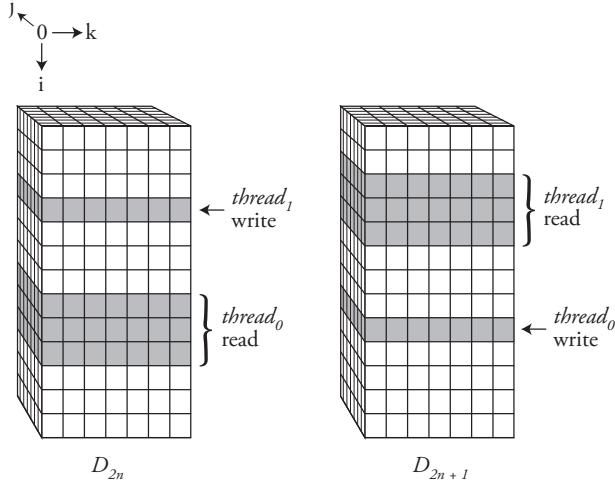


Fig. 1. Multiple threads may operate simultaneously on the data so long as they do not collide. Here,  $thread_0$  reads from  $D_{2n}$  and writes to  $D_{2n+1}$ . A second thread,  $thread_1$ , may be deployed that reads from  $D_{2n+1}$  and writes to  $D_{2n}$  so long as mechanisms are in place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

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 1).

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 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.

$$W_i = \frac{W_{max} - W_{min} - 1}{2} \quad (1)$$

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 this interval grows, synchronization overhead decreases.

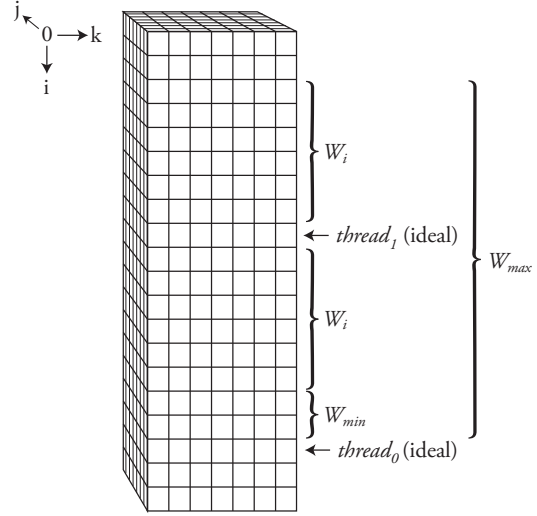


Fig. 2. Here,  $W_i$  is the synchronization granularity,  $W_{min}$  is the minimum window size and  $W_{max}$  is the maximum window size ( $2W_i + 1 + W_{min}$ ).

The optimal synchronization interval would maximize the exploitation of data locality while incurring the least amount of synchronization overhead. Therefore,  $W_i$ , and  $W_{max}$ , should be as large as possible. However, if the synchronization interval between  $thread_0$  and the last thread ( $thread_n$ ) is too large, later threads will compete with earlier threads for cache rather than being able to capitalize on data cached from previous accesses. For this reason, the distance, in terms of data, between  $thread_0$  and  $thread_n$  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  $thread_0$  consumes the newest data in cache. The fraction of the cache between  $thread_0$  and  $thread_n$  can be expressed by the following equation:

$$U = \frac{\alpha}{C} \quad (2)$$

where  $U$  is the amount of cache consumed by data that  $thread_0$  has accessed but that  $thread_n$  has not yet accessed,  $\alpha$  is the cost, in terms of cache, of updating the elements that  $thread_n$  has not yet updated 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:

$$\alpha = (T - 1) \times W_i \times \beta \times \gamma \quad (3)$$

where  $T$  is the total number of threads deployed to update the dataset,  $\beta$  is the number of elements in the dataset for each unit of  $W_i$  and  $\gamma$  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,  $\gamma$  must be considered in terms of cache lines and not bytes.

The  $\gamma$  term may be expressed in the following way:

$$\gamma = \frac{l}{S/d} \quad (4)$$

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. Dividing  $l$  by  $S/d$  accounts for the locality inherent in contiguous accesses to the same cache-line. Of course, this expression of the cache-cost term presupposes a linear and ordered access pattern across multiple element updates. If the access pattern were to randomly select elements from the dataset, the  $S/d$  term would be incorrect and  $\gamma$  would have to be accounted for in a different way.

Finally,  $U$  may be explicitly defined as:

$$U = (T - 1) \times \frac{W_i \times \beta \times l \times d}{S \times L} \quad (5)$$

where  $L$  is the total number of lines in cache.  $L$  replaces  $C$  in equation 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, the denominator of equation 5 reverts to  $C$ , as in (2).

For the synthetic benchmark described in this work,  $\beta$  is the product of the second and third dimensions of the dataset. Also, since the ultimate goal is to maximize  $U$  without exceeding the cache capacity, the target value for  $U$  is one. Therefore, based on equation 5, the following represents the optimal synchronization granularity for the synthetic benchmark.

$$W_i = \frac{C}{(T - 1) \times j \times k \times l \times d} \quad (6)$$

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 eight bytes. With 64-byte cache lines, and an update function that is dependant on all of an element's neighbors, the  $l$  term is nine (one cache line is accessed for each of the nine rows that make up the cube surrounding the element in question). Using the cache-use model in equation 6 with a four mega-byte last-level cache shared between two threads, the theoretical ideal synchronization interval is calculated to be 14.2 units on the

innermost dimension of the dataset.

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

$$W_i = 14.2 \quad (7)$$

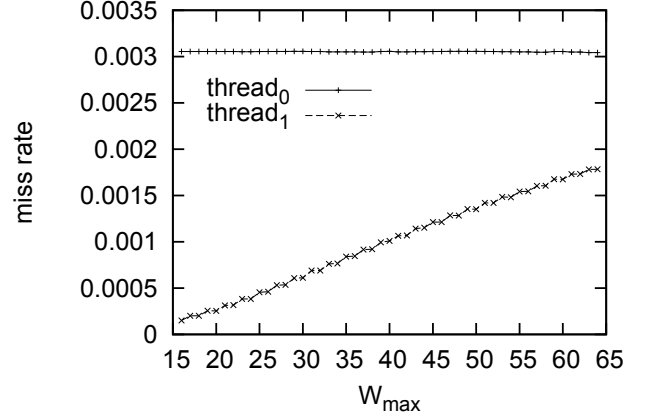


Fig. 3. Data miss rate for synthetic benchmark with four-kilobyte pages.

## IV. EXPERIMENTAL RESULTS

### A. Testing Environment

Experimental results were collected by running the synthetic benchmark on a 2.33GHz Intel<sup>®</sup> Core<sup>™</sup> 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 [7]. 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

TABLE I  
PERFORMANCE OF SEQUENTIAL VERSUS PARALLEL EXECUTION ON FOUR-KILOBYTE PAGES.

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

TABLE II  
PERFORMANCE OF 4KB VERSUS 16MB PAGES ( $W_{max} = 31$ ).

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

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 1 shows the relationship between  $W_{max}$  and  $W_i$ . To evaluate the model presented in equation 6, the benchmark is repeated over a range of maximum window sizes. The results presented are the averaged results of five test runs under the same testing conditions.

### B. 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 I). Although performance increased over sequential execution, the parallelized code did not exhibit the expected optima when manipulating the synchronization window sizes.

Intuitively, there should be an optimal synchronization window size, where the maximum amount of data locality is exploited while incurring the minimum amount of synchronization overhead. Based on equations 6 and 1, an optima should be at  $W_{max}$  of 31. However, in tests using four-kilobyte pages, the increase in miss rates is linear relative to the increase in  $W_{max}$  near the theoretical optimal synchronization window size (see Figure 3).

The wall-clock time results with four-kilobyte pages were similar. Experiments did not reveal a minima that 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 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, which in turn increases the overall miss rate.

In addition to utilizing cache inefficiently, spreading the dataset across multiple pages caused an increase in TLB misses (Table II). 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.

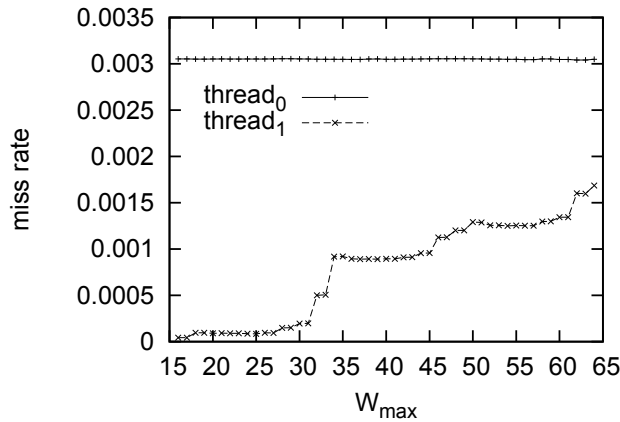


Fig. 4. Data miss rate for synthetic benchmark with 16-megabyte pages.

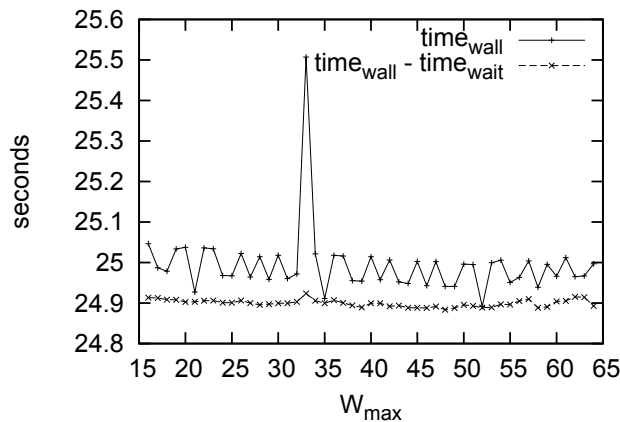


Fig. 5.  $thread_0$  time for synthetic benchmark on 16-megabyte pages.

### C. 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 the overall cache miss rate (Table II). It is worth noting that the miss rate for  $thread_0$  is the same for both page sizes. This is because 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<sup>TM</sup> 2 Duo has 256 entries, there would be a

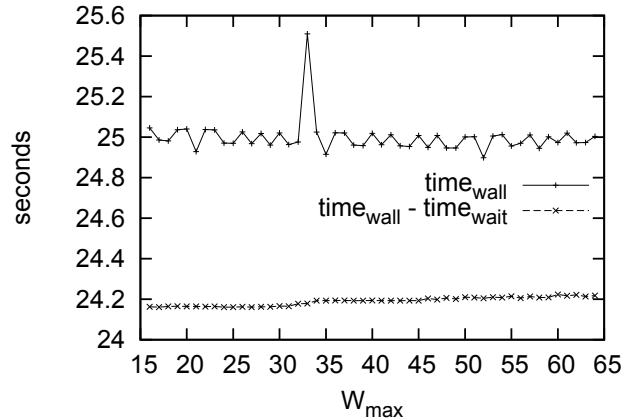


Fig. 6.  $thread_1$  time for synthetic benchmark on 16-megabyte pages.

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. With the dataset mapped into memory on a single 16-megabyte page, all of the data accesses for the 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 6, it is only with 16-megabyte pages that we see the model supported in the test results.

Figure 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 5 and 6 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 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  $thread_0$ .

## V. 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 6). 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 3).

## VI. 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.

Because the appropriate place to make architecturally dependant optimizations is at compile-time [3], further research should be performed with the ultimate goal of translating the model into a generalized set of heuristics. These rules could then be implemented in a compiler tool-chain. Since automatically identifying opportunities for parallelization is a difficult problem to solve, future efforts might also include the development of syntactical hints. These hints would be used in high-level code to indicate loops that are candidates for pipeline parallelization, as well as the basic structure of the parallelization that the compiler should produce.

## REFERENCES

- [1] L. A. Barroso, "The price of performance," *Queue*, vol. 3, no. 7, pp. 48–53, 2005.
- [2] 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.
- [3] 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.
- [4] E. Daylight, D. Aienza, 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, no. 3, pp. 269–280, March 2004.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] D. Greer, "Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.
- [9] J. Gummaraaju, 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: ACM, 2008, pp. 297–307.
- [10] 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.
- [11] 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.
- [12] 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: IEEE Press, 2007, pp. 155–159.
- [13] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 39, no. 4, pp. 442–459, 2004.
- [14] J. Laudon, "Performance/watt: the new server focus," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 5–13, 2005.
- [15] F. Li and M. Kandemir, "Locality-conscious workload assignment for array-based computations in mpoc architectures," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM, 2005, pp. 95–100.
- [16] 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*. IEEE Computer Society Washington, DC, USA, 2004, p. 176.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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: IEEE Computer Society, 2007, pp. 339–352.