Exposing Tunable Parameters in Multi-threaded Numerical Code*

Apan Qasem[1] Jichi Guo[2] Faizur Rahman[2] Qing Yi[2]

[1] Texas State University
[2] University of Texas at San Antonio

Abstract. Achieving high performance on today's architectures requires careful orchestration of many optimization parameters. In particular, the presence of shared-caches on multicore architectures makes it necessary to consider, in concert, issues related to both parallelism and data locality. This paper presents a systematic and extensive exploration of the combined search space of transformation parameters that affect both parallelism and data locality in multi-threaded numerical applications. We characterize the nature of the complex interaction between blocking, problem decomposition and selection of loops for parallelism. We identify key parameters for tuning and provide an automatic mechanism for exposing these parameters to a search tool. A series of experiments on two scientific benchmarks illustrates the *non-orthogonality* of the transformation search space and reiterates the need for integrated transformation heuristics for achieving high-performance on current multicore architectures.

1 Introduction

The emergence of chip multiprocessor systems has greatly increased the performance potential of modern computer systems. However, much of the responsibility in exploiting the on-chip parallelism lies with system software like the compiler and the operating system. The complexity of modern architectures implies that compilers, in addition to analyzing the code for profitability, need to account for a large number of architectural parameters to achieve a high fraction of peak performance. Since information about many of these hardware parameters is not readily available, achieving high performance on modern architectures is an extremely challenging task for a compiler.

The problem of achieving portable high-performance is often more pronounced for numerical code in scientific domains. Scientific applications are characterized by high degrees of temporal reuse and large working sets that often do not fit in the higher-level caches. These codes also involve significant amount of floating-point computation and contain loop nests that are amenable to parallelization at one or more levels. Thus, to achieve high-performance for numerical code, the

* This research is funded by the National Science Foundation under Grant No. 0833203 and No. 0747357 and by the Department of Energy under Grant No. DE-SC001770

compiler needs to find ways to extract parallelism *and* apply optimizations that exploit data reuse. This task of exploiting locality and parallelism is further complicated by the presence of shared-caches on current multicore platforms [24]. 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 access 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_i for thread t_i, then to improve data locality, the compiler needs to ensure that $\Sigma_{i=0}^{k} WS_i < CS$, where CS is the size of the shared cache. To satisfy this constraint, the compiler may consider several options including tiling each WS_i, finding a suitable schedule for $t_0 - t_k$ 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 for multicore-core architectures, it is important to find the right balance between data locality and parallelism, which involves considering a large number of code transformations and parameters. This combinatorial explosion of performance influencing parameters has made static compiler heuristics largely ineffective in delivering portable performance on current architectures.

In response to this daunting challenge, several research groups have proposed methods and techniques for automatic performance tuning [3, 6, 13, 17]. 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 reasonable success and have reduced the need for manual tuning of code in certain domains [17, 9]. However, one area where autoutning research has lagged is in considering a search space that covers both parallelism and data locality parameters for current multicore architectures. 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 [14]. There is one notable work that considers both parallelism granularity and data locality in the search space of stencil computations [6]. However, the exploration of the search space is done in an *orthogonal* manner and does not consider the interaction between search dimensions.

This paper takes a systematic approach to characterize and explore the search space of transformation parameters that affect both data locality and parallelism in multi-threaded applications. We observe that the interaction between locality and parallelism can be captured by considering both the *shape* and *size* of problem decomposition. The *shape* of a problem decomposition can be expressed as a combination of multiple blocking factors used in the code. This is the key insight that has driven this research. In this work, we identify code transformations and transformation parameters that can be used to control both the granular-

ity of parallelism 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 withing 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. We use a transformation scripting language [21] to implement each optimization and expose the parameters to a search engine. To explore the search space in a non-orthogonal manner, we employ several multi-dimensional search methods, including direct search and simulated annealing. Analysis of our experimental results suggest that the shape of a problem decomposition does indeed have significant impact on performance of numerical kernels on multicore architectures. The main contributions of this paper include :

- identification of key transformation parameters for optimizing parallel numerical code on multicore architectures
- an automatic method of exposing these parameters for tuning
- a non-orthogonal exploration of a search space that includes parameters for exploiting both parallelism and data locality

2 Related Work

2.1 Exploiting Parallelism and Data-locality on CMPs

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 parallelism and data locality.

Many techniques for extracting parallelism and controlling granularity is described in the literature [2]. Recent work has focused on extracting fine-grained parallelism and exploring different models of parallelism such as pipelined parallelization. Thies *et al.* [15] describe a method for exploiting coarse-grain pipelined parallelism in C programs. They also develop a set of parallel programming primitives to support pipeline parallelism. 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 [4]. 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. [12].

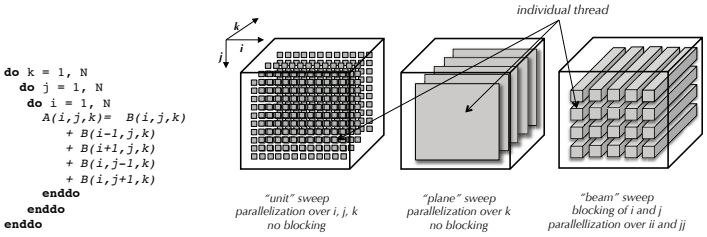
Locality transformations described in this paper have been widely studied [2]. Loop blocking or tiling is the predominant transformation for exploiting temporal locality for numerical kernels [19, 5]. The use of unroll-and-jam to improve register reuse is also common in both commercial and research compilers. Loop fusion and array contraction have been used in conjunction to improve cache behavior and reduce storage requirements [7]. Loop alignment has been used as an enabling transformation with loop fusion and scalarization [2]. 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 [19, 5], attempts at integrating all these transformations have been less common [18]. 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 [16] 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.* [11] describe a strategy for automatic parallelization of stencil computations. Their work addresses both parallelism and data locality issues and is similar to the work presented in this paper. However, Krisnamoorthy *et al.* uses static heuristics for selecting tile sizes and does not employ empirical search, as we do in this work.

2.2 Autotuning Multi-threaded Applications

Since the autotuning effort started prior to the multi-core era, much of the earlier work focused on tuning for single-core machines. Among these, several autotuned libraries for specific scientific domains have been quite successful. ATLAS which provides highly tuned linear algebra routines is widely used within the scientific community and has become the *de facto* standard for evaluating other autotuning systems [17]. 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] and those that concentrate on finding the best parameter values for transformations that use numerical parameters [6, 13]. Some of the autotuning work has transitioned into multicore architectures and have made great strides in improving performance on these systems. Autotuned libraries for FFT on multicore has already been proposed [8]. Autotuning techniques for multicore processors have also been applied to stencil computations and other numerical code within the scientific domain. Hall *et al.* developed the Chill framework that can tune numerical code on a wide range of memory transformation parameters [10]. Datta *et al.* propose a framework that can tune stencil code for both parallelism and locality on current multicore systems [6]. The work by Datta *et al.* comes closest to the work presented in this paper. The key difference between their approach and ours is that our framework ex-



(a) example numerical kernel

(b) thread execution patterns

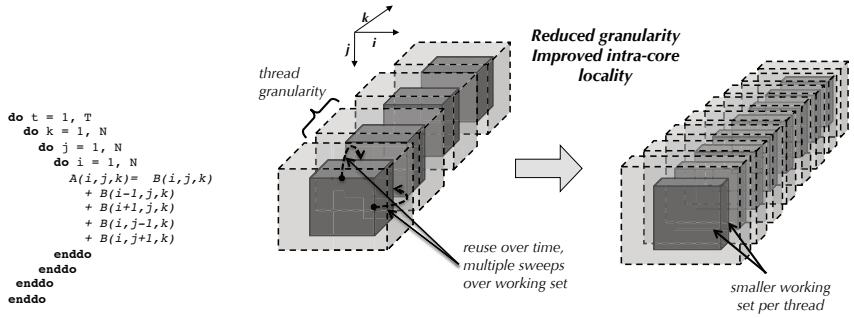
Fig. 1. Example execution patterns of parallelized and blocked numerical code

plores a multi-dimensional search space, whereas their framework performs an orthogonal search, looking at one dimension at a time.

3 Characterizing Performance Trade-offs

In this section, we characterize performance trade-offs in parallelizing and blocking memory intensive numerical code. For this discussion, we will consider a simple three dimensional loop nest, as presented in Fig. 1(a). We assume the loop is fully parallelizable and there is reuse of data along all three dimensions (i.e., the only carried dependence in the loop nest are input dependencies). Although these assumptions are somewhat simplistic, this example captures the core computation pattern for many scientific code and is a suitable tool for illustrating the complex interaction between blocking and parallelizing transformations.

Fig. 1(b) depicts example execution patterns for the code in Fig. 1(a). 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 completed* 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.



(a) time-step kernel (b) thread execution patterns

Fig. 2. Exploiting intra-core locality by reducing task granularity

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 either by a thread running on the same core (i.e., core p). Exploiting intra-core locality is particularly important for numerical code that sweeps over data domains multiple times (e.g., time-step computations).

An example code is shown in Fig. 2(a). To achieve efficient sequential execution, this code would typically be tiled in all three spatial dimensions (with the aid of loop skewing [20]) 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(b), 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(b). 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 host of factors including the number of cores, the cache size and associativity, the current system load and the input data set. Thus, finding a suitable block size for intra-core locality and parallelism is best achieved through autotuning, as we demonstrate later in this paper.

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 for the code in Fig. 1(a). In this scenario, each co-running thread is sweeping over three planes of the data set (due to reuse in the k 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 as a parameter when tuning for both locality and parallelism.

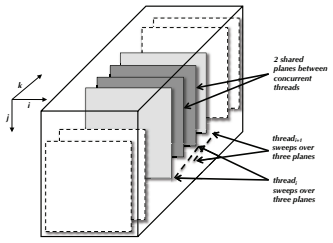


Fig. 3. Data sharing among concurrent threads

4 Tuning Framework

Fig. 4 gives an overview of our tuning framework, which includes two key components: a transformation engine based on the POET language [21] for generating alternate code variants, and a parameterized search engine (PSEAT) for searching the transformation search space. We use HPCToolkit to probe HW performance counters of the target platform and collect a variety of performance metrics [1]. Since we do not explicitly deal with feedback issues in autotuning in this research, the rest of this section is devoted to describing our transformation scripting language and the search engine.

4.1 Parameterization of Compiler Optimizations

We have used the POET transformation engine shown in Figure 4 to optimize the thread-level parallelism, memory locality, variable privatization, and register reuse for two SPEC95 benchmarks, *mgrid* and *swim*.

The core computation of *mgrid* is a 27-point stencil computation on a three dimensional space. We have focused on optimizing two performance critical subroutines, *RESID* and *PSINV*, as they take around 58% and 23% of the *mgrid* execution time respectively. Each subroutine has three perfectly-nested loops, all of which can be parallelized. In particular, none of the loops carry any dependence, but each loop carries a large number of inter-interation reuses of the

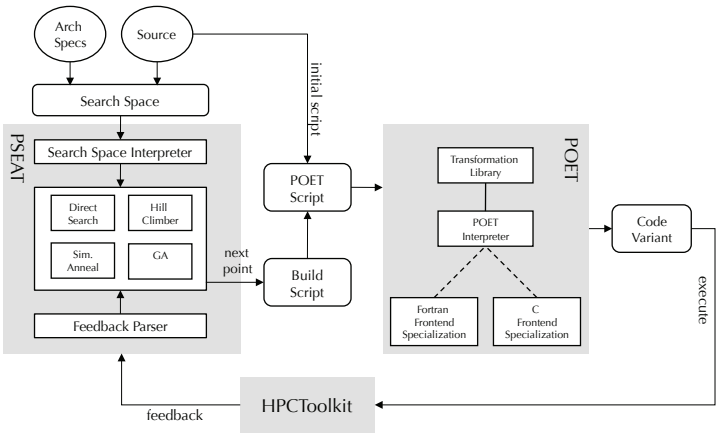


Fig. 4. Example execution patterns of parallelized and blocked numerical code

input data. We have applied three optimizations: OpenMP parallelization, loop blocking, and scalar replacement, for both subroutines, and have parameterized each optimization in the following fashion.

- Parallelization. We have parameterized which loops within each loop nest to parallelize, whether nested parallelism should be considered, the thread scheduling and chunk size computed by each thread, and the number of threads that will be used to evaluate the loops.
- Loop blocking. We have parameterized which loops to block and the blocking factor of each loop.
- Scalar Replacement. We have parameterezied whether to apply scalar replacement for each array referenced with the loop nests.

The core computation of *swim* includes two subroutines, *CALC1* and *CALC2*, each of which takes more than 30% of the overall execution time of *swim*. Each routine includes a sequence of three loop nests that can be selectively fused to improve register performance. Specifically, the first loop nest can be fused with either of the following loops, but not both. All loops can be parallelized. To optimize these loop nests, we have combined the application of OpenMP parallelization, loop blocking, loop fusion, loop unroll-and-jam, and scalar replacement. All transformations are parameterized in the following fashion.

- Parallelization. We have parameterized which loops to parallelize, and how many threads to use.
- Loop blocking. We have parameterized which loops to block, and the block factor for each loop.
- Loop fusion. We have parameterized which loops to fuse together.
- Scalar replacement. We have parameterized which loops to apply scalar replacement.

```
100                     # maximum number of program evaluations
3                       # number of dimensions in the search space
R 1 16                  # range : 1 .. 16
P 4                     # permutation : sequence length 4
E 2 8 16                # enumerated : two possible value 8 and 16
```

Fig. 5. Example configuration file for PSEAT

– Unroll and Jam. We have parameterized which loops to apply the transformation, and what unroll factor to use for each loop.

All the parameterized optimizations are implemented as source-to-source program transformations using POET [22, 21] and by invoking a POET optimization library which implements a large collection of compiler transformations such as loop fusion, loop blocking, scalar replacement, etc. First, an annotation is inserted as a comment within the original Fortran source code to identify each important sequence of loop nests to optimize. Then, a separate POET transformation script is written to apply necessary optimizations to the interesting loop nests. A large collection of command-line parameters are used to control how to apply each optimization in different ways. POET is a language specifically designed for parameterizing optimizations of application code for auto-tuning. For more details, see [21, 23].

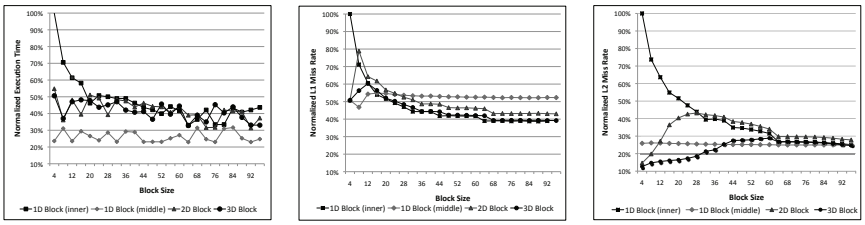
4.2 Searching the Space of Alternate Variants using PSEAT

Searching for alternate code variants is accomplished through the use of PSEAT - a parameterized search engine for automatic tuning. In most existing autotuning systems, the search module is tightly coupled with the transformation engine. PSEAT is designed to work as an independent search engine and provides a search API that can be used by other autotuning frameworks. This section discusses some of the design features of PSEAT and its integration into the the tuning system.

Input to PSEAT is a configuration file that describes the search space of optimization parameters. Fig. 5 shows an example configuration file. The syntax for describing a search space is fairly simple. Each line in the configuration file describes one search dimension. A dimension can be one of three types: *range* (R), *permutation* (P) or *enumerated* (E). *range* is used to specify numeric transformation parameters such as tile sizes and unroll factors. *permutation* specifies a transformation sequence and is useful when searching for the best phase-ordering.

An *enumerated* type is a special case of the *range* type. It can be used to describe a dimension where only a subset of points are feasible within a given range. An example of an *enumerated* type is the prefetch distance in software prefetching. In addition, PSEAT supports inter-dimensional constraints for all three dimension types. For example, if the unroll factor of an inner loop needs to be smaller than the tile size of an outer loop then this constraint is specified using a simple inequality within the configuration file.

PSEAT implements a number of search strategies including genetic algorithm, direct search, window search, taboo search, simulated annealing and random search. We include *random* in our framework as a benchmark search strat-



(a) Execution time sensitivity (b) L1 miss rate sensitivity (c) L2 miss rate sensitivity

Fig. 6. `mgrid` performance sensitivity to block size

egy. A search algorithm is considered effective only if it does better than random on a given search space.

5 Experimental Results

We performed a series of experiments exploring the search space of two SPEC95 benchmarks, *mgrid* and *spec*. This section summarizes our findings.

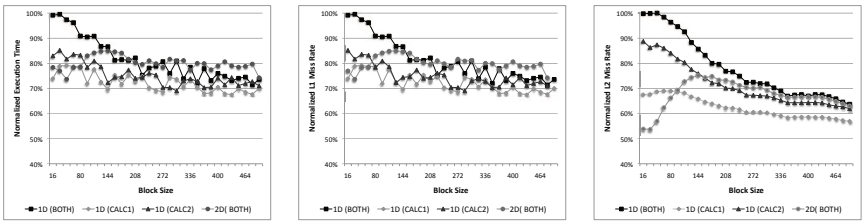
5.1 Experimental Setup

Platforms We present experimental results on three Intel-based multicore systems: a dual-core (*Core2Duo, Conroe*), a quad-core (*Core2Quad, Kentsfield*) and an eight-core machine with two quad-core processors (*Xeon, Nehalem*). Thus, these platforms serve as a good basis for evaluating our tuning strategy for portable performance.

Benchmarks For this study, we look at the performance characteristics of `mgrid` and `swim`, two scientific benchmarks from the SPEC95 benchmark suite. Both `mgrid` and `swim` contain several loops nests that can be fully parallelized. Most of these loop nests also exhibit a high-degree of temporal locality.

5.2 Performance Impact of Blocking

We first examine the blocking search space of `mgrid` and `swim`. 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. 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 2^2. 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.



(a) Execution time sensitivity (b) L1 miss rate sensitivity (c) L2 miss rate sensitivity

Fig. 7. `swim` performance sensitivity to block size

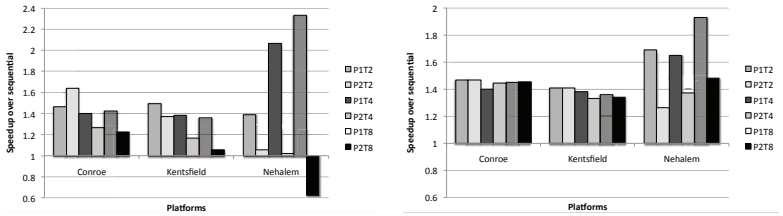
For `mgrid`, we explored four different blocking options the loop nests appearing in `resid` and `psinv` routines. For each blocking selection we explored a range of block sizes, starting from four and going up to 64 in steps of four. This gave rise to a two-level 15 dimensional search space with about 15^{16} points. The loop nests in `swim` are two-dimensional. Thus, the blocking search space of `swim` has fewer dimensions. However, the range within each dimension was larger, starting at 16 and going up 512.

Fig 6 shows how selection of blocking loops and choice of blocking factors impact performance of `mgrid`. The numbers presented are from experimental runs on *Kentsfield*. As we can see, there is significant variation in performance and L1 and L2 miss rates as we vary the block sizes. This is not surprising, since scientific code like `mgrid` are known to be sensitive to changes in the blocking factor. We notice that miss rates for L1 are high for smaller block sizes, and they gradually go down as we increase the block size. For L2, we observe a slightly different behavior. We notice that when multiple loops are blocked, the L2 miss rates are very low for smaller block sizes. This implies that multi-level blocking is able to exploit locality at multiple levels. However, for this to happen block sizes need to be very small (< 8). Interestingly, we observe that a reduction in the L1 or L2 miss rate does not necessarily correspond to performance gains. This means that although some blocking factors may be good for improving locality, they may have other adverse affects in terms of high loop instruction overhead and possibly reduced ILP. These results reiterate the need for tuning to find the optimal block size.

Fig 7 shows performance impact of block sizes on `swim`. We see a similar pattern in terms of L1 and L2 miss rates; although the variations are somewhat less for `swim`. In this case, we notice that the L1 miss rates do play a big role in determining overall performance. However, once again, L2 miss rates do not really seem to have a direct impact on performance.

5.3 Performance Impact of Parallelization Granularity

Similar to blocking, selecting loops for parallelization can have a significant impact on performance. We explored several different parallelization options in



(a) `mgrid` (b) `swim`

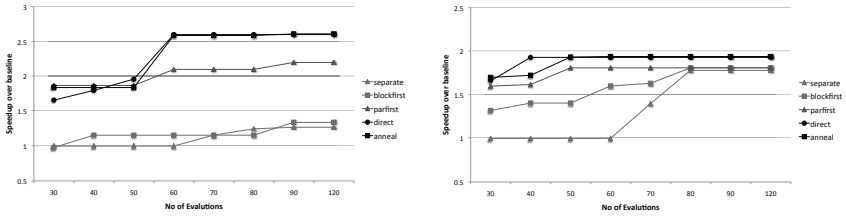
Fig. 8. Performance sensitivity to parallelization parameters

terms of number of loops selected for parallelization *and* the number of threads used for the parallel variant. Although thread scheduling and chunk size can also have a huge impact on performance, we did not explore these dimensions for this study. 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 `mgrid` and `swim` have significantly fewer dimensions then the blocking search spaces. For `mgrid`, we considered parallelizing loops at each level of the three dimensional loop nests in `resid` and `psinv` subroutines. For each variant, we also explored implementations with two, four and eight threads. For `swim`, we considered parallelizing both the inner and outer loops of `initial`, `calc1`, `calc2` and `calc3` routines.

Fig. 8(a) shows performance of six different parallel variants of `mgrid` on *Conroe*, *Kentsfield* and *Nehalem*. In the figure, `P1TK` means that the outermost loops in both `resid` and `psinv` was paralleled, and the parallel variant used `K` threads. `P2TK` means the middle loop was parallelized. We observe that generally, parallelizing the outermost loop is most profitable. However, this is not universally true. For example, on *Conroe*, the variant where the middle loop is parallelized performs the best. In terms of number of threads, as expected, setting the thread number to the available cores appear to work best. But again, there are exceptions. For example, on *Kentsfield* the best performance is obtained for `P1T2` where the number of threads is 2.

For `swim`, the performance sensitivity to different parallelization strategies is depicted in Fig. 8(b). For `swim`, we do not show numbers for parallelizing the inner dimension, as they always ran orders of magnitude slower than the sequential version. In Fig. 8(b), `P1TK`, refers to a parallel variant where all the outer loop of all four subroutines are parallelized, with `K` threads. The `P2TK` variant refers to the case where we only parallelize `calc1` and `calc2`. Overall, the selection of parallel loops tend to have less of an impact on `swim` than `mgrid`. However, even in this case, there is no single configuration that works well for all three platforms. On *Conroe*, `P2` performs better than `P1` when using four threads, whereas for the other two platforms `P1` appears to be the best choice.



(a) `mgrid` search space (b) `swim` search space

Fig. 9. Orthogonal vs. Multi-dimensional search

5.4 Non-orthogonality of Search Space

One of the main goals of this experimental study was to validate our claim that blocking and parallelization search dimensions are indeed non-orthogonal. To accomplish this, we set up an experiment where we explored the combined search space of blocking and parallelism using both orthogonal and non-orthogonal search methods. We chose three orthogonal search methods : one where the blocking dimensions are explored first (`blockfirst`), one where the parallelism dimensions are explored first (`parfirst`), and one where blocking and parallelism dimensions are explored independently and best value obtained from each search is used (`separate`). For multidimensional search, we selected simulated annealing (`anneal`) and direct search (`direct`), both of which are known to be effective in exploring the search space of transformation parameters [13]. To keep the comparison fair, individual dimension in the orthogonal search were searched using simulated annealing. We allowed each search algorithm to run for 120 iterations. We instrumented the search algorithms to output the *current best value* every ten iterations.

Fig. 9(a) shows results of exploring the combined search space of `mgrid` on the quad-core platform. We observe the both `direct` and `anneal` have a clear advantage over the orthogonal search methods when the number of evaluation goes beyond 50. For fewer number of iterations `parfirst` is able to compete with the multidimensional strategies but in the long run does not yield the desired performance. `separate` performs the worst, not finding a better value over the baseline until about the 60th iteration. In fact, most of the variants picked by `separate` lead to worse performance. Since we only display the best value *so far*, the speedup appears as 1. The poor performance of `separate` and `blockfirst`, and less than average performance of `parfirst` indicate that there is indeed interaction between the blocking and parallelization dimensions that are not captured by orthogonal search method. For the multi-dimensional search methods, there is no clear winner between `direct` and `anneal`. However, the overall speedup obtained by the two search methods is not that high. We speculate this could be attributed to the absence of scheduling and chunk size as a search space parameter.

Fig. 9(b) presents performance comparison of different search algorithms on the `swim` search space. Again, we notice that the multi-dimensional search strategies outperform the orthogonal techniques. However, in this case the performance gap is not as much. We also observe that the overall performance achieved by any search algorithm is less for `swim` than `mgrid`. This can be attributed to the finer thread granularity in `swim`. Because `swim` contains only two-dimensional loops, the amount of work per thread is not sufficient to offset the synchronization overheads. Moreover, blocking proved to be less effective for the two-dimensional case.

6 Conclusions

In this study, we explored the search space of parallelism and data locality transformations for multi-threaded applications. We presented a method for identifying and exposing tunable parameters to a search tool. Our experimental results illustrate the non-orthogonality of the search spaces and reinforces the need for application tuning through integrated transformation heuristics.

References

1. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience, To Appear*, 2009.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
3. L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the conference on Languages, compilers, and tools for embedded systems*, pages 231–239, 2004.
4. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. *LECTURE NOTES IN COMPUTER SCIENCE*, 4967:639, 2008.
5. S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
6. K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC08)*, 2008.
7. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *International Parallel and Distributed Processing Symposium*, San Francisco, CA, Apr. 2001. (Best Paper Award.).
8. F. Franchetti, M. Puschel, Y. Voronenko, S. Chellappa, and J. Moura. Discrete fourier transform on multicore. *Signal Processing Magazine, IEEE*, 26(6):90 –102, november 2009.

9. M. Frigo. A fast Fourier transform compiler. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
10. M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*, 2009.
11. S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.
12. K. Papadopoulos, K. Stavrou, and P. Trancoso. Helpercore_db: Exploiting multicore technology for databases. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, page 420, Washington, DC, USA, 2007. IEEE Computer Society.
13. A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on Supercomputing*, June 2006.
14. F. Song, S. Moore, and J. Dongarra. Feedback-directed thread scheduling with memory considerations. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, 2007.
15. W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *International Symposium on Microarchitecture*, 2007.
16. S. N. Vadlamani and S. F. Jenks. The synchronized pipelined parallelism model. In *The 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.
17. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Nov. 1998.
18. M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on MicroArchitecture*, pages 274–286, 1996.
19. M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
20. D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS00)*, page 171, Washington, DC, USA, 2000. IEEE Computer Society.
21. Q. Yi. The POET language manual, 2008. www.cs.utsa.edu/∼qingyi/POET/poet-manual.pdf.
22. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.
23. Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.
24. E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010.