On the Interaction of Tiling and Automatic Parallelization [⋆]

Zhelong Pan Brian Armstrong Hansang Bae Rudolf Eigenmann

Purdue University, School of ECE, West Lafayette, IN, 47907
{ zpan, barmstro, baeh, eigenman }@purdue.edu

Abstract. Iteration space tiling is a well-explored programming and compiler technique to enhance program locality. Its performance benefit appears obvious, as the ratio of processor versus memory speed increases continuously. In an effort to include a tiling pass into an advanced parallelizing compiler, we have found that the interaction of tiling and parallelization raises unexplored issues. Applying existing, sequential tiling techniques, followed by parallelization, leads to performance degradation in many programs. Applying tiling *after* parallelization without considering parallel execution semantics may lead to incorrect programs. Doing so conservatively, also introduces overhead in some of the measured programs. In this paper, we present an algorithm that applies tiling *in concert with* parallelization. The algorithm avoids the above negative effects. Our paper also presents the first comprehensive evaluation of tiling techniques on compiler-parallelized programs. Our tiling algorithm improves the SPEC CPU95 floating-point programs by up to 21% over non-tiled versions (4.9% on average) and the SPEC CPU2000 Fortran 77 programs up to 49% (11% on average). Notably, in about half of the benchmarks, tiling does not have a significant effect.

1 Introduction and Motivation

With processor speeds increasing faster than memory speeds, many compiler techniques have been developed to improve cache performance. Among them, iteration space tiling is a well known technique, used to reduce capacity misses [18, 20]. Tiling combines stripmining and loop-permutation to partition a loop's iteration space into smaller chunks, so as to help the data stay in the cache until it is reused. Several contributions have improved the initial tiling algorithms, by tiling imperfectly-nested loops [1, 16], carefully selecting the tile size, and avoiding conflict misses by copying and/or padding [7, 8, 11, 15, 17].

Enhancing locality is an important optimization technique to gain better performance, not only on a single processor, but also on a parallel machine. Tiling has been applied in parallelizing compilers, based on the sequential tiling algorithms [3, 9]. It has also been used in distributed memory machines [14]. The present paper was motivated by an effort to include a tiling technique into

[⋆] This work was supported in part by the National Science Foundation under Grants 0103582-EIA, and 0429535-CCF.

our Polaris parallelizing compiler [5, 13] for shared memory machines. (Polaris translates sequential Fortran 77 programs into parallel OpenMP form. The transformed parallel program will be compiled by the OpenMP backend compiler.) We have found existing tiling techniques to be insufficient for this purpose, as they are defined on a sequential program. Although a performance memory model has been presented that trades off parallelism against locality [9], it has not been discussed in the context of tiling. Also, a tiling technique for parallel programs was introduced in [3], however the interaction of the technique with other compiler passes has not been considered.

Without considering the interaction of tiling and parallelization, two approaches are open: *pre-parallelization tiling* and *post-parallelization tiling*. The *pre-parallelization tiling* algorithm performs tiling on the sequential program, followed by the parallelization pass. We have measured that this approach causes substantial performance degradation, primarily due to load imbalance of small loops. The *post-parallelization tiling* algorithm performs tiling after parallelization. To avoid incorrect results, this transformation needs to be conservative, also causing overheads. In Section 5, we will use these two tiling options as reference points and discuss their overheads in more detail.

The goal of this paper is to present an algorithm for *tiling in concert with parallelization*. First, the algorithm selects the candidate loop nests for tiling, based on data dependence and reuse information. Next, it trades off parallelism versus locality and performs the actual tiling transformation through loop stripmining and permutation. It factors parallelism information into tile sizes and the load balancing scheme. It also interacts with other parallelization passes by properly updating the list of private and reduction variable attributes.

Our algorithm outperforms both *pre-parallelization* and *post-parallelization* tiling. It improves the SPEC CPU95 floating point benchmarks by up to 21% (4.9% on average) over the parallel codes without tiling. The SPEC CPU 2000 Fortran 77 benchmarks are improved by up to 49% (11% on average). Our measurements confirm that tiling can have a significant performance impact on individual programs. However, they also show that, on today's architectures, about half of the programs benefit insignificantly.

The specific contributions of this paper are as follows:

1. We show how tiling affects the parallelism attributes of a loop nest. We prove these properties from data dependence information.
2. We introduce a new parallelism-aware tiling algorithm and show that it performs significantly better than existing techniques.
3. We discuss tiling-related issues in a parallelizing compiler: load balancing, tile size, and the trade-off between parallelism and locality.
4. We compare the performance of our algorithm with best alternatives. We discuss the measurements relative to an upper limit that tiling may achieve.

In the next section, we review some basic concepts of tiling, data reuse analysis, and data dependence directions. Section 3 analyzes the parallelism of tiled loops. Section 4 presents the algorithm for tiling in concert with parallelism and discusses related issues arising in a parallelizing compiler. Section 5 shows experimental results using the SPEC benchmarks and compares our new tiling algorithm to the pre-parallelization and post-parallelization tiling algorithms.

2 Background

2.1 Tiling Algorithm

Tiling techniques combine stripmining and loop permutation in order to reduce the volume of data accessed between two references to the same array element. Thus, it increases the chances that cached data can be reused. It has often been shown that tiling can significantly improve the performance of matrix multiplication and related linear algebra algorithms [8, 9, 10].

(a) Matrix Multiply

```
DO I = 1, M
  DO K = 1, M
    DO J = 1, M
      Z(J,I) = Z(J,I) + X(K,I) * Y(J,K)
```

(b) Tiled Matrix Multiply

```
DO K2 = 1, M, B
  DO J2 = 1, M, B
    DO I = 1, M
      DO K1 = K2, MIN(K2+B-1,M)
        DO J1 = J2, MIN(J2+B-1,M)
          Z(J1,I) = Z(J1,I) + X(K1,I) * Y(J1,K1)
```

Fig. 1. Tiling of a matrix multiplication code

Figure 1 shows a simple example of the original and the tiled versions of a matrix multiplication code. Loop K in the original version is stripmined into two loops, K1 and K2. Loop K1 in the tiled version iterates through a strip of B; we call it the *in-strip* loop. Loop K2 in the tiled version iterates across different strips; we call this the *cross-strip* loop. B is called the tile size.

2.2 Data Reuse Analysis

Data reuse analysis [10] identifies program data that is accessed repeatedly, and it quantifies the amount of data touched between consecutive accesses. To improve data locality, one attempts to permute loop nests so as to place the loop that carries the most reuse in the innermost position. If there are multiple accesses to the same memory location, we say that there is temporal reuse. If there are multiple accesses to a nearby memory location that share the same cache line, we say that there is spatial reuse. Both types of reuse may result from a single array reference, which we call self reuse, or from multiple references, which we call group reuse [12, 18].

2.3 Direction Vectors

In this paper, we use data dependence direction vectors to determine parallelism and the legality of a loop permutation. The direction "<" denotes a forward cross-iteration dependence. The direction ">" denotes a backward cross-iteration dependence. We refer to [2, 4, 22, 21] for a thorough description of direction vectors. We make use of the following lemmas, given in these papers.

Lemma 1 ***Reordering:*** *Permuting the loops of a nest reorders the elements of the direction vector in the same way.*

Lemma 2 ***Permutability:*** *A loop permutation is legal as long as it does not produce an illegal direction vector. In a legal direction vector, the leftmost non-equal direction must be "<" (i.e., it cannot be ">").*

Lemma 3 ***Parallelism:*** *Given a direction vector, its leftmost "<" direction makes the corresponding loop serial. Furthermore, serializing this loop* covers *the given dependence on all inner loops. That is, w.r.t. this dependence, all inner loops are parallel.*

Lemma 4 *After stripmining the loop L into (L', L''), its direction vector changes from $[d]$ to $[d', d'']$, as follows: $[=] \to [=,=]$; $[<] \to [=,<]$ or $[<,*]$; $[>] \to [=,>]$ or $[>,*]$. That is, either the cross-strip direction is "=" and the in-strip loop takes on the direction of the original loop, or the cross-strip loop takes on the original direction ("<" or ">") and the in-strip direction becomes unknown ("*").*

Direction vectors of the original (pre-tiling) loops can be used to determine the direction vectors of the tiled loops [23]. Lemma 1 and 4 aid in deriving those new direction vectors. Lemma 2 aids in finding all legally tiled versions of a loop nest. Lemma 3 decides parallelism of the tiled loops.

3 Parallelism of Tiled Loops

Theorem 1 *After tiling, the in-strip loops have the same parallelism as the original ones. The cross-strip loop L'_i is serial, if the corresponding original loop L_i is serial. But the cross-strip loop may become serial, even if the corresponding original is parallel.*

```
       (a) Original loop                              (b) Tiled loop
                                          DO J1 = 1, M, B                (serial)
DO I = 1, N              (serial)           DO I = 1, N                  (serial)
  DO J = 1, M            (parallel)           DO J = J1, MIN(J1+B-1,M)   (parallel)
    A(J,I) = A(J+1,I+1)                         A(J,I) = A(J+1,I+1)
```

Fig. 2. Reduced parallelism as a result of tiling.

This theorem can be strictly proved based on the previous lemmas. In this paper, limited by space, we only explain the rationales. The dependence inside one tile is essentially the same as the dependence in the pre-tiling loops. So, parallelism of the in-strip loops is not changed. According to Lemma 4, tiling introduces new dependence across the tiles. So, a cross-strip loop may become serial. Figure 2 shows an example. In the original loop, loop I is serial and loop J is parallel. After tiling, the cross-strip loop J1 is serial, loop I is serial, and loop J is parallel.

Theorem 1 shows that a parallelizing compiler cannot simply apply tiling after parallelization. The compiler needs to analyze the parallelism of the cross-strip loops, unless it only chooses to parallelize the in-strip loops, whose parallelism does not change. The following section develops the tiling algorithm considering the interaction of tiling and parallelization.

4 Parallelism-aware Tiling

4.1 Algorithm

Our tiling in concert with parallelization algorithm uses the direction vectors of the original loop nest to determine the parallelism of the tiled loop nest, as discussed in Section 2 and Section 3. It then trades off parallelism and locality and determines a balanced tile size. Figure 3 shows the pseudo code.

```
Subroutine ParallelTiling(LoopNest $L$)
1  $P$ = the number of processors;
2  $DVs$ = the set of all direction vectors;
3  Perform data reuse analysis;
4  For each possible tiled version $V$ of $L$
5      Decide parallelism of $V$ based on the $DVs$;
6      $C$ = the cost of $V$ based on its parallelism and reuse information;
7  $X$ = the tiled version with the least cost;
8  $T$ = raw tile size computed by LRW [10], considering
         loop parallelism and cache configuration;
9  $S$ = BalancedTileSize($X$,$T$,$P$);
10 Substitute the tile size $S$ into the tiled version $X$;
11 Update reduction/private variable attributes;
12 Generate two versions if iteration number unknown;
   //$L$ is called when not enough iterations; Otherwise, $X$ is called.
```

Fig. 3. Parallelism-aware tiling algorithm

Our algorithm considers all legally tiled loop nest versions and selects the one with the least cost. It is worth noting that the order of the cross-strip loops may be different from the order of the in-strip loops. Enumerating all possible tiled versions is feasible, because most loops are nested with two or three levels. Step 5 follows Section 2 and Section 3. Step 6 uses a simple model that assumes that placing a parallel loop in an outer position is preferable over increased reuse, which will be discussed in Section 4.2. (This model suffices for our machine environment; more advanced schemes can be used without change of the algorithm). Steps 8 and 9 follow Sections 4.3 and 4.4, respectively. Step 12 is important for reducing potential tiling overheads. If the number of loop iterations is unknown at compile time, a two-version loop is created that selects between the tiled and non-tiled variants at runtime.

Our compiler pass also deals with imperfectly nested loops. It transforms such loops into perfect nests through loop fusion, loop distribution and code sinking [19]. Inner loops with fixed small number of iterations are unrolled. Then, tiling is applied to the perfectly nested loops. We have verified that this approach generates comparable results to the methods proposed in [1, 16] for the SPEC CPU benchmarks, except in TOMCATV and SWIM. (In SWIM, the higher performance was achieved through manual source modifications; TOMCATV is an obsolete benchmark.)

4.2 Trading off Parallelism and Locality

Locality enhancement and parallelization may have conflicting performance goals. Per Theorem 1, although the parallelism of the in-strip loops is the same as that of the original loops, the parallelism of the cross-strip loops can be different. For example, in Figure 2, loop I is serial in both versions, while loop J is parallel in both versions. However, after tiling, the cross-strip loop, $J1$, becomes serial. Thus, the tiled nest invokes the parallel loop more times than the original loop nest, causing higher fork-join overhead.

Table 1. Effect of tiling on fork-join overhead

Original parallelism	Parallelism after tiling	Fork-join overhead
$[S, P]$	$[S, S, P]$	increased
$[S, P]$	$[P, S, P]$	decreased
$[P, S]$	$[S, P, S]$	increased
$[S, S]$	$[S, S, S]$	not changed
$[P, P]$	$[P, P, P]$	not changed

Five different scenarios may occur after tiling a doubly nested loop, depending on the parallelism. We list these cases in Table 1. S indicates that the corresponding loop is serial; P indicates the loop is parallel. (If there are more than two loops, the change in fork-join overhead can be determined by similar analysis.) For example, in the first and second rows of Table 1, the original outer loop is serial and the original inner loop is parallel. Per Section 4, after tiling, the cross-strip loop is serial (in Row 1) or parallel (in Row 2), which results in an increase or decrease of fork-join overhead.

In summary, the parallelism of the cross-strip loops determines if tiling will increase or reduce fork-join overhead. Thus, tiling a parallel program can result in either higher or reduced parallel loop execution cost. In an advanced performance model, both the fork-join overhead and the benefit of increased locality need to be considered.

4.3 Tile Size Selection

The tile size is a critical parameter for tiling. We use the LRW algorithm [10] to compute the raw tile size, which fits in cache. In addition, for distributed caches, if the parallel loop is an in-strip loop, a tile needs to fit in multiple caches; for a shared cache, if the parallel loop is a cross-strip loop, the cache needs to hold multiple tiles. So, the computation of the raw tile size depends on the cache configurations and parallelism of the loops. This computed raw tile size is tuned further to balance the loads among processors, which will be described in Section 4.4.

4.4 Load Balancing

If the cross-strip loop is executed in parallel, load balancing can become an important issue. Tiling splits the number of iterations of the parallel loop into chunks. If the split is uneven, load imbalance results. This effect is more pronounced for programs or program sections that operate on small data sets relative to the available cache size. Hence, for a given data size, this issue tends to increase with newer generations of processors.

For example, in Figure 4, all loops are parallel and the tile size is 80. Suppose we run the program on a four processor machine. Before tiling, loop I has 512 iterations and each processor executes 128 iterations. But after tiling, the cross-strip loop $J1$ has $512/80 + 1 = 7$ iterations, which cannot be evenly divided among the processors, causing load imbalance.

```
(a) Before tiling (balanced)    (b) After tiling (not balanced)
                                DO J1 = 1, 512, 80
DO I = 1, 512                     DO I = 1, 512
  DO J = 1, 512                     DO J = 1, MIN(J1+79,512)
    ...                               ...
```

Fig. 4. Load imbalance after tiling

Tiling sequential loops does not require balanced strip-mining. The tile size is obtained by computing the number of memory references that fit in the cache. However, the parallelizing compiler needs to tune the tile size, so that each processor will execute nearly the same number of iterations. For the previous example, the compiler can set the tile size to be 64. Then, after tiling the cross-strip loop $J1$ has $512/64 = 8$ iterations. Each processor will get 2 iterations with the same load. A more general rule is to find the largest size that is less than the original tile size and that creates a balanced load:

Suppose that the un-tuned, raw tile size is T, the number of iterations is I, and the number of processors is P. We choose a tile size S such that $S * P$ is divisible by I and S is as close to T as possible, based on the following formula.

$$S = \frac{I}{\lceil I/(P*T) \rceil * P}$$

This formula applies to parallel cross-strip loops (cases 2, 3, and 5 in Table 1). If the in-strip loop is parallel (case 1), it suffices to make the tile size a multiple of the number of processors.

5 Experiments

5.1 Reference Points: Tiling Independent of Parallelization

In order to verify the effectiveness of our tiling algorithm, we compare it with two algorithms that apply tiling independent of parallelization: *pre-parallelization* *tiling* and *post-parallelization tiling*. To our knowledge, they represent the best that can be realized with tiling techniques for sequential programs, as proposed in related work.

Pre-parallelization tiling determines the tiled loop shape and tile size before the parallelization passes. In most cases, the chosen parallel loop is a cross-strip loop. Load balancing, as discussed in Section 4.4 is not applied. Sequential tiling semantics is fully valid in this case, as parallelization has not yet been applied.

Post-parallelization tiling would generate incorrect code, if the tiling algorithm simply propagated parallel loop attributes from an original loop to its stripmined pair. So, conservatively, the cross-strip loop is *always* serialized. To further increase the fairness of our comparison, we have added an optimization to reduce fork-join overheads, when the scheduled parallel loop does not carry cross-processor dependences. In that case, this optimization moves the parallel region to the outer loops and reduces the number of barrier synchronizations by using the OpenMP "*nowait*" clause on the parallel loop.

5.2 Experimental Environment

We implemented the tiling algorithm presented in Section 4 in the Polaris [6, 13] parallelizing compiler. The experiments were done on an Ultra SPARC II machine with four 250 MHZ processors. Each processor is equipped with a $16K$ direct-mapped L1 cache and a $1M$ direct-mapped L2 cache. Both caches are distributed. We measured the performance of all compiler-parallelized SPEC CPU95 floating point benchmarks with and without tiling. In addition, in order to evaluate how increasing data sets impacts the performance of tiling, we measured all of the six SPEC CPU2000 Fortran 77 benchmarks.

5.3 Experimental Results

The baseline of our experimentation is parallelization without tiling. In Figure 5 and Figure 6, the first two bars for every benchmark show the performance of the reference points. The third bar shows the performance of our new algorithm for tiling in concert with parallelization.

In most benchmarks of the SPEC CPU95 suite, pre-parallelization tiling does not improve performance over parallelization without tiling. Two main effects degrade the performance of APSI, HYDRO2D, MGRID, SWIM and TOMCATV significantly. First, the data size is small relative to the available cache size, so that the important loops contain very few tiles, causing load imbalance. Second, since no information about parallel loops is known, the tiling algorithm does not permute the most beneficial loops to outermost positions. In the SPEC CPU2000 codes, the data size is much larger, reducing the load imbalance effect. For the measured SPEC CPU2000 benchmarks, APSI, APPLU and SWIM show improvements over parallelization alone. SWIM is improved by 49%, most of which is due to the fact that tiling yields many stride-one access patterns.

In the SPEC CPU95 suite, post-parallelization tiling also degrades the performance over parallelization without tiling for APSI, HYDRO2D, MGRID and

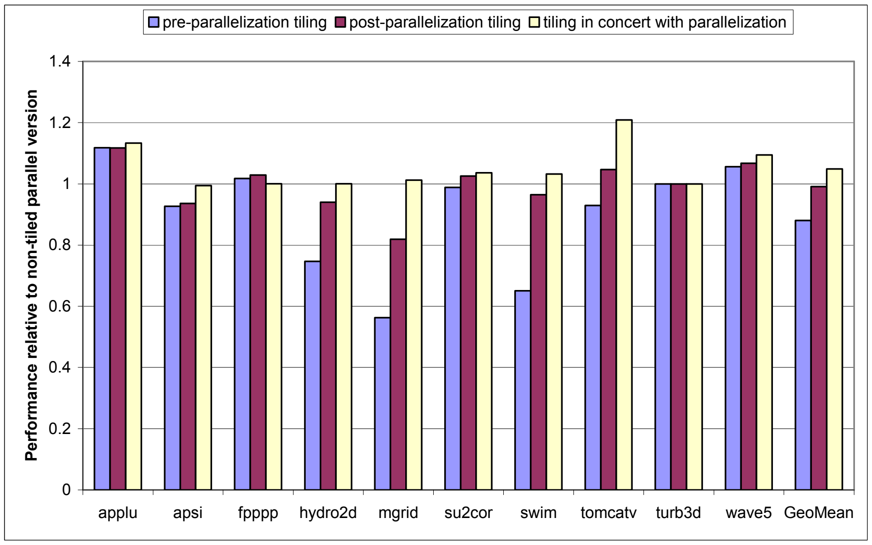

Fig. 5. Performance of tiling relative to non-tiled parallel codes for SPEC95

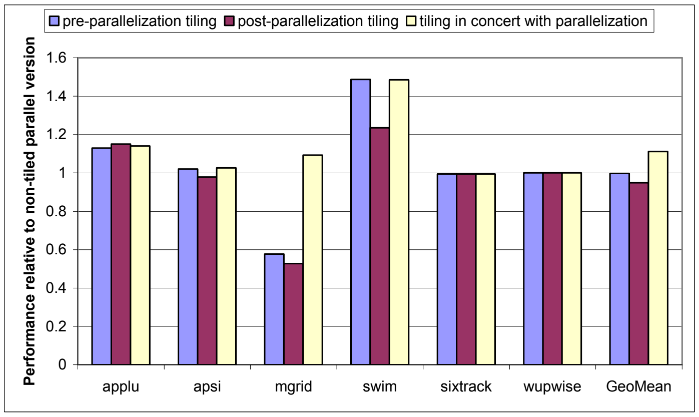

Fig. 6. Performance of tiling relative to non-tiled parallel codes for selected SPEC2000 benchmarks

SWIM. First, post-parallelization tiling may cause load imbalance, if it parallelizes the in-strip loop and the tile size is small. Second, in post-parallelization tiling, the chosen parallel loop tends to have finer granularity than in tiling in concert with parallelization. Although the synchronization optimization reduces the fork-join overhead, some of this overhead remains. Another observation is that, in general, post-parallelization tiling performs better than pre-parallelization tiling for SPEC CPU95 benchmarks, but not for SPEC CPU2000. The reason is that a large data size reduces load imbalance for pre-parallelization tiling, but not for post-parallelization tiling.

Our experiments show that tiling in concert with parallelization performs significantly better than tiling independent of parallelization. Our new algorithm never degrades performance. Five out of the ten benchmarks in the SPEC CPU95 suite show improvements over parallelization alone. The largest improvement is 21% in TOMCATV (TOMCATV is a small kernel benchmark, which is now considered obsolete). Tiling can add some control overhead, offsetting parallel performance. We have found this to be the reason for very minor performance degradation to APSI and HYDRO2D. In FPPPP, post-parallelization tiling performs slightly better than our algorithm. It is a rare case where the cost of computing a balanced tile size at runtime is noticeable. Most SPEC CPU2000 benchmarks show improvements. SWIM is improved by 49%. It is important to note, that although matrix multiply (a code frequently used to demonstrate tiling) is very important in WUPWISE, all matrices are small and do not benefit from tiling.

As expected, our measurements show only small improvements on the SPEC CPU95 codes, whose data sets mostly fit in cache. Tiling improves more significantly the SPEC CPU2000 codes, which have larger data sets and, consequently, increased cache misses in the original programs.

5.4 On Performance Bounds for Tiling

The fact that half of our program suite does not benefit from tiling raises the question of how much better a further improved algorithm could perform. In order to find an upper bound on the performance achievable by tiling, we measured the percentage of tilable loops in the SPEC CPU95 benchmarks based on reuse analysis. A tilable loop nest must satisfy two conditions. First, at least two loops in the nest carry reuses, otherwise loop interchanging would suffice. Second, it does not contain subroutine calls or I/O operations. The last column in Table 2 shows the execution time percentage of the loop nests satisfying both conditions. This percentage gives us an upper bound on tilable loops.

For all benchmarks other than HYDRO2D, the result in Table 2 is consistent with that in Figure 5. The benchmarks gaining significant performance from our tiling algorithm spend a large percentage of execution time in tilable loop nests, and vice versa. In HYDRO2D, although 53.7% of the execution time is spent in

Table 2. Percentage of tilable loops based on reuse analysis. Each column shows, respectively, the numbers of loops, loops carrying reuses, loop nests with at least two loops carrying reuses, and those loop nests without subroutine calls or I/O operations. The data in the parentheses are the execution time percentage of the loop nests with more than two loops carrying reuses and without subroutine calls.

Benchmark	Total	Reuse	Nested	w/o Call
APPLU	149	125	55	54 (97.60%)
APSI	388	310	111	59 (19.50%)
FPPPP	49	37	15	8 (5.80%)
HYDRO2D	170	117	21	21 (53.70%)
MGRID	38	24	8	8 (86.40%)
SU2COR	208	177	37	22 (14.90%)
SWIM	24	15	3	3 (60.10%)
TOMCATV	16	14	5	5 (95.90%)
TURB3D	64	43	12	11 (22.20%)
WAVE5	362	274	59	57 (19.70%)

tilable loops, each loop nest only refers to a small amount of memory, which can fit into cache. Therefore, tiling does not reduce cache misses.

Our results also show that, while tiling can be an important locality enhancement technique for individual programs, especially for stencil operations, its performance benefit is not as broad as commonly assumed. Tiling does not gain significant performance in half of the benchmarks. The major reason is limited data reuse that is amenable to tiling.

6 Conclusions

We have presented a new tiling algorithm that works in concert with other parallelization passes. We have shown that applying existing tiling techniques, designed for sequential programs before or after parallelization, would lead to significant performance degradation or incorrect programs. Our algorithm avoids these negative effects, hence it represents new technology, relevant to any parallelizing compiler.

Furthermore, in evaluating tiling techniques comprehensively, we have found that the benefit is less than commonly assumed. Tiling – along with other locality enhancement techniques – is believed to be very important, as the memory-to-processor speed ratio in new computer architectures keeps decreasing. However, this technique has often been demonstrated on simple linear algebra kernels. Although our measurements confirm improvements on stencil computations, tiling has only limited effect on other programs, which is due to limited data reuse, amenable to tiling. Increasing cache sizes and increasing data sets are two opposite trends that will impact the performance of tiling techniques on future computer systems.

References

1. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 31, 2000.
2. Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*, chapter Dependence: Theory and Practice, pages 45–55. Morgan Kaufman Publishers, 2002.
3. Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the conference on Programming language design and implementation*, pages 112–125, 1993.
4. Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.
5. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with polaris, 1996.

6. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
7. Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th international conference on Supercomputing*, pages 492–499, 1999.
8. Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the conference on Programming language design and implementation*, pages 279–290, 1995.
9. Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 6th international conference on Supercomputing*, pages 323–334, 1992.
10. Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, 1991.
11. Zhiyuan Li. Optimal skewed tiling for cache locality enhancement. In *17th International Parallel and Distributed Processing Symposium*, 2003.
12. Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
13. Seung Jai Min, Seon Wook Kim, Michael Voss, Sang Ik Lee, and Rudolf Eigenmann. Portable compilers for OpenMP. In *Lecture Notes in Computer Science, 2104*, pages 11–19, Jul 2001.
14. J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.
15. Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the Eighth International Conference on Compiler Construction*, 1999.
16. Yonghong Song and Zhiyuan Li. A compiler framework for tiling imperfectly-nested loops. In *Languages and Compilers for Parallel Computing*, pages 185–200, 1999.
17. Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 215–228, 1999.
18. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the conference on Programming language design and implementation*, pages 30–44, 1991.
19. Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, 1996.
20. M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, 1989.
21. M. J. Wolfe. *Optimizing supercompilers for supercomputers.* PhD thesis, 1982.
22. Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *Int. J. Parallel Program.*, 16(2):137–178, 1987.
23. Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.