Autotuning Strategies For Reducing Synchronization Costs In Multithreaded Kernels

Apan Qasem
Department of Computer Science
Texas State University, San Marcos, Texas, USA
{apan}@txstate.edu

ABSTRACT

Emergence of multicore architectures has opened up new opportunities for thread-level parallelism and dramatically increased the theoretical peak on current systems. However, achieving a high fraction of peak performance requires careful orchestration of many architecture-sensitive parameters, both on-chip and across the interconnect. In particular, the presence of shared-caches on multicore architectures makes it necessary to consider, in concert, issues related to thread synchronization and data locality. This paper studies the complex interaction among several compiler-level code transformations that affect data locality, achieved parallelism and synchronization and communication costs. We characterize this interaction using static analysis and generate a search space suitable for efficient automatic performance tuning. We also develop a heuristic based on number of threads; data reuse patterns, and the size and configuration of the shared cache, to estimate the optimal synchronization interval for pipeline-parallelized code. We validate our choice of tuning parameters and evaluate our heuristic with experiments on a set of scientific kernels on four different multicore platforms. The results show that our proposed heuristic is able to estimate the optimal synchronization window with reasonable accuracy and able to achieve significant performance improvement.

Keywords: Compiler Design, Parallelism, Software Infrastructure

1. INTRODUCTION

The last few years have witnessed the meteoric rise of multicore systems, from being a fledgling experimental platform to becoming the defining technology shaping the future of high-performance and network computing. Be that as it may, the promise of multicore cannot be realized through hardware alone. It is widely agreed that much of the responsibility of harnessing the true potential of multicore platforms lies with system software. In particular, we need to develop software-based strategies to discover hidden parallelism, automatically transform code for parallel execution and perhaps most importantly, minimize communication costs without paying a penalty in terms of unexploited inter-thread data locality. Exploiting data locality and reducing synchronization costs becomes particularly challenging on today’s architectures because of the presence of shared-caches at different levels of the memory hierarchy [62]. A shared-cache poses an inherent trade-off between data locality and parallelism. On one hand, any parallel decomposition of the application inevitably influences data reuse patterns in each concurrent thread. On the other hand, transformations for improving locality often impose constraints on how much parallelism can be extracted. For example, if a data-parallel decomposition of an application creates a working set of size, WS, for thread, ti, then to improve data locality, a compiler needs to ensure that

$$\sum_{i=0}^{k} WS_i < CS$$

where CS is the size of the shared cache and k is the number of threads. To satisfy this constraint, a compiler may consider several options including tiling each WS, finding a suitable schedule for each thread or reconfiguring the decomposition itself. Each approach will not only impact the data reuse of an individual thread but also the synchronization cost and task granularity. Thus, when parallelizing an application on current scalable architectures, it is critical to find the right balance between exploited data locality and synchronization and communication overhead. This entails considering a large number of transformation and architectural parameters. Since information about many hardware and runtime parameters is not readily available, achieving a high fraction of peak on modern architectures using static techniques has become an extremely challenging task.

In response to this daunting challenge, several research groups have proposed methods and techniques for automatic performance tuning [3,12,15,28,31,37,43,53]. In an autotuning framework, a code is analyzed, alternate code variants are generated with tuning parameters and then a heuristic search is performed based on execution time feedback to obtain an implementation that yields optimal or near-optimal performance for the target platform. Many of these tuning efforts have achieved significant success within specific domains. A major challenge that remains, however, is finding effective ways to cut down on the prohibitively long search times. It has been shown that analytical modeling and user guidance can help cut down the search space within manageable levels to make searches less expensive. However, modeling is sometimes difficult because of a lack understanding of the search space. In particular, the combined search spaces of locality and parallelism have not been studied to a great extent. The few autotuning efforts that have considered tuning for parallelism have limited themselves to single dimensional problem decomposition and have not considered the issue of data locality in concert [45]. This work aims to reduce this gap by providing a framework for constructing a representative search space for multithreaded programs.

In this paper, we take a systematic approach to characterizing and exploring the search space of transformation parameters that affect both data locality and communication costs in multithreaded applications. We focus on scientific kernels, since the issue of portable high-performance is often more pronounced for numerical code in scientific domains, where applications are
characterized by high degrees of temporal reuse and large working sets that often do not fit in the higher-level caches. We identify code transformations and transformation parameters that can be used to control granularity of parallelism, synchronization interval and the memory access patterns of concurrent threads. We establish a set of criteria to characterize the relationship of transformation parameters that affect both data locality and parallelism. Since multicore systems with shared-caches give rise to both intra and inter-core locality, we consider both types of reuse in constructing the initial search space. Additionally, we also incorporate the issue of false sharing within co-running threads. In terms of parallelism, we consider problem decomposition and thread creation at all levels of a given loop nest. We combine all of these parameters into one unified multi-dimensional search space. To explore the search space in a non-orthogonal manner, we employ several multi-dimensional search methods, including direct search [22,33] and simulated annealing [11,25]. We devote particular attention to pipelined parallelized code and develop a cost model that captures the interaction between data locality and parallelism and establishes a criterion for choosing a suitable synchronization interval. This model is constructed empirically through the use of a synthetic benchmark. 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. The main contributions of this paper include:

- Identification of key transformation parameters for optimizing parallel numerical code on networked multicore architectures
- An architecture-sensitive cost model for estimating optimal synchronization interval for pipeline parallelized code

The rest of the paper is organized as follows: in Section 2, we describe related work, in Section 3, we present the analysis for characterizing the search space; in Section 4 we describe the synthetic benchmark and the heuristic for estimating synchronization interval; in Section 5, we present experimental results and finally we conclude in Section 6.

2. LITERATURE REVIEW

2.1. Exploiting Parallelism and Data Locality

The dominance of multicore technology within the processor industry has lead to a plethora of work in code improvement techniques for this platform. Software-based approaches have been proposed to create new parallel abstractions, extract more parallelism, exploit data locality in the shared memory hierarchy, improve thread schedules, and control synchronization delays. In our treatment of related research, we limit the discussion to work most relevant to our approach, namely strategies for exploiting data locality and reducing communication overhead.

Many techniques for extracting parallelism and controlling granularity are described in the literature [8,24,32,38,47,50]. Recent work has focused on extracting fine-grained parallelism and exploring different models of parallelism such as pipelined parallelization. Buttari et al. present algorithms for Cholesky, LU and QR factorizations, where operations are represented as sequences of small tasks that can operate in parallel on square blocks of data [8]. They utilize the asynchronicity of short threads to hide the latency of memory accesses to improve performance. Papadopoulos et al. show that adding more execution threads is not beneficial and can be prohibitively difficult to implement for database applications [38].

Locality transformations described in this paper have been widely studied [9,10,14,17,18,57]. Loop blocking or tiling is the predominant transformation for exploiting temporal locality for numerical kernels [56,14]. The use of unroll-and-jam to improve register reuse is also common in both commercial and research compilers. Loop fusion and array contraction have been used in conjunction to improve cache behavior and reduce storage requirements [17]. Fraboulet et al. proposed a technique for loop alignment for improving memory performance [18]. A more sophisticated alignment strategy is used by Zhao and Kennedy as an enabling transformation with loop fusion and scalarization [63]. Loop Skewing and time skewing serve as enabling transformations in strip-mining and parallelizing loop with carried dependencies.

Although the literature is replete with heuristics for selecting tile sizes and choosing unroll factors [14,29,34,56], attempts at integrating all these transformations have been less common [55]. These approaches target single-core machines, and thus does not deal with the problem of exploiting parallelism.

Relatively few papers have addressed the issue of data locality and parallelism in concert. Among these, Vadlamani and Jenks [51] present the synchronized pipelined parallelism model for producer-consumer applications. Although their model attempts to exploit locality between producer and consumer threads, they do not provide a heuristic for choosing an appropriate synchronization interval (i.e., tile size). Krishnamoorthy et al. describe a strategy for automatic parallelization of stencil computations [30]. Their work addresses both parallelism and data locality issues and is similar to the work presented in this paper. However, Krishnamoorthy et al. uses static heuristics for selecting tile sizes and does not employ empirical search, as we do in this work.

2.2. Autotuning Single-threaded Applications

A number of successful empirical tuning systems provide efficient library implementations for important scientific domains, such as those for dense and sparse linear algebra [7,16,53], signal processing [19,40] and tensor contraction [5]. Among these, ATLAS [53], is the most widely used within the scientific community and has become the de facto standard for evaluating other autotuning systems. The success of automatically tuned domain specific libraries has sparked considerable interest in applying search-based methods for tuning general applications. Research that aims to autotune general applications fall into two categories: those that tackle the phase-ordering problem and aim to find the best sequence
of transformations [3,31,37,36,49] and those that concentrate on finding the best parameter values for transformations that use numerical parameters [12,15,28,43]. More recent efforts strive to combine the two methods to provide a more unified solution that involve compile-time tuning with source-to-source transformations and runtime tuning and optimization [4].

Earlier work in autotuning mainly focused on finding good search strategies or modifying existing ones to reduce tuning times. Genetic algorithms [3,31], stochastic hill climbers [3] and greedy constructive algorithms [3] have been used to explore the search space of optimizations sequences. In terms of the search space of numerical parameters, there has been work in applying direct search methods [43,44], simulated annealing [20,26,41,44], pyramid search [27,41], window search [26,41], binary search [12] and random search [26,41]. However, none of the search strategies proved particularly effective in the context of autotuning; in most cases, yielding at most a 5% improvement over random search [61,41]. The limited success of search algorithms lead to the development of model-based tuning where some form of analytical modeling or guidance is used to prune the search space, guide search heuristic or reduce time spent in program evaluation during tuning.

The issue of model-guided tuning has been approached from several different angles. Most notable among these is the use of compiler-based analytical models in limiting the search space [13,31,43,48,60]. Chen et al. showed the analytical models can significantly cut down the search space for a set of transformations including tiling, loop interchange and unroll-and-jam [12]. Qasem and Kennedy have used models for pruning the combined search space of loop fusion and tiling [43]. The Active Harmony project focuses on runtime optimizations and use analytical models to establish an ordering of transformations rather than reducing the size of the search space [21]. The OSE compiler uses static heuristics for generating a pruned search space for optimization sequences [48]. Kulkarni et al. use techniques such as detecting redundant sequences and identifying equivalent code to cut down the number of program evaluations [31]. Apart from compiler models, machine learning techniques have been applied to tune unroll factors [46] and also for selecting the best optimization set (without reordering) [2]. There has been some work in using statistical models to explore the search space of optimization parameters. Vuduc et al. establish an early stopping criterion to eliminate less promising search space regions on the fly [52]. Pinkers et al. use a statistical method based on orthogonal arrays to choose the optimal sequence of transformations [39].

3. CONSTRUCTING THE SEARCH SPACE

This section outlines the compiler analysis used to construct the representative search space for multithreaded numerical code. In lieu of an extended formal description of the framework, we describe the key aspects of the analysis through a sequence of examples. These examples characterize the main performance trade-offs in parallelizing and tiling memory intensive numerical code.

For this discussion, we consider a simple three-dimensional loop nest that is fully parallelizable and there is reuse of along all three dimensions. Although these assumptions are somewhat simplistic, this example captures the core computation pattern for many scientific codes and is a suitable tool for illustrating the complex interaction between blocking and parallelizing transformations. Fig. 1 depicts example execution patterns for this code.

![Figure 1: Execution patterns in 3D loop](image)

As we can see, the number of loops that are parallelized and the number of dimensions that are blocked can result in widely varying thread granularity and data access patterns for each thread. For example, parallelizing across i, j and k loops creates extremely fine-grained parallelism, where each thread updates only one value in the array. We achieve a high-degree of concurrency with this decomposition. However, this variant is unlikely to perform well on most systems because the thread creation time to task completion ratio is very high. Moreover, parallelizing the innermost loop is going to negatively impact spatial locality, which can lead to performance loss (as we discuss later in this section). This issue of extreme fine-granularity can be addressed by parallelizing a subset of the loops in the nest (e.g., “plane” sweep) and by blocking in one or more dimensions and then parallelizing the blocked loops (e.g., “beam” sweep). However, both these methods have potential drawbacks. Parallelizing a subset of loops might imply that available parallelism on the target platform is not fully exploited. On the other hand blocking a loop with an unfavorable block size may lead to poor locality in threads. We now discuss these trade-offs in terms of exploitable intra-core and inter-core locality.

3.1 Intra-core Data Locality and Granularity of Parallelism

Intra-core temporal locality occurs when a data value, touched by a thread running on core p, is reused by a thread running on the same core (core p). Exploiting intra-core locality is particularly important for numerical code that sweeps over data domains multiple times (e.g., time-step computations). To achieve efficient sequential execution, such code would typically be tiled in all three spatial dimensions (with the aid of loop skewing [58]) to exploit temporal locality across different time-steps. If we parallelize this code along the time dimension, the execution resembles the one shown on the left in Fig. 2, where the time dimension is broken up into four blocks.
Each of the blocks are executed concurrently as a separate thread and each thread sweeps a block of data multiple times. In this scenario, it is important to ensure that the working set of each thread is made small enough to fit in the cache. To enforce this, we can further subdivide the time blocks as shown on the right in Fig. 2.

By selecting a sufficiently small block size, we can ensure that the working set of each individual thread fits into the cache. However, as we observe, reducing the block size also causes a reduction in thread granularity. Thus, reducing the block size for improved intra-core locality may result in an unbalanced load for the entire application and also add to thread creation overhead. The optimal blocking factor that exploits intra-core locality and finds a suitable granularity depends on a number of factors including the number of cores, the cache size and associativity, the current system load and the input data set. We express these tunable parameters for blocking as a set as follows

$$B_{SPACE} = \{B_i | lb \leq B_i \leq ub \land B_i = f(P, T, C, I)\}$$

where $P$ is the number of processing cores, $T$ is the number of executable threads, $C$ is the cache configuration and $f$ is input data set. $lb$ and $ub$ refer to the range of valid values for the blocking factor, which also may be stipulated using the parameters specified in the constraint of the set membership. The cardinality of $B_{SPACE}$ depends, among other factors, on the number of spatial dimensions in each loop nest.

### 3.2 Inter-core Data Locality and Thread Schedule

As mentioned earlier, the presence of shared-caches on multicore processors makes it imperative that we consider inter-core locality, which occurs when a data value, touched by a thread running on core $p$, is used by another thread running on core $q$, where $p \neq q$. Consider the “plane” sweep execution pattern depicted in Fig. 3.

In this scenario, each co-running thread is sweeping over three planes of the data set (due to reuse in the outermost dimension). We also observe, that for two consecutive threads, $thread_i$ and $thread_{i+1}$, two of the three planes are overlapping, which results in a combined cache footprint of just four planes. Therefore, if $thread_i$ and $thread_{i+1}$ execute on two cores that share a cache, much of the inter-core data locality will be exploited. To enforce this, again, we need to consider task granularity. By changing the block sizes we can control the amount of overlapped data between co-running threads. However, to ensure that $thread_i$ and $thread_{i+1}$ do indeed execute on the same chip, we need to adjust the scheduling heuristics. Hence, it is important to include scheduling and affinity parameter when tuning for both locality and parallelism.

### 3.3 False sharing

Finally, we consider the effects of false sharing on multithreaded code. False sharing has been studied extensively in the context of shared-memory multiprocessor systems and most of the ideas apply to multicore architectures as well. Consider the code fragment in Fig. 4.

This simple loop nest scales values in the two-dimensional array $A$ by values in $B$ and a factor of $k$ and initializes the one-dimensional arrays $C$, $D$ and $E$. We notice that the outer loop carries no dependencies and therefore can be safely parallelized. The OpenMP directives to run the outer loop in parallel are also shown in Fig. 4. Now, although there are no carried dependencies in the outer loop, we observe spatial locality across different iterations of this loop for references to $C(i)$, $D(i)$ and $E(i)$. If we assume we have 16 words per cache line then after every store to an element in the $C$, $D$ and $E$ arrays, the next 15 iterations of the outer loop will be touching the same cache line. Thus, for the sequential version this loop nest exhibits a high degree of spatial locality, easily exploited at all cache levels. However, the situation becomes problematic when we execute the loop in parallel. In the parallel version, concurrent threads will attempt to write values to a shared cache line. These writes will have to be serialized. Depending on how the architecture handles simultaneous write requests (e.g., whether it uses store buffers or queues), it may be necessary that the cache line be invalidated and reloaded from memory before the second write is allowed. This can lead to cache misses and increased bandwidth demands and force running threads to wait for the value to be brought back into cache. All these factors can contribute to significant performance loss. The inclusion of thread affinity and scheduling in the tuning search space addresses this issue.
4. A HEURISTIC FOR ESTIMATING AN OPTIMAL SYNCHRONIZATION INTERVAL

In this section, we describe a cost model for estimating an optimal synchronization interval for pipelined parallelized applications. Although the model can be applied statically, our goal is to use it to generate a pruned search space for tuning. We refer to this model as OSW (optimal synchronization window). OSW 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 OSW 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 $time_n$. Since the data in position $x$ at $time_{n+1}$ is dependent on the data at position $x$ and all of $x$’s neighbors at $time_n$, the data at $x$ cannot be updated to its $t$ $time_{n+1}$ 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 is completed and the data is in its final state, $D_{new}$. 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. At some point, thread1 would be far enough along to allow thread2 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, thread0 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. In order to preserve data dependencies, the threads must be prevented from colliding. If a thread that is making an update for $time_{n+1}$ gets too close, in terms of data, to the thread that is making the update for $time_n$, the data necessary for the $time_n$ update will be overwritten by the update for $time_{n+1}$ and the final result will be corrupted.

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 as shown in Fig. 5.

![Synchronization window](image)

Figure 5: Synchronization window

Synchronization for the 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. The execution window allows for the concept of synchronization granularity. The synchronization granularity is the number of iterations that either thread may safely execute and still maintain the execution window constraints. If thread1 and thread1 are the ideal distance from one another, then either thread may execute no more than $\frac{1}{2} (W_{max} - W_{min} - 1)$ iterations on the innermost index of the dataset without regard for the other thread’s progress. The synchronization granularity, $W_s$, is expressed in the following equation

$$W_s = \frac{W_{max} - W_{min} - 1}{2}$$

(1)

After every $W_i$ 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 thread0 and the last thread (threadn) is too large, threads coming after thread0 will compete with thread0 for cache rather than being able to capitalize on the data that is already in cache from thread0’s previous accesses. For this reason, the distance between thread0 and threadn 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 thread0 and thread n fits within cache. In other words, the ideal situation has thread0 accessing the oldest data in cache while thread0 accesses the newest data in cache. The fraction of the cache between thread0 and threadn can be expressed as $E = \alpha / CS$, where $E$ is the amount of cache utilized (effective capacity) by data that threadn has accessed but that thread0 has not yet accessed. $\alpha$ is the cost, in terms of cache, of updating the elements that thread0 has not yet updated in this pass but thread0 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

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

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 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 as in the following way $\gamma = 1 / (S / d)$, 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, $E$ may be explicitly defined as

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

where $L$ is the total number of lines in cache. $L$ replaces $C$ 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 this to

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

Replacing $\beta$ with the second and third dimensions of the dataset can further refine OSW. For this model, $E$ is defined by the equation below

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

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

$W_i = E \times C / ((T - 1) \times j \times k \times l \times d)$

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

$W_i = C / ((T - 1) \times j \times k \times l \times d)$

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 $W_i$ is calculated to be $2^{22} / (2 - 1) \times 2^6 \times 2^6 \times 9 \times 2^1 = 14.2$ units on the innermost dimension of the dataset.

5. EXPERIMENTAL RESULTS

We conducted extensive experimentation on a number of different platforms. This section summarizes our findings.

5.1 Experimental Setup

5.1.1 Tuning Framework

Fig. 6 provides an overview of our tuning framework.

![Figure 6: Overview of tuning framework](image)

The major components of the framework include a source-to-source transformer (LoopTool) [42], a set of performance measurement tools (HPCToolkit) [1], an analytic modeler and a search engine (PSEAT) that uses the measurements to guide selection of program transformations [59]. At the onset of the tuning process, the analytic modeler analyzes the source with architectural information and generates a pruned search space specification. The search module reads in the search specification and generates a set of transformation parameters that are applied to the input program by LoopTool. The program is then compiled using the native compiler and run on the target machine. During program execution, performance measurement tools collect a variety of performance measurements to feed to the search module. The search module uses these metrics in combination with results from previous passes to generate the next set of tuning parameters. This process continues until some pre-specified optimization time limit is reached or the search algorithm converges to a local minima.

PSEAT implements a number of search strategies.
including genetic algorithm, direct search [22,33], window search, taboo search, simulated annealing [25,11] and random search.

5.1.2 Platforms

We present experimental results on four Intel and AMD based multicore systems. Table 1 shows hardware and software configurations of each system. The three platforms show variations in number of available cores, processor speed, and cache capacity and organization. Thus, these platforms serve as a good basis for evaluating our tuning strategy for portable performance. GCC 4.3.2 with the -O2 flag was used to compile variants on each platform. In the rest of the section, the platforms are referred to using the names listed in Table 1.

<table>
<thead>
<tr>
<th>Core2</th>
<th>Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>2</td>
</tr>
<tr>
<td>Processor</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td></td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>L1</td>
<td>16 KB, 4-way</td>
</tr>
<tr>
<td></td>
<td>32 KB, 4-way</td>
</tr>
<tr>
<td>L2</td>
<td>4 MB, 4-way</td>
</tr>
<tr>
<td></td>
<td>2x4 MB, 4-</td>
</tr>
<tr>
<td>Sharing</td>
<td>L2, per core</td>
</tr>
<tr>
<td></td>
<td>L2, per socket</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.3.2</td>
</tr>
<tr>
<td></td>
<td>GCC 4.3.2</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
</tr>
</tbody>
</table>

Table 1: Experimental Platforms

5.1.3 Benchmarks

For this study, we selected seven multithreaded kernels from different domains in science and engineering. Table 2 provides a brief description and the source of each kernel. The kernels are important in their respective domains and have wide applicability. Each benchmark contains at least one loop that can be parallelized and exhibit a high-degree of temporal locality. The chosen kernels exhibit many performance issues, often achieving only a small fraction of the peak performance, making them good candidates for autotuning.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>advect3d</td>
<td>3D advection</td>
<td>NCOMMAS[54]</td>
</tr>
<tr>
<td>knapsack</td>
<td>integral</td>
<td>Morales[35]</td>
</tr>
<tr>
<td>stencil</td>
<td>heat equation</td>
<td>Parlab, UCB[23]</td>
</tr>
<tr>
<td>mgrid</td>
<td>multi-grid solver</td>
<td>SPEC FP 2000</td>
</tr>
<tr>
<td>dedup</td>
<td>pipelined</td>
<td>PARSEC [6]</td>
</tr>
</tbody>
</table>

Table 2: Benchmarks

5.2 Validating the Search Space

To determine the quality of the search space constructed in Section 3, we conduct a set of experiments where we examine the kernels’ performance sensitivity to variations in the selected tuning parameters.

5.2.1 Block Size and Performance

We first examine the blocking search space. Although we examined the six of the seven kernels listed in Table 2, we choose mgrid as a case study here. A blocking search space is divided into two levels: loop selection and block sizes. The loop selection level refers to which loops are selected for blocking (and parallelization). For example, in a two-dimensional nest, we may choose to block just the inner loop, just the outer loop, both the inner and outer loop, or none of the loops. These choices can be represented with a bit string of size two and represents a search space of size 22. For each loop selection level, there is a multi-dimensional search space that consists of all the valid block sizes for each loop that is blocked. For mgrid, we explore four different blocking options for the loop nests appearing in resid and psinv routines. For each blocking selection we explore a range of block sizes, starting from four and going up to 64 in steps of four. This gives rise to a two-level 15 dimensional search space with about 1615 points.

Fig. 7 shows how selection of blocking loops and choice of blocking factors impact the execution time and communication time of mgrid. The numbers presented are from experimental runs on Quad and execution times are normalized based on the time for the sequential version. As we can see, there is significant variation in execution time as blocking factors are varied for different parallelization configurations. Overall, parallelizing the middle loop provides the best performance, cutting the execution time by almost 80%. The communication overhead for these schemes also appears to be the lowest. Although, parallelizing the middle loop does appear to be most profitable on average, it is difficult predict which blocking size will work best. For example, when blocking the middle, blocking factors between 8 and 16 produce significantly worse results than any blocking size used when two of the loops are blocked and parallelized. Hence, to apply 1D blocking we need a way to weed out these bad points in the search space. Furthermore, we observe that for the 3D blocking, the overhead of the communication cost is lowest, although the overall execution time is still relatively high. This implies, that a strategy that is based solely on minimizing communication overhead is unlikely to succeed. The results for mgrid (and the other benchmarks) indicate that the blocking parameters selected for tuning cause enough variations in the cache miss rates to have a significant influence on multithreaded program performance. Thus, these results reiterate the need for tuning to find the optimal block size, even for parallel code.
5.2.2 Performance Impact of Thread Granularity

Similar to blocking, selecting loops for parallelization can have a significant impact on performance. Here, we evaluate performance sensitivity of mgrid and stencil on the parallelization tuning parameters derived in Section 3. We explore several different parallelization options in terms of number of loops selected for parallelization, number of threads used and the affinity of each thread. Among the choices for parallelizing a loop nest, we observed that nested parallelism always resulted in huge performance loss due to thread synchronization overhead. This may be due to deficiencies in the GCC OpenMP library. Nevertheless, we discard these options in our study. Thus, parallelization search spaces for have significantly fewer dimensions then

Figure 7: Block size and parallelism granularity impact on execution time and communication time

Figure 8: mgrid performance sensitivity to parallelization

Performance sensitivity to different parallelization strategies for stencil is depicted in Fig. 9. Overall, the selection of parallel loops tends to have less of an impact on stencil than mgrid. However, even in this case, there is no single configuration that works well for all four platforms. Parallelizing the outermost loop appears to be the best choice, except in the case of Athlon, where P1 variants perform better. Using four threads, whereas for the other two platforms P1 appears to be the best choice. No matter how many threads are used, the quad-core machines yield higher speedups in almost all cases. Thread affinity does appear to have an impact on performance but it is only significant on Athlon.
Parameters warrant inclusion in the search space.

Experimental results reported here were collected on a Core2 environment using a synthetic benchmark. The optimal synchronization interval in a controlled experiment is 

5.3.1 Validation on a Synthetic Benchmark

We first validate our model for estimating the optimal synchronization interval in a controlled environment using a synthetic benchmark. The experimental results reported here were collected on Core2. The synthetic benchmark performs a simple stencil computation and closely resembles the model of computation described in Section 4. A dataset of substantial size is necessary to smooth the data collected and to emphasize differences in measurements under different testing conditions. Iterating over the dataset a large number of times can also support these goals. 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. Because a contiguous allocation of memory is assumed by the OSW model, we run these experiments using large pages of size 16 MB. This creates a contiguous allocation for the chosen data set.

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 $\text{gettimeofday}$, 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 $\text{gettimeofday}$, one immediately before entering a barrier and another immediately after exiting a barrier. Synchronization counts are collected with counters in the benchmark code itself.

Multi-threaded test cases involve two parallel threads of execution. Each thread is bound to a unique core using calls to $\text{pthread_setaffinity_np}$ 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, which is two in the case of the synthetics benchmark. Synchronization granularity, or $W_i$, is directly related to the maximum window size; increasing $W_{\text{max}}$ increases the synchronization granularity. Equation 1 shows the relationship between $W_{\text{max}}$ and $W_i$. To evaluate the model presented in equation 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.

Fig. 11 shows L2 data cache miss rates for varying window sizes. Clearly, the synchronization interval has an impact on the miss rates for thread1. Miss rates thread0 remains more or less constant, this is due to the fact that the miss encountered by thread0 are largely compulsory misses. Figs. 12 and 10 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 thread1 than for thread0. This is because, thread1 encounters far fewer cache misses since it is capitalizing on the data cached by thread0. The OSW model aims to predict the optimal synchronization window size, where the maximum amount of data locality is exploited while incurring the minimum amount of synchronization overhead. Based on equations 9 and 1, that optimal point predicted for the synthetic benchmark is on Core2 is 31. From Fig. 10, we observe that although this predicted value does not correspond to the optimal running time, it comes very close. Since we have constructed the model to be conservative, it picks a value, where the first spike in L2 miss rates is observed.

![Figure 9: stencil performance sensitivity](image_url)

These results indicate that choice of parallelization parameters can cause significant variations on performance based on the architecture. Thus, these parameters warrant inclusion in the search space.

5.3 Accuracy of the Synchronization interval Estimation Model

5.3.1 Validation on a Synthetic Benchmark

We first validate our model for estimating the optimal synchronization interval in a controlled environment using a synthetic benchmark. The experimental results reported here were collected on Core2. The synthetic benchmark performs a simple stencil computation and closely resembles the model of computation described in Section 4. A dataset of substantial size is necessary to smooth the data collected and to emphasize differences in measurements under different testing conditions. Iterating over the dataset a large number of times can also support these goals. 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. Because a contiguous allocation of memory is assumed by the OSW model, we run these experiments using large pages of size 16 MB. This creates a contiguous allocation for the chosen data set.

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 $\text{gettimeofday}$, 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 $\text{gettimeofday}$, one immediately before entering a barrier and another immediately after exiting a barrier. Synchronization counts are collected with counters in the benchmark code itself.

Multi-threaded test cases involve two parallel threads of execution. Each thread is bound to a unique core using calls to $\text{pthread_setaffinity_np}$ 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, which is two in the case of the synthetics benchmark. Synchronization granularity, or $W_i$, is directly related to the maximum window size; increasing $W_{\text{max}}$ increases the synchronization granularity. Equation 1 shows the relationship between $W_{\text{max}}$ and $W_i$. To evaluate the model presented in equation 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.

Fig. 11 shows L2 data cache miss rates for varying window sizes. Clearly, the synchronization interval has an impact on the miss rates for thread1. Miss rates thread0 remains more or less constant, this is due to the fact that the miss encountered by thread0 are largely compulsory misses. Figs. 12 and 10 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 thread1 than for thread0. This is because, thread1 encounters far fewer cache misses since it is capitalizing on the data cached by thread0. The OSW model aims to predict the optimal synchronization window size, where the maximum amount of data locality is exploited while incurring the minimum amount of synchronization overhead. Based on equations 9 and 1, that optimal point predicted for the synthetic benchmark is on Core2 is 31. From Fig. 10, we observe that although this predicted value does not correspond to the optimal running time, it comes very close. Since we have constructed the model to be conservative, it picks a value, where the first spike in L2 miss rates is observed.

![Figure 10: thread, time for synthetic benchmark](image_url)
5.3.2 Performance Across Architectures

To evaluate the effectiveness of the OSW model across different architectures, we apply the model to advect3d, knapsack, canneal, dedup and ferret and run experiments on all four platforms listed in Table 1. We choose these kernels because they can be parallelized in a pipelined fashion. The platforms provide good variation in terms of number of thread contexts, cache capacity and sharing. In spite of the variations, we found in our experiments that the dominant architectural parameter in this case is the number of cores. Performance patterns were mostly similar for the two dual-core platforms and the two quad-core machines. Thus, for the sake of brevity, we closely examine performance results on Athlon and Quad and summarize the results for all platforms at the end of this section.

Fig 13 shows performance of advect3d as a function of the synchronization interval (window size). L1 and L2 cache miss rates broken down by producer and consumer threads are shown in Figs 14 and 15, respectively.

Execution time on both Athlon and Quad appear sensitive to changes in the windows size. However, the cache miss rates show much more variation on Quad than Athlon. The L1 miss rates on Quad appear almost erratic. On Quad, there is a definite downward trend for the execution time as window sizes are increased. This is caused by the reduction in both L1 and L2 miss rates. The cache miss rates do not seem to affect the execution time on Athlon. Hence, the performance changes are being caused mostly by the changes in wait time of the consumer threads.

The performance patterns for pipelined knapsack on Athlon and Quad are shown in Figs. 16-18. knapsack exhibits less sensitivity to the window size over the long run. Most of the changes occur for very small window sizes, between 16 and 32. These variations, however, are massive. The execution time for a window size was 10 times longer than the execution time when the window size is 64. The large increase in execution time is attributed to the high synchronization to computation ratio of smaller window sizes. Although the working sets for knapsack comfortably fit in the caches on Quad, we observe some significant changes in both the L1 and L2 miss rates. We were unable to determine the factors that lead to these fluctuations in the miss rates. In spite of the different performance patterns, the OSW model did reasonably well in selecting the optimal synchronization window.
Overall, OSW fares somewhat better on the dual-core machines, where the model is able to get to within 90% on average for all kernels. Performance on the quad-core machines is not as good, providing accuracy in the 80-85% range. One of the reasons why OSW was not always able to pick the best value is because of its conservative nature. Generally, OSW will stop at the first indication of an upward spike in the cache miss rates. For example, for advect3d on Athlon, OSW picks a window size of 4, when clearly there are better performance points for larger window sizes. In terms of the kernels, the model worked best on dedup and ferret, delivering close to 100% accuracy on both Core2 and Athlon. OSW proved least effective for canneal. This particular kernel was explicitly hand-optimized by the authors with a very aggressive synchronization strategy [6]. This explains, to some extent, why OSW for performed poorly on this particular kernel.

6. CONCLUSIONS

In this paper, we presented a strategy for exploring the search space of parallelism and data locality on current multicore architectures. Our strategy uses static compiler analysis to derive and prune a representative search space. The experimental results suggest the parameters chosen by our model do indeed deserve their place in the autotuning search space. This research also provides a model for predicting a profitable synchronization window for pipeline parallelism given the target platforms cache size and configuration as well as some properties of the workload to be parallelized. Although not achieving the optimal target, the model performs reasonably well in estimating profitable synchronization intervals and reducing communication costs in multithreaded code. Our experiments also exposed several scenarios where our cost models were not able to construct a good search space. These results suggest that not all key tunable parameters are being captured by our models. In particular, incorporating different thread scheduling heuristics for tuning would be a worthwhile endeavor.

REFERENCES


