Tile Size Selection Revisited

SANYAM MEHTA, GAUTHAM BEERAKA, and PEN-CHUNG YEW, University of Minnesota

Loop tiling is a widely used loop transformation to enhance data locality and allow data reuse. In the tiled code, however, tiles of different sizes can lead to significant variation in performance. Thus, selection of an optimal tile size is critical to performance of tiled codes.

In the past, tile size selection has been attempted using both static analytical and dynamic empirical (auto-tuning) models. Past work using static models assumed a direct-mapped cache for the purpose of analysis and thus proved to be less robust. On the other hand, the auto-tuning models involve an exhaustive search in a large space of tiled codes. In this article, we propose a new analytical model for tile size selection that leverages the high set associativity in modern caches to minimize conflict misses. Our tile size selection model targets data reuse in multiple levels of cache. In addition, it considers the interaction of tiling with the SIMD unit in modern processors in estimating the optimal tile size. We find that these factors, not considered in previous models, are critical in developing a robust model for tile size selection. We implement our tile size selection model in a polyhedral compiler and test it on 12 benchmark kernels using two different problem sizes. Our model outperforms the previous analytical models that are based on reusing data in a single level of cache and achieves an average performance improvement of 9.7% and 20.4%, respectively, over the best square (cubic) tiles for the two problem sizes. In addition, the tile size chosen by our tile size selection algorithm is similar to the best performing size obtained through an extensive search, validating the analytical model underlying the algorithm.

Categories and Subject Descriptors: D.3.4 [Processors]: Compilers, Optimization

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Loop tiling, tile size selection, multilevel cache, vectorization

ACM Reference Format:

Mehta, S., Beeraka, G., and Yew, P.-C. 2013. Tile size selection revisited. *ACM Trans. Architect. Code Optim.* 10, 4, Article 35 (December 2013), 27 pages.

DOI: <http://dx.doi.org/10.1145/2555289.2555292>

1. INTRODUCTION

Loop tiling [Wolfe 1989; Wolf and Lam 1991; Ramanujam and Sadayappan 1992; Lim et al. 2001] is a widely used loop transformation to enhance data reuse in higher levels of memory hierarchy. In essence, loop tiling reduces the reuse distance from being a function of the problem size to a function of the tile size. Loop tiling thus minimizes the cache capacity misses. However, tiling leads to noncontiguous data accesses in memory, resulting in an increased probability of conflict misses in the cache. These conflict misses are a function of the tile size in tiled code. For example, for a problem size of ($N = 2,000$), a 2D tiled code of the *dsyr2k* kernel from BLAS library [Dongarra et al. 1990] with the tile size ($TS_1 = 8 \times 128$) performs $2.44\times$ better than the one with the tile size ($TS_2 = 128 \times 8$) even though both codes have the same working set size

New Article, Not an Extension of a Conference Paper.

Authors' addresses: S. Mehta, University of Minnesota - Computer Science; email: mehta121@umn.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481 or permissions@acm.org.

© 2013 ACM 1544-3566/2013/12-ART35 \$15.00

DOI: <http://dx.doi.org/10.1145/2555289.2555292>

in the L1 cache, when tested on an Intel Xeon processor based on the Sandy Bridge microarchitecture. One of the key reasons for this significant difference in performance is that the tile size TS_2 leads to pronounced conflict misses that are 22 times more compared to those observed for tile size TS_1 . Thus, tile size selection is critical to performance of the tiled code, and an optimal tile size is one that minimizes not only capacity but also conflict misses within a tile.

In the past, the problem of tile size selection has been attempted using analytical model-driven approaches [Lam et al. 1991; Esseghir 1993; Coleman and McKinley 1995; Ghosh et al. 1997; Sarkar 1997; Chame and Moon 1999; Sarkar and Megiddo 2000; Shirako et al. 2012]. However, these approaches have proved to be less robust because the models used did not fully capture the interaction between the source program (features like problem size and reuse characteristics of the arrays) and critical features in the modern processor microarchitecture such as multilevel and set-associative caches and the SIMD unit. This has resulted in a widening gap between performance delivered by best known tile sizes and that achieved by using tile sizes predicted by the previous analytical models.

Tile size selection has also been widely studied using empirical auto-tuning, as in ATLAS and PHiPAC for linear algebra [Bilmes et al. 1997; Whaley et al. 2001], FFTW and SPIRAL for signal processing [Frigo 1999; Xiong et al. 2001], ETile [Tavarageri et al. 2011], and others [Knijnenburg et al. 2003; Yuki et al. 2010]. However, each of these techniques is faced with a large search space of tile sizes when considering multidimensional, noncubic tiling—for example, the size of search space for a level-three BLAS kernel such as *gemm* scales at n^3 , where n is a function of the size of cache being considered. If reuse opportunity in multiple levels of cache must be considered, the search space expands significantly for the larger caches, lower in the memory hierarchy. As a result, the adopted approach is either too time consuming [Bilmes et al. 1997; Frigo 1999; Whaley et al. 2001; Xiong et al. 2001; Knijnenburg et al. 2003] or less accurate when heuristics are used to reduce the search space [Yuki et al. 2010].

In this article, we propose a new analytical model for selecting tile size in modern processors. The Tile Size Selection (TSS) algorithm proposed in our model chooses a tile size such that the following four objectives are met:

- It leverages the high set associativity in modern caches to minimize interference and yield stable performance for all ranges of problem sizes in a variety of source programs.
- It considers data reuse at multiple levels of cache, which further boosts the performance of the tiled code.
- It considers the interaction of tiling with the SIMD unit on the host architecture and chooses a tile size that best benefits from it.
- It considers the impact of tiled code execution in a multithreaded environment (both Chip Multiprocessing [CMP] and Simultaneous Multithreading [SMT] environments) and achieves good performance.

Although the analytical model developed in this article chooses a particular tile size, it could also be used in conjunction with an auto-tuning framework to prune/navigate the search space.

We implement the TSS algorithm within a source-to-source polyhedral compiler framework, PLuTo [Bondhugula et al. 2008], which automatically tiles the source programs. For the generated tiled code, PLuTo chooses a cubic tile size by default. The TSS algorithm, instead, estimates the optimal tile size that is then used by PLuTo to generate the corresponding tiled code. We tested our model on 12 benchmarks comprising a mix of linear algebra, data mining, and stencil kernels, all of which are known to benefit significantly from tiling. We tested each kernel with two different problem sizes on

two different machines with different microarchitectures. Experimental results show that accommodating the impact of multilevel caches and the SIMD unit within the analytical model adds significant performance to the tiled code, and the tile size thus chosen is similar to that obtained from an exhaustive search for the best tiled code in a constrained search space. In comparison to the best square (cubic) tiled code for the test benchmarks, our TSS algorithm chooses tile sizes that perform 9.7% and 20.4% faster on average, respectively, for the two problem sizes. We also show that our model achieves good performance for tiled codes run in a multithreaded environment using either the CMP or the SMT technology.

The rest of the article is organized as follows. Section 2 re-evaluates the problem of tile size selection from the point of view of a compiler via a case study and reinforces the motivation for this work. Section 3 gives a background about the problem of tile size selection. Section 4 discusses the multiple factors that influence tile size selection and explains our approach to accommodate them in our TSS model. In Section 5, we describe our algorithm for estimating the optimal tile size and discuss its scope. Section 6 describes the experimental setup and the benchmarks used for our experiments. We discuss the experimental results in Section 7. The related work is presented in Section 8. Finally, we conclude in Section 9.

2. MOTIVATION

Tile size selection has been deemed a complex problem from the point of view of a compiler. In this section, we re-evaluate the major reasons behind such a view and present a viable alternative to tackle the problem of tile size selection at compile time.

2.1. Modeling the Effect of Set Associativity

An analytical model for tile size selection must choose a tile size such that the conflict misses are minimized. Past works did not consider set-associative caches and proposed probabilistic approaches [Coleman and McKinley 1995; Chame and Moon 1999; Rivera and Tseng 1999; Panda et al. 1999; Hsu and Kremer 2004] for minimizing conflicts among array references. In such a scenario, a conservative approach based on eliminating conflict misses as in Coleman and McKinley [1995] and Chame and Moon [1999] results in tiles with undesirable shapes (too “skinny” or too “fat” tiles) for some problem sizes, leading to suboptimal performance in those cases. On the other hand, an optimistic approach as in Rivera and Tseng [1999], Panda et al. [1999], and Hsu and Kremer [2004] is based on ignoring the nonunity set associativity of caches. This too gives less than optimal performance because multiple cache lines accessed in the source program may map to the same set of an n -way set-associative cache, causing conflicts more likely than that predicted by such an optimistic analytical model.

With multiple array references within the loop body of a source program, modeling the effect of set associativity on the execution time behavior of a tiled code has been considered nontrivial. We re-evaluate this problem in the example of matrix-multiplication (*matmul*) kernel shown in Figure 1.

In *2D-tiled matmul*, the maximum working set that should fit the cache comprises all of the data accessed in a single iteration of outermost untiled loop i as shown in Figure 1(b). Thus, the working set consists of a tile row for each of arrays A and C (i.e., K and J elements, respectively) and an entire tile of array B (i.e., $K \times J$ elements).¹ Clearly, the working set is dominated by the data accessed by array reference B . Further, if this working set fits a particular level of cache, the entire tile of array B can be reused in each iteration of loop i . Thus, for *matmul* (and similar other kernels that allow data

¹If an LRU cache replacement policy is considered instead of an optimal replacement policy, then the working set will also include another tile row of array C and an element of array A [Yotov et al. 2005].

```

for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
(a) matmul

for(jT=0; jT < N/j; jT++)
  for(kT=0; kT < N/K; kT++)
    for(i = 0; i < N; i++)
      for(k = K*kT; k < K*kT + K-1; k++)
        /* vector loop */
        for(j = jT*T; j < jT*T + J-1; j++)
          C[i][j] += A[i][k] * B[k][j];
(b) 2D-tiled matmul

for(iT=0; iT < N/i; iT++)
  for(jT=0; jT < N/j; jT++)
    for(kT=0; kT < N/K; kT++)
      for(i = i*T; i < i*T + I-1; i++)
        for(k = K*kT; k < K*kT + K-1; k++)
          /* vector loop */
          for(j = j*T; j < j*T + J-1; j++)
            C[i][j] += A[i][k] * B[k][j];
(c) 3D-tiled matmul

for(t=0; t < T; t++) {
  for(i=2; i < N-1; i++)
    for(j=2; j < N-1; j++)
      S1: B[i][j]=0.2*(A[i][j]+A[i][j-1]+A[i][1+j]+A[1+i][j]+A[i-1][j]);
  for(i=2; i < N-1; i++)
    for(j=2; j < N-1; j++)
      S2: A[i][j]=B[i][j];
}
(d) Jacobi-2D

for(tT..) // serial loop
for(iT..) // parallel loop
for(jT..)
for(t..)
for(i..)
for(j..) {
  S1: B[-2*t+i][j-2*t+j]=0.2*(A[-2*t+i][j-2*t+j]+...);
  S2: A[-2*t+i-1][j-2*t+j-1]=B[-2*t+i-1][j-2*t+j-1];
}
(e) Time-skewed Jacobi-2D

```

skewing factors for
loop i and loop j

S1 and S2
fused

Fig. 1. Source codes of *matmul* and *Jacobi-2d*, before and after tiling through PLoTo.

reuse), the task of modeling the effect of set associativity for multiple array references in the source program is reduced to only those references with temporal reuse in the outermost loop, such as the array reference B . Therefore, in a set-associative cache, most ways could be assigned to such a reference and the effect of set associativity be modeled only for that reference, independent of the other array references. However, if there are multiple array references with temporal reuse in the outermost loop (as in the *Jacobi-2d* kernel shown in Figure 1(d) that has two such references, $A[i][j]$ and $B[i][j]$), the available ways in each set can be equally partitioned among them and the effect of set associativity be modeled for any one of them, as all such references exhibit similar runtime behavior.

2.2. Tapping into Data Reuse at Multiple Levels of Cache

With multilevel caches on modern processor architectures, tapping into the reuse opportunities at the different levels of cache is important for effective tiling. For example, the best 2D tile for *matmul* performs 13% worse than the best 3D tile that considers data reuse in the L2 cache on an Intel Sandy Bridge core for a problem size of 2,000. *The possibility of reusing data in different levels of cache arises from the fact that different array references have different reuse distances.* For example, in the 3D-tiled *matmul* shown in Figure 1(c), each tile of array B is reusable in every iteration of loop i , whereas each tile of array C is reusable in every iteration of loop kT .

Although reusing data in multiple levels of cache is important, past works have assumed 2D tiling and determine tile sizes that target data reuse in a single level of cache, generally the L1 cache. If data reuse in caches that are lower in the memory hierarchy (such as L2, L3 caches) must be considered, the impact of set associativity assumes greater significance, as those caches usually have higher set associativity than the L1 cache. Hence, ignoring set associativity will lead to unoptimal cache utilization and limited performance gains.

Figure 2 substantiates the various arguments made in this section to motivate our work. Figure 2 plots three different metrics, execution time, L2 data cache misses, and L1 data cache misses for the *matmul* benchmark tiled in three dimensions with nine different tile sizes. The experiments were all performed on an Intel Xeon processor (E5-2650, 2.0GHz) with 32KB private L1 cache, 256KB private L2 cache, and a 20MB L3 cache shared by eight Sandy Bridge cores. In each tiled code, the outermost dimension was fixed at 128 to illustrate the performance impact of L2 cache misses. For each metric, the figure plots values normalized within the range 0...1. Normalized values

Fig. 2. Normalized metrics for *matmul* ($N = 2,000$), L1 size = 32KB, and L2 size = 256KB.

for each metric are computed using the standard min-max normalization as the ratio, $(x - \min) / (\max - \min)$, where x , $\min$, and $\max$ are the absolute, minimum, and the maximum values, respectively, of that particular metric. Each tiled code uses values of J and K such that the tile maximally occupies the L1 cache without incurring conflict misses. The metric *L1_DCM** shows the L1 cache misses for tile sizes that are greater than the nine sizes shown on the x -axis by just one cache line in the K dimension. From the figure, the following messages can be taken.

- (1) The working set for each of the nine maximal tiled codes is nearly 24 KB,² which is well within the L1 cache size of 32KB. A small increase in tile size leads to a significant increase in the L1 cache misses, as shown by the metric *L1_DCM** in the figure. This increase in the L1 cache misses is attributed to the conflicts that occur in the L1 cache, even though the cache capacity is not reached. The increase in L1 cache conflict misses is reflected in the increased execution time as compared to that of the maximal tiles, as shown in the figure through the metric *Ex. Time**. Thus, it is critical to consider the impact of set associativity when determining the optimal tile size.
- (2) With the L1 misses held nearly constant for the nine maximal tiled codes as shown by the metric *L1_DCM* in the figure, the execution time follows the pattern of L2 cache misses, with a sharp increase in the execution time beyond $J = 152$. This is because, beyond this point, the working set comprising the data accessed within each iteration of the kT loop overflows the L2 cache, leading to loss of reuse of data accessed by reference C . Thus, it is critical to choose a tile size that targets reuse of data in the L2 cache in addition to the L1 cache. A similar analysis was used to motivate the need to consider data reuse in multiple levels of memory hierarchy in Shirako et al. [2012].

The algorithm presented in this article emulates execution time behavior of references with reuse in a particular level of cache by considering the cache size, line size, and set associativity. The algorithm considers data reuse in multiple levels of cache to provide an estimate of the optimal tile size. The details of our algorithm are presented in Sections 4 and 5.

²The size of the working set in the L1 cache, as derived earlier in Section 2.1, is given by $(K * J + 2 * J + K + 1)$.

3. BACKGROUND

In this section, we briefly describe the terminology and concepts that are key to understanding the problem of tile size selection and our proposed framework.

It is important to first understand the difference between data reuse and data locality. *Data reuse* is reusing the same data during a program execution in multiple iterations of a loop nest. However, the data may not stay in the cache after the first use. If a data item manages to stay in the cache between successive uses, there is said to be *data locality*. There are two types of data reuse that are especially relevant to tile size selection: self-temporal reuse and self-spatial reuse.

Self-Temporal Reuse. This happens when the same reference reuses a data item in distinct loop iterations. For example, the array reference $B[k][j]$ in Figure 1(b) and 1(c) reuses the same data items in every iteration of loop i and thus can be said to have *self-temporal reuse* in loop i . Similarly, array references $A[i][k]$ and $C[i][j]$ in the same figures have self-temporal reuse in loops j and k , respectively. However, array reference $B[k][j]$ reuses an entire tile in each iteration of the outermost loop i , array reference $C[i][j]$ reuses a tile row in each iteration of loop k , and the array reference $A[i][k]$ reuses only an element in each iteration of loop j .

Thus, only array references with self-temporal reuse in the outermost loop such as $B[k][j]$ provide opportunity for considerable data reuse. It is only in the presence of such references in the source code that loop tiling gives a significant benefit. Our algorithm tracks such references and focuses on minimizing conflict misses caused by such references.

Self-Spatial Reuse. This happens when a reference reuses the same cache line in distinct loop iterations. For example, the array references $C[i][j]$, $A[i][k]$, and $B[k][j]$ in Figure 1(c) all reuse the same cache lines for multiple successive iterations of loops j , k , and j , respectively, and can be thus said to have *self-spatial reuse* in the respective dimensions. It is for this reason that self-spatial reuse is best availed when the tile dimensions are a multiple of the cache line size.

Data reuse is enabled by data locality—that is, data is reused if the data is not replaced from the cache before subsequent use. Thus, corresponding to the two types of data reuse, there are two types of data locality: *self-temporal* and *self-spatial* locality. Loop tiling improves the self-temporal locality of data by reducing its reuse distance. Since the reuse distance is a function of the tile size, tile size should be chosen such that data items accessed within a tile are not replaced from the cache due to capacity or conflict misses.

In the tiled code, capacity misses can be easily avoided by choosing a tile whose working set size is smaller than the cache capacity. This, however, does not avoid conflict misses that result from noncontiguous memory accesses within the tile. The conflict (or interference) misses are of two types: *self-* and *cross-interference* misses. Self-interference misses occur when a reference with self-temporal reuse accesses multiple data items that collide in the cache—that is, they are mapped to the same set. For example, in Figure 1(c), self-interference occurs when data items accessed by the reference $B[k][j]$ collide. This is demonstrated in Figure 3(a), which shows the snapshot of a 32KB four-way set associative L1 cache with 16-element cache lines, at the instance when self-interference begins to cause misses during the execution of an array tile. Self-interference is pronounced for problem sizes³ that are a power of 2. This is because the cache size is also a power of 2. In such a case, different rows in the tile are prone to map to the same set. This is seen in Figure 3(a), where the leading dimension

³Problem size refers to the size of the array or matrix in a program. However, in this article, we mention problem size to particularly refer to the leading dimension of the array, which is critical for tile size selection. For row-major 2D arrays as in C, elements from one row to the next are a *leading dimension* apart.

Fig. 3. (a) Demonstration of self-interference ($ld_B = 512$). (b) Demonstration of cross-interference ($ld_A = 2,000$). ld_A and ld_B are leading dimensions of arrays A and B, respectively.

of array B (ld_B) is 512—the 17th row of the tile interferes with the first row of the tile. Thus, for the given cache configuration and problem size, the tile height, or number of rows in the tile, should not exceed 16 or else conflict misses will result. In this example, the tile width was chosen to be 16, the size of the cache line, but the case of pronounced conflict misses depends primarily on the tile height.

Cross-interference misses occur when two or more references collide in the cache and one of them has self-temporal reuse. For example, in the *jacobi-2d* kernel shown in Figure 1(d), cross-interference occurs when data items accessed by the references to arrays A and B collide. Figure 3(b) shows a snapshot of the cache demonstrating cross-interference between references to arrays A and B, where the leading dimension of the arrays is 2,000. In the figure, we assume that references A[0][0] and B[0][0] map to sets 0 and 6, respectively. However, runtime addresses of the arrays are not known at compile time and thus cross-interference is harder to account for. Among works based on an analytical model, Coleman and McKinley [1995] and Chame and Moon [1999] have attempted to minimize cross-interference misses, but only for direct-mapped caches. Our algorithm in this work (TSS) solves this problem by leveraging the high set associativity in modern caches, leading to consistent near-optimal performance.

4. OUR APPROACH

In this section, we discuss the various factors that influence tile size and our approach for accommodating them in our algorithm. For this purpose, we first consider linear algebra and data mining kernels that have considerable reuse and thus benefit from tiling. The factors affecting tile size selection in such kernels are considered in Sections 4.1 through 4.4 through the example of *matmul* introduced in Section 2. The conclusions presented are, however, applicable to a variety of kernel programs that allow data reuse and thus benefit significantly from the tiling transformation. Although the example of *matmul* that has a 3D loop nest is considered, the approach is applicable to nD loop nests as described in Section 5.1. It must be noted that we assume single-level nD tiling (i.e., n intertile loops and n intratile loops) for a source code that has n loops and still achieve data reuse in multiple levels of cache—the array reference(s) that has reuse in the outermost intratile loop is reused in the L1 cache, and the array

reference(s) that has reuse in the innermost intertile loop is reused in the L2 cache. In other words, we do not consider multilevel tiling to achieve data reuse in multiple levels of cache.

Apart from linear algebra and data mining kernels, stencils are another class of codes that benefit significantly from tiling. In stencils, tiling is performed after loop skewing and is called *time skewing* [Wonnacott 2002]. The factors affecting tile size selection in stencils are considered separately in Section 4.5.

4.1. Data Reuse in the L1 Cache

Data reuse in the L1 cache—that is, the fastest among all levels of cache—is critical to performance of a tiled code. For the 3D-tiled *matmul* shown in Figure 1(c), data reuse in the L1 cache is achieved by reusing the data accessed within a tile of array *B* in every iteration of the outermost intratile loop *i*. This requires the tile dimensions (*K*, *J*) to be chosen such that data locality for array reference *B* is ensured between successive iterations of loop *i*. We define two metrics to analyze the impact of tile size selection on data reuse:

(1) *Total Cache Misses (TCM)*: The total number of misses incurred in a particular level of cache for the entire execution of the program.

(2) *Reuse Ratio (RR)*: The ratio of the amount of reusable data to the total working set size for a particular level of cache.

TCM for the L1 cache as a function of the tile dimensions (*I*, *J*, *K*) is calculated as follows.

The number of cold misses incurred in the first iteration of loop *i* equals the number of cache lines accessed in each iteration of loop *i*—that is, a tile (*J* * *K* elements) of array *B*, and a tile row for each of arrays *A* and *C* (*K* and *J* elements, respectively). It is given by

$$K * \left\lceil \frac{J}{CLS} \right\rceil + \left\lceil \frac{K}{CLS} \right\rceil + \left\lceil \frac{J}{CLS} \right\rceil, \quad (1)$$

where *CLS* is the cache line size.

Since our goal is to reuse a tile of array *B* in each iteration of loop *i*, we assume that the tile size is such that there are no capacity or conflict misses that cause the data accessed within a tile of array *B* to be evicted from the cache. Thus, the total number of cache misses for the *I* iterations of loop *i* equals

$$K * \left\lceil \frac{J}{CLS} \right\rceil + I * \left\lceil \frac{K}{CLS} \right\rceil + I * \left\lceil \frac{J}{CLS} \right\rceil \quad (2)$$

Thus, TCM for the L1 cache equals

$$\left(K * \left\lceil \frac{J}{CLS} \right\rceil + I * \left\lceil \frac{K}{CLS} \right\rceil + I * \left\lceil \frac{J}{CLS} \right\rceil \right) * \frac{N^3}{I * J * K}, \quad (3)$$

where $\frac{N^3}{I * J * K}$ are the number of loop iterations in the intertile loops. This, when simplified (assuming *I*, *J*, and *K* are multiples of *CLS*), becomes

$$\left(\frac{1}{I} + \frac{1}{J} + \frac{1}{K} \right) * \frac{N^3}{CLS} \quad (4)$$

RR for the L1 cache is defined as the ratio of the reusable data (i.e., *J* * *K* elements) to the total working set size at the L1 cache. The total working set size at the L1 cache can be calculated as the sum of *Minimum Working Set Lines* [Shirako et al. 2012] and

the correction for the cache replacement policy (as discussed in Section 2). It is given by

$$K * J + 2 * J + K + 1 \quad (5)$$

Thus, RR for the L1 cache is

$$\frac{K * J}{K * J + 2 * J + K + 1} \quad (6)$$

From Equations (4) and (6), two inferences can be drawn:

- (1) Each of the tile dimensions should be as large as possible for effective data reuse and minimization of cache misses.
- (2) RR for the L1 cache is close to 1. This implies that the array reference B dominates the working set and would occupy most of the ways in a set-associative cache.

4.2. Data Reuse in the L2 Cache

Data reuse in the L2 cache is critical, as L2 misses are more costly than L1 misses. Data reuse in the L2 cache is achieved by reusing the data accessed within a tile of array C in every iteration of the innermost intertile loop kT . We again consider the TCM and RR for the L2 cache to analyze the impact of tile size selection on data reuse. TCM for L2 cache as a function of the tile dimensions can be calculated as follows.

The number of cold misses incurred in the first iteration of loop kT equals the number of cache lines accessed in each iteration of loop kT —that is, a tile each of arrays A , B , and C ($I * K$, $K * J$, and $I * J$ elements, respectively).

$$I * \left\lceil \frac{K}{CLS} \right\rceil + K * \left\lceil \frac{J}{CLS} \right\rceil + I * \left\lceil \frac{J}{CLS} \right\rceil \quad (7)$$

Since our goal is to reuse a tile of array C in every iteration of loop kT , we assume that the tile size is chosen such that there are no capacity or conflict misses that cause the data accessed within a tile of array C to be evicted from the cache. Thus, the total number of cache misses for the $\frac{N}{K}$ iterations of loop kT equals

$$I * \left\lceil \frac{K}{CLS} \right\rceil * \frac{N}{K} + K * \left\lceil \frac{J}{CLS} \right\rceil * \frac{N}{K} + I * \left\lceil \frac{J}{CLS} \right\rceil, \quad (8)$$

which when simplified becomes

$$\frac{I * N + J * N + I * J}{CLS} \quad (9)$$

Thus, TCM for the L2 cache equals

$$\frac{I * N + J * N + I * J}{CLS} * \frac{N^2}{I * J}, \quad (10)$$

which when simplified becomes

$$\left(\frac{1}{J} + \frac{1}{I} + \frac{1}{N} \right) * \frac{N^3}{CLS} \quad (11)$$

RR for the L2 cache is defined as the ratio of the reusable data (i.e., $I * J$ elements of array C) to the total working set size at the L2 cache. The total working set size at the L2 cache can be calculated as the sum of number of distinct lines [Ferrante et al. 1992] accessed within the intratile loops and the correction for the cache replacement

Fig. 4. Choosing a tile size that exploits data reuse in both the L1 and L2 cache.

policy. This allows L2 reuse in the innermost intertile loop. The working set at the L2 cache is given by

$$(I + 1) * K + 2 * K * J + I * J \quad (12)$$

Thus, RR for the L2 cache is

$$\frac{I * J}{(I + 1) * K + 2 * K * J + I * J} \quad (13)$$

From Equations 11 and 13, two inferences can be drawn:

- (1) The tile dimensions I and J should be as large as possible for effective data reuse and minimization of L2 cache misses.
- (2) Unlike L1 cache, the value of RR for the L2 cache cannot be easily estimated.

4.3. Data Reuse in Both the L1 and L2 Cache

The TSS algorithm presented in this article emulates execution-time behavior of a single reference in a particular level of cache by considering the size of the cache, its line size, and associativity. The algorithm thus aims to minimize conflict misses at that level of cache to enable effective data reuse. For the L1 cache, clearly, the reference(s) with temporal reuse in the outermost intratile loop, such as the array reference B in *matmul*, dominates the working set and occupies most ways in a set-associative cache. Thus, the TSS algorithm (described later in Section 5) works on the assumption that the array reference B occupies all but one way of the L1 cache and returns a list of (K, J) tuples, all of which correspond to maximal tiles—that is, tiles whose sizes can be increased no further without incurring conflict misses. The best (K, J) tuple and the size of the I dimension are chosen by considering further criteria.

We concluded from Equation (11) that tile dimensions I and J should both be large for effective data reuse in the L2 cache. Further, we observe from Figure 4 that the

product $J * K$ is nearly constant in each of the (K, J) tuples that minimize conflict misses in the L1 cache. Thus, for J to be large, a correspondingly smaller K must be chosen. With a large J (and a large I) and a small K , it can be concluded that the RR for the L2 cache as given by Equation (13) will also tend toward 1. This shows that the working set that should fit the L2 cache is also dominated by a single array reference, C , that has reuse in the innermost intertile loop, kT . In such a scenario, the TSS algorithm assumes that the array reference C occupies all but one way of the available ways in the L2 cache, and similarly yields a list of (I, J) tuples, all of which correspond to maximal tiles in L2 cache. From the list, the tuple (I, J) that minimizes $(\frac{1}{I} + \frac{1}{J})$ is selected in accord with Equation (11) (marked as step 1 in Figure 4). For the chosen J , a corresponding K is chosen from the list of tuples (K, J) obtained earlier (marked as step 2 in the figure). Thus, the three tile dimensions (I, J, K) are determined using the TSS algorithm that considers reuse at both the L1 and L2 cache (marked as step 3 in the figure). Although we have only considered reuse in the L1 and L2 caches in this article, the latest microarchitectures employ a shared L3 cache and a similar analysis can be extended to analyze reuse in the L3 cache.

4.4. Interaction with Vectorization

Short-vector SIMD instruction sets such as Altivec and SSE (and now AVX) have proved to be promising for enhancing performance on modern processors. It is thus critical to consider the interaction of tiling with the SIMD unit on the host architecture.

As a result of loop tiling, all loops including the vector loop are strip mined—that is, they are fragmented into smaller segments or strips. Although this helps data reuse, it reduces the length of the vector pipeline. Thus, strip mining the vector loop deprives the vector unit of the needed fodder for Instruction-Level Parallelism (ILP). This effect is more pronounced in the latest microarchitectures, as they have larger vector registers requiring longer loops and multiple vector load/store units that reveal more opportunity for ILP.

To make the best use of a bad bargain, the tile size should be so chosen that while data reuse is exploited, losses from a reduced vector pipeline length are minimized. This is ensured when the tile dimension corresponding to the vector loop is large, but only large enough that the tile does not cause interference misses. We evaluate this interplay between vectorization and data reuse in three different scenarios. These scenarios are shown in Figure 5, where loops j , i , and k are the vector loops in *matmul*, *trisolv*, and *strsm* benchmarks, respectively. In all three benchmarks, the *intertile* loops are in the same order (iT-jT-kT), but the order of *intratile* loops is different in each case and is decided by the PLuTo compiler such that efficient vectorization could be achieved. The PLuTo compiler applies the loop interchange transformation within the polyhedral framework to achieve this, and as a result, any of the three loops could be made the innermost (vector) loop. We thus consider all three possible scenarios.

According to Equation (13), tile dimensions I and J should be large for effective data reuse in the L2 cache. This further implies that the tile dimension K should be small for effective data reuse in the L1 cache. Thus, in *matmul* and *trisolv* benchmarks, where loops j and i are the vector loops, respectively, the goals of achieving efficient vectorization and data reuse are concomitant. However, this is not the case for the *strsm* benchmark, where loop k is the vector loop. Choosing a small value of K would severely hurt vectorization. However, although choosing a large value of K prevents data reuse in the L2 cache, data reuse within the L1 cache can still be achieved in addition to effective vectorization. Empirical verification supports that in such cases, 2D tiling that aims at data reuse in only the L1 cache and effective vectorization gives

<pre> for(i = 0; i < N; i++) for(j = 0; j < N; j++) for(k = 0; k < N; k++) C[i][j] += A[i][k] * B[k][j]; </pre> <p>Matmul benchmark</p> <pre> for (iT=0; iT< N/I; iT++) for (jT=0; jT< N/J; jT++) for (kT=0; kT< N/K; kT++) for (i=iT; i<I*iT+I-1;i++) for (k=K*kT; k<K*kT+K-1;k++) /* vector loop */ for (j=jT; j<J*jT+J-1;j++) C[i][j] += A[i][k] * B[k][j]; </pre> <p>Tiled matmul</p> <p>(a)</p>	<pre> for(i = 0; i < N; i++) for(j = 0; j < N; j++) for(k = 0; k < j; k++) B[j][i] = B[j][i] - L[j][k]*B[k][i]; </pre> <p>Trisolv benchmark</p> <pre> for (iT=0; iT< N/I; iT++) for (jT=0; jT< N/J; jT++) for (kT=0; kT< N/K; kT++) { for (j=jT; j<J*jT+J-1;j++) for (k=K*kT; k<K*kT+K-1;k++) /* vector loop */ for (i=iT; i<I*iT+I-1;i++) B[j][i] = B[j][i] - L[j][k]*B[k][i]; } </pre> <p>Tiled trisolv</p> <p>(b)</p>	<pre> for(i = 0; i < N; i++) for(j = 0; j < N; j++) for(k = i+1; k < N; k++) if(k==i+1) B[j][i]/=A[i][i]; B[j][k] -= A[i][k]*B[j][i]; </pre> <p>Strsm benchmark</p> <pre> for (iT=0; iT< N/I; iT++) for (jT=0; jT< N/J; jT++) for (kT=0; kT< N/K; kT++) { for (i=iT; i<I*iT+I-1;i++) for (j=jT; j<J*jT+J-1;j++) /* vector loop */ for (k=K*kT; k<K*kT+K-1;k++) B[i][k] -= A[j][k]*B[i][j]; } </pre> <p>Tiled strsm</p> <p>(c)</p>
---	---	--

Fig. 5. Vectorization, reuse, and tile size.

the best performance. This is incorporated in our framework by choosing the largest *rectangular* tile from the list of tile size tuples returned by our algorithm for the L1 cache; the outermost tile dimension is effectively left untiled by choosing its size to be the same as the problem size.

In addition, since the TSS algorithm chooses each tile dimension to be a multiple of the cache line size (for achieving spatial locality), the tile dimension corresponding to the vector loop is ensured to be a multiple of the vector size. This leads to effective vectorization.

4.5. Interaction with Time Skewing

For stencil codes such as *jacobi-2d* (shown earlier in Figure 1(d)), *time skewing* can significantly improve temporal locality. For time-skewed codes, however, estimating the optimal tile size is more involved because of the *shifting tiles*, as shown in Figure 6—the tile space shifts by the corresponding skewing factors (S_H and S_W in the case of 2D tiles) in each dimension for each iteration of the tiled time loop (loop t in Figure 1(e)). This leads to the following essential differences in the case of time-skewed codes:

- (1) In non-time-skewed codes, adjacent tiles access distinct data (except for some possible overlap at tile boundary). However, as a result of *shifting* tiles in time-skewed codes, there is an opportunity for reusing common data between adjacent tiles. For example, $tile_{i,j}$ and $tile_{i,j+1}$ access common data for $t = 1$, shown by the darkly shaded region in Figure 6. Thus, if the data accessed by $tile_{i,j}$ in all T (length of the tiled time loop, t) time steps fits the cache, the common data can be reused by $tile_{i,j+1}$ in all T iterations. In such a scenario, the only misses incurred by $tile_{i,j+1}$ for $t = 1$ (or for $t = k$, in general) are due to the new data accessed, as shown by the unshaded region in $tile_{i,j+1,t=1}$ in the figure. Thus, the number of misses are given by $\frac{W \cdot S_H}{CLS}$, where CLS is the cache line size in terms of the number of elements. The total misses incurred in timesteps 1 through T are given as $\frac{W \cdot S_H \cdot T}{CLS}$. Similarly, the total misses incurred in the program that has a total of $\frac{N}{W} \times \frac{N}{H}$ tiles and that iterates

Fig. 6. Illustration of skewed-tile traversal.

for T_0 timesteps is given as

$$\frac{S_H * T_0 * N^2}{CLS * H} \quad (14)$$

Since the TCM are inversely related to the tile height, the TSS algorithm chooses a tile with the largest height such that the tile width is equal to a single cache line from the list of tile size tuples that minimize interference in the L1 cache. This ensures that $\text{tile}_{i,j}$ for any $t(=k)$ fits the L1 cache. However, to enable intertile data reuse in the L2 cache, the data accessed by $\text{tile}_{i,j}$ in timesteps 1 through T should fit the L2 cache. In addition, more data is brought into the cache by $\text{tile}_{i,j+1}$, with the possibility of interference in the L2 cache. As can be seen from the figure, any two consecutive tiles, such as $\text{tile}_{i,j}$ and $\text{tile}_{i,j+1}$, access data within a block of dimensions $(2*W + T*S_W) \times (H + T*S_H)$ for all T iterations of the tiled time loop. Thus, the TSS algorithm assumes the worst case scenario and chooses T such that this block fits in the L2 cache. This also minimizes interference in the L2 cache. With W and H known, the value of the tile in the time dimension, T , is thus calculated, and the tile chosen achieves reuse both in the L1 and L2 caches.

- (2) The previous discussion favors tiles with smaller widths to achieve data reuse in the L2 cache for stencil codes. This, however, is unproductive for efficient vectorization, as explained. Thus, if a stencil code is vectorizable in the innermost loop, the two factors conflict with each other. Empirical results show that for stencil codes that are not vectorizable because of dependences, such as *seidel*, smaller tile widths perform better, and vice-versa for vectorizable stencils such as *jacobi* and *fdtd*. Thus, for vectorizable stencils, we choose tiles with larger widths (such that the tile height is no less than a cache line) in favor of effective vectorization. In such cases, since intertile data reuse in the L2 cache is not achieved, the height and width of the tile are chosen such that a single tile fits the L2 cache and the time dimension is chosen to be the total number of timesteps.

4.6. Other Factors That Influence Tiling

Interaction with Multilevel TLB. Hsu and Kremer [2004] argue that in addition to data reuse in the L1 cache, the tile size chosen must also minimize TLB misses. Their rationale was based on the assumption that TLB misses are much more costly than cache misses (and also possibly quite frequent given a small TLB). However, the recent architectures employ a two-level TLB. For example, Intel's Sandy Bridge has a 64-entry L1 TLB and a 512 entry L2 TLB, whereas there is just a single level 64-entry TLB in the Intel's Netburst microarchitecture. On Sandy Bridge that uses a four-level page table, the cost of a page walk is also reduced to merely 30 cycles, which is same as the latency for an L2 cache miss. In addition, tiling compilers such as PLuTo (used for this work) permute the intratile loops such that most references achieve spatial locality and effective vectorization in the innermost loop. This also minimizes the TLB misses, because for such references, consecutive elements reside on the same page. It is for these reasons that empirical results reveal minimal impact of TLB misses on tile size selection.

Interaction with Shared Caches. CMP and SMT are the two techniques currently employed to extract the inherent ILP in programs run in a *multithreaded* environment. In a multithreaded environment, multiple cores may bring in different data to the shared cache on a chip multiprocessor, and similarly, multiple threads may bring in different data to the private cache on a core using the SMT technology. We account for this in our algorithm by adjusting the set associativity to $(\frac{1}{T})$ of the actual value, where T is the number of threads/cores that share the cache. This is done to prevent any cache conflicts even for the worst case when all T threads/cores execute different tiles and thus bring different data into the cache.

Data Re-layout of the Tiles/To Copy or to Not Copy. Copy optimization, or copying, is a technique to adjust the data layout for reducing cache conflicts within a tile, wherein a tile is copied into a temporary *linear* array and copied back to its original memory location after execution. As a result, all elements within a tile are mapped to contiguous locations instead of disparate locations in the cache. This data re-layout improves the cache behavior by minimizing conflict misses [Temam et al. 1993]. If we thus re-layout the tiled arrays, the tile size is then largely governed by cache capacity and is easily estimated [Yotov et al. 2005]. However, production quality compilers such as GCC and ICC do not support such data re-layout because (1) the additional cost of copying may offset the savings, (2) there is a need to write complex clean-up code when the problem size is not a multiple of the tile size, and (3) the complexity of the transformation itself—that is, it must be determined what to copy and when to copy. In addition, another important reason is that copying cannot be performed for stencil codes, as the tiles shift in every iteration of the tiled time loop.

The TSS algorithm avoids copying. It instead achieves the purpose of minimizing interference by analyzing the source program and its interaction with the architecture and the compiler. It thus saves the overhead of copying and relieves the compiler from the burden of performing the complex copy optimization.

Problem Size and Array Padding. As discussed earlier in Section 3, conflict misses are pronounced for problem sizes that are a power of 2. Such problem sizes are called *pathological* problem sizes, as the various algorithms employed for tile size selection yield either too skinny or too fat tile sizes, resulting in suboptimal performance. Array padding is a technique proposed in previous works [Rivera and Tseng 1999; Panda et al. 1999] to handle such cases, where the problem size (particularly the leading dimension) of the arrays is increased to prevent conflicts in the cache and achieve reasonable performance even for the pathological problem sizes. As for copying, compilers such as GCC and ICC do not support padding because (1) it transforms the array structure that must be reflected in the entire code, and (2) it cannot be directly incorporated

into a library routine, as the problem size is not known a priori; copying the arrays into another array with padding helps, but copying has its own disadvantages and overheads.

5. THE TILE SIZE SELECTION ALGORITHM

As described in Sections 4.1 and 4.2, only references such as the array reference B for the L1 cache (and reference C for the L2 cache), in the example of *matmul*, access reusable data and dominate the working set in the respective caches. For a given source program and cache parameters, the TSS algorithm emulates cache behavior for one such array reference (say, k) during tile execution. For the emulation, the reference k is allowed to occupy $n - 1$ ways of an n -way set-associative cache, whereas the remaining references that have small contributions to the working set are assumed to occupy the remaining one way of the cache. If there are K such array references, cross-interference among them is prevented by assuming the effective set associativity of the cache to be $n_e = \lfloor \frac{n}{K} - 1 \rfloor$ instead of the actual value of n . This emulation is done for different values

ALGORITHM 1: Tile Size Selection

INPUT:

Size of cache: $cSize$

Set associativity of cache: n

Size of cache line: $lSize$

Leading dimension, or Problem-size: N

No. of arrays with self-temporal reuse in outermost loop: K

No. of active threads per core: T

No. of elements in a cache line: $CLS \left(= \frac{lSize}{size(DataType)} \right)$

No. of sets in cache: $nSets \left(= \frac{cSize}{lSize * n} \right)$

Effective set associativity: $n_e \left(= \lfloor \frac{n}{K * T} - 1 \rfloor \right)$

Step 1:

for $w = 1$ to $\min \left(\frac{cSize}{lSize}, \frac{N}{lSize} \right)$ **do** { w is tile width in number of cache lines */}

$r = 0$

repeat { r iterates over rows in a tile */}

$nset \leftarrow \left(\frac{r * N}{CLS} \right) \% nSets$

for $c = 0$ to w **do** { c brings a tile row into cache */}

if $cache_emu[nset + c] = n_e$ **then**

$h = r$ { r tile height is maximum rows in tile */}

break { c interference begins at this point */}

else

$cache_emu[nset + c] ++$

end if

end for

$r ++$

until $r = N$

if $h < CLS$ **then**

Add the tile size $(h, w * CLS)$ to list L

else

Add the tile size $(\lfloor \frac{h}{CLS} \rfloor * CLS, w * CLS)$ to list L

end if

end for

OUTPUT: List of tile size estimates, L

of the tile width⁴ that are multiples of the cache line size, and the algorithm returns in each case a corresponding tile height at which cache conflicts ensue. Thus, a list, L , of tile sizes that aim to minimize interference and maximize cache usage is generated. The same algorithm is used to determine the list of tile size tuples (K, J) and (I, J) for minimizing conflicts in the L1 and L2 cache, respectively. The estimate of the best tile size (I, J, K) is determined by further criteria as described earlier in Section 4.3.

The algorithm emulates the fetching of cache lines into the cache for the array reference k . The emulation is done using *cache_emu*, an array whose every element stores the count of the number of occupied ways in a set in the cache. The elements of *cache_emu* are initialized to a count of 0, indicating that no way of the set is occupied. The count (of a set) is increased to emulate the fetching of a cache line into the set. A row in the tile maps to a set given by $nset$, as shown in the algorithm. Since the memory addresses wrap around the cache, our algorithm can safely assume that the first element of an array maps to the first line (set) of the cache. Henceforth, for a chosen width of w cache lines in a tile row, the algorithm brings the lines comprising a tile row into the cache. The cache is thus filled row-wise until all available ways (given by n_e) in a set of the cache are filled, and interference begins to replace reusable data in the cache. Thus, the tile height ($h = r$, the number of rows in tile at this point) for the chosen tile width is obtained such that there are no interference misses. The outermost for loop over w thus generates a list of tile sizes, L . Each tile size is the tuple $(h, w * CLS)$ for a chosen value of w . The tile height in each tuple is truncated to the nearest multiple of the cache line size for efficient utilization of the spatial reuse.

5.1. Scope of the Algorithm

This section describes the nature of programs that can benefit from the TSS algorithm. First, it is important to note that programs containing references that carry reuse in one of the loops in the loop nest can alone benefit significantly from the loop tiling transformation, and from a good tile size chosen by the TSS algorithm.

The TSS algorithm relies on exploiting reuse at two loop levels—the outermost intratile loop and the innermost intertile loop. Thus, for a program to benefit from TSS, it must contain an array reference that has reuse in the outermost intratile loop and one that has reuse in the innermost intertile loop. The TSS algorithm gives a list of good tile dimensions corresponding to the inner loops based on considering intratile reuse, but the size of the outermost tile dimension must be obtained after considering the intertile reuse. The intertile reuse, however, depends significantly on the loop order—if the innermost intertile and innermost intratile loops correspond to the same loop in the original program as in the *strsm* kernel, intertile reuse is sacrificed in favor of intratile reuse and effective vectorization. In such a case, the outermost intratile loop is left untiled. In all other cases, effective intertile reuse can be achieved without hurting vectorization by using the cost function obtained from Equation (11). This strategy proves applicable for kernels that yield nonskewed rectangular tiles, such as those in basic linear algebra (BLAS), data mining, and image processing applications.

In stencils, the inner space loops are skewed with respect to the outermost time loop. Thus, whereas intratile reuse can be achieved in a manner similar to the nonskewed tiled codes, intertile reuse is achieved in a different way, as explained in Section 4.5.

TSS Algorithm for Imperfectly Nested Loop Nests. An important point to note is that when imperfectly nested loops are tiled, statements are distributed into different nests at one of the outer intertile loops. As a result, the cost functions derived in this work

⁴Here, the algorithm is presented for the case of 2D tiles for understandability and is extended for the general case of nD tiles in Section 5.1. Width in general represents the size of the tile in leading dimension.

Fig. 7. Our framework.

for the intratile loops, as well as the innermost intratile loop, can still be applied to such codes. However, in such cases, multiple imperfectly nested loop nests will have to use the same tile size, which may not be good for all nests.

TSS Algorithm for Programs with Many Array References. The TSS algorithm depends on the effective set associativity, computed as $n_e = \lfloor \frac{n}{K*T} - 1 \rfloor$. Thus, if the number of array references in the source program (K) is large (or, the number of threads sharing the cache, T , is large), then all array references cannot be assigned a unique way, given the limited number of ways in the cache. In such a case, we improvise the TSS algorithm by doubling the set associativity (n) and halving the number of available sets. Although this does not guarantee conflict minimization due to the overlapping of data accessed by multiple references in the cache, it nonetheless serves as a good approximation for two reasons: (1) with the cache size remaining constant, it still partially emulates cache conflicts and does not fill the cache to capacity, and (2) since the number of sets remain a power of 2, it still accounts for the case of pronounced cache misses for certain pathological problem sizes.

TSS Algorithm for nD Tiles and Nonsquare Problem Sizes. The TCM/RR analysis and the TSS algorithm presented thus far assume 3D loop nests and 2D tiles. Thus, we refer to tile dimensions, I , J , and K , and to the “height” and “width” of a tile. However, in the general case of nD tiles, the tile size corresponding to the outermost intratile loop (I) represents the product of multiple outer intratile loops. Thus, the same analysis gives information about the relative impact of each tile dimension on data reuse for an nD tile. Similarly, the TSS algorithm is easily extended for nD tiles—tile width corresponds to the tile size in the leading dimension and tile height is split into multiple dimensions, and the mapping of a cache line to a particular set in the cache is accordingly computed. Thus, a list of tile size tuples, $(t_1, \dots, t_n)$, is obtained, and one of them is chosen as an estimate of the optimal tile size based on criteria determined from the TCM/RR analysis, exactly as in the case of 2D tiles. This is shown in Section 7 through the example of a tensor contraction kernel, *doitgen*, that involves 3D tiles and is tiled in four dimensions.

In addition, the TSS algorithm only depends on the leading dimension of the array reference with self-temporal reuse in the innermost loop. It is thus not restricted to *square* (*cubic*) problem sizes but will find an estimate of optimal tile size for any problem size.

Table I. Details of the Microarchitectures

Microarchitecture	Cache Size (L1 L2 L3)	Cache Type (L1 L2 L3)	Set Associativity (L1 L2 L3)
Sandy Bridge	(32KB 256KB 12.5MB)	(priv. priv. shared)	(8 8 20)
Core	(32KB 1.25MB —)	(priv. shared —)	(8 8 —)

5.2. The Framework

The TSS algorithm is implemented within P_{Lu}To, a source-to-source transformation system based on the polyhedral model. The complete framework of our implementation is shown in Figure 7.

The P_{Lu}To transformation framework takes as input the original source code and generates *statement-wise affine transformations*. These statement-wise transformations are representative of a composition of loop transformations, including *loop tiling*. However, in the absence of a user-specified tile size, P_{Lu}To uses a default cubic tile size of 32 to generate the updated (statement-wise) transformations through its *Polyhedral Tile Specifier*. Our algorithm, instead, interacts with P_{Lu}To at this step to provide it with its estimate of the optimal tile size. The updated transformations with the tile size information are input to CLooG [Bastoul 2004] to generate transformed source code, which can then be compiled on the target machine using any backend compiler such as GCC or ICC.

The TSS algorithm needs three architectural parameters (cache size, line size, and set associativity) and three program specific parameters (the order of intratile loops, *K*—that is, the number of arrays with self-temporal reuse in a particular loop, and the skewing factors). The cache parameters can be determined through the OS (e.g., by a kernel call to *GetLogicalProcessorInformation* function in Windows or through specific files in `/sys/devices/system/cpu/cpu0/cache/` in Linux). They can also be obtained using microbenchmarks such as those in ATLAS. In our implementation, we used the former option to determine the cache parameters. The information about the program-specific parameters is obtained from the P_{Lu}To transformation framework. The order of the intratile loops is particularly needed, as P_{Lu}To might interchange the loops to favor vectorization, and this information is needed to determine the size of the tile in each dimension, as explained in Section 4.6. P_{Lu}To also provides information about the skewing factors and loop vectorization for codes that are time skewed to determine the tile size for such codes, as discussed in Section 4.5.

6. EXPERIMENTAL SETUP

We tested the effectiveness of our algorithm on an Intel Xeon E5-2650 processor with 8 cores (running at 2GHz) based on the Sandy Bridge microarchitecture, and an Intel Core2 E6400 processor with 2 cores (running at 2.13GHz) based on the Core microarchitecture. The details of each microarchitecture is given in Table I.⁵ For our experiments, we used 12 kernel benchmark programs from P_{Lu}To and PolyBench [Pouchet 2013] that are listed in Table II.

Of these 12 kernel benchmarks, the first 8 are various linear algebra kernels and solvers, the next two (*corcol* and *covcol*) are kernels employed in data mining programs, and the last 2 (*jacobi* and *seidel*) are stencil kernels. Since the tile size varies with the problem size, we tested the algorithm for two different problems sizes—one that is a power of 2, and the other that is not. The problem sizes that are a power of 2 are termed as *pathological*, because they cause pronounced conflict misses for certain tile sizes, whereas all other problem sizes do not demonstrate such a behavior. It is for this reason that we chose a problem size in each of these two categories. In addition, since the tile

⁵In the table, we list the effective L2 cache size as presented in Cooper and Sandoval [2011].

Table II. Summary of the Benchmarks

Kernel	Description	Problem Size	Data Type
matmul	Matrix Multiplication	N1=512; N2=2000	Double
dsyrk	Symmetric rank-k operations	N1=512; N2=2000	Double
dsyr2k	Symmetric rank-2k operations	N1=512; N2=2000	Double
lu	Lower Upper Decomposition	N1=512; N2=2000	Double
trisolv	Multiple Triangular Solver	N1=512; N2=2000	Double
doitgen	Multiresolution analysis kernel (MADNESS)	N1=128; N2=150	Double
strsm	Linear Equation Solver	N1=512; N2=2000	Float
tmm	Triangular Matrix Product	N1=512; N2=2000	Float
corcol	Correlation Computation	N1=512; N2=2000	Float
covcol	Covariance Computation	N1=512; N2=2000	Float
jacobi-2d	2-D Jacobi stencil computation	T=128; N1=512; N2=2000	Float
seidel-2d	2-D Gauss Seidel stencil computation	T=128; N1=512; N2=2000	Float

Table III. Comparison of the Algorithms

Algorithm	Cost Function	Pad Size		Set Associativity	Data Reuse in L2
		single prec.	double prec.		
eucpad	$\min\left(\frac{1}{k} + \frac{1}{j}\right)$	6	7	Not considered	Not considered
TSS	$\min\left(\frac{1}{i} + \frac{1}{j}\right)$	-	-	Considered	Considered

size also depends on the data type of the problem arrays, we show the performance results of our algorithm for different benchmarks; some have single-precision data arrays, whereas the others have double-precision data arrays. All tiled codes used in the experiments were generated with P_{Lu}T_o (version 0.9.0) using the options ‘-tile’ and ‘-parallel’ (for extracting pipelined parallelism in time-skewed stencils). For all experiments, we used the Intel C Compiler (ICC v13.0.1 with ‘-O3’ optimization option) as the backend compiler to compile the transformed source programs. We used the ‘-parallel’ compiler option with ICC only to obtain results on the performance of the TSS algorithm on multiple cores.

7. EXPERIMENTAL RESULTS

Among past works that used an analytical model for tile size selection, most algorithms suffered from poor performance for the pathological problem sizes. As a result, later models such as *eucpad* [Rivera and Tseng 1999] mentioned in Table III use padding to overcome these pronounced conflict misses that result at such problem sizes. However, *eucpad* considers neither the impact of set associativity nor of data reuse in multiple levels of cache. The TSS algorithm, on the other hand, considers the impact of set associativity in generating the list of maximal tile sizes for both the L1 and L2 cache. The TSS algorithm further considers data reuse in the L1 and L2 cache to generate its estimate of the optimal tile size, as discussed in Section 4. We compare *eucpad* and TSS in Table IV.

The *eucpad* algorithm chooses nonconflicting tile widths using the Euclidean GCD algorithm [Coleman and McKinley 1995] and tile heights using a simple recurrence. For pathological problem sizes, it allows for a padding of zero to seven elements and chooses a tile size from a list of sizes obtained by considering the different padding options. The cost function employed by *eucpad* tends to choose a square tile from the list of nonconflicting tile sizes obtained. It must be noted that their cost function is precisely based on Equation 4, which minimizes misses at the L1 cache. However, since the impact of set associativity is ignored, the tile size chosen does not minimize

Table IV. Performance of the TSS Algorithm

Benchmark	Scheme	Sandy Bridge						Core					
		Problem Size = 2,000			Problem Size = 512			Problem Size = 2,000			Problem Size = 512		
		Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)
matmul	eucpad	40 × 88	3.20	64 × 48	3.07	40 × 88	2.81	40 × 88	2.81	64 × 48	1.51	64 × 48	1.51
	TSS	168 × 32 × 104	4.1	40 × 7 × 504	3.45	248 × 8 × 328	3.2	248 × 8 × 328	3.2	192 × 7 × 504	1.46	192 × 7 × 504	1.46
	Best Tile	176 × 32 × 96	4.1	144 × 48 × 176	3.42	128 × 16 × 128	3.22	128 × 16 × 128	3.22	144 × 64 × 16	1.78	144 × 64 × 16	1.78
dsyrk	eucpad	40 × 88	3.22	64 × 48	2.55	40 × 88	2.80	40 × 88	2.80	64 × 48	1.96	64 × 48	1.96
	TSS	168 × 32 × 104	3.72	40 × 7 × 504	3.35	248 × 8 × 328	3.19	248 × 8 × 328	3.19	192 × 7 × 504	2.46	192 × 7 × 504	2.46
	Best Tile	160 × 16 × 144	3.89	96 × 32 × 176	3.22	160 × 144 × 16	3.09	160 × 144 × 16	3.09	128 × 16 × 176	2.31	128 × 16 × 176	2.31
dsyr2k	eucpad	40 × 88	3.98	64 × 48	2.93	40 × 88	2.76	40 × 88	2.76	64 × 48	2.01	64 × 48	2.01
	TSS	120 × 8 × 88	5.11	16 × 4 × 504	3.96	248 × 8 × 136	3.72	248 × 8 × 136	3.72	96 × 4 × 504	2.16	96 × 4 × 504	2.16
	Best Tile	56 × 8 × 112	5.27	176 × 16 × 176	3.90	96 × 16 × 96	3.58	96 × 16 × 96	3.58	176 × 32 × 176	2.20	176 × 32 × 176	2.20
lu	eucpad	40 × 88	2.82	64 × 48	2.07	40 × 88	1.89	40 × 88	1.89	48 × 64	1.63	48 × 64	1.63
	TSS	168 × 32 × 104	2.87	40 × 7 × 504	2.79	248 × 8 × 328	2.30	248 × 8 × 328	2.30	192 × 7 × 504	2.01	192 × 7 × 504	2.01
	Best Tile	176 × 16 × 160	3.01	96 × 32 × 176	2.63	176 × 16 × 144	2.01	176 × 16 × 144	2.01	160 × 16 × 176	1.86	160 × 16 × 176	1.86
trisolv	eucpad	40 × 88	3.76	64 × 48	2.43	40 × 88	2.52	40 × 88	2.52	64 × 48	2.19	64 × 48	2.19
	TSS	168 × 32 × 104	3.74	40 × 7 × 504	3.43	248 × 8 × 328	2.66	248 × 8 × 328	2.66	192 × 7 × 504	2.48	192 × 7 × 504	2.48
	Best Tile	176 × 80 × 128	3.81	80 × 32 × 176	2.84	176 × 112 × 176	1.80	176 × 112 × 176	1.80	176 × 176 × 176	2.09	176 × 176 × 176	2.09
strsm	eucpad	80 × 64	1.52	80 × 64	1.49	80 × 64	1.18	80 × 64	1.18	80 × 64	1.17	80 × 64	1.17
	TSS _{3D} *	2000 × 16 × 320	1.68	512 × 14 × 496	4.63	2000 × 16 × 320	1.26	2000 × 16 × 320	1.26	512 × 14 × 496	1.96	512 × 14 × 496	1.96
	Best Tile	160 × 144 × 256	1.47	208 × 48 × 256	2.58	144 × 240 × 224	1.11	144 × 240 × 224	1.11	128 × 48 × 256	1.48	128 × 48 × 256	1.48
tmm	eucpad	80 × 64	5.74	80 × 64	4.27	80 × 64	4.49	80 × 64	4.49	80 × 64	3.72	80 × 64	3.72
	TSS	160 × 16 × 208	7.3	80 × 14 × 496	4.55	632 × 16 × 320	6.00	632 × 16 × 320	6.00	384 × 14 × 496	5.25	384 × 14 × 496	5.25
	Best Tile	192 × 16 × 256	7.62	160 × 16 × 256	4.02	256 × 16 × 256	5.28	256 × 16 × 256	5.28	48 × 16 × 256	4.70	48 × 16 × 256	4.70
corcol	eucpad	80 × 64	4.31	80 × 64	3.58	80 × 64	2.14	80 × 64	2.14	80 × 64	2.09	80 × 64	2.09
	TSS	160 × 16 × 208	6.16	80 × 14 × 496	4.68	632 × 16 × 320	2.69	632 × 16 × 320	2.69	384 × 14 × 496	2.46	384 × 14 × 496	2.46
	Best Tile	144 × 16 × 256	6.41	112 × 16 × 176	4.51	176 × 16 × 256	2.38	176 × 16 × 256	2.38	208 × 16 × 176	2.37	208 × 16 × 176	2.37
covcol	eucpad	80 × 64	4.35	80 × 64	3.69	80 × 64	2.18	80 × 64	2.18	80 × 64	2.17	80 × 64	2.17
	TSS	160 × 16 × 208	6.26	80 × 14 × 496	4.98	632 × 16 × 320	2.72	632 × 16 × 320	2.72	384 × 14 × 496	2.58	384 × 14 × 496	2.58
	Best Tile	112 × 16 × 256	6.51	96 × 16 × 176	4.77	224 × 16 × 256	2.41	224 × 16 × 256	2.41	112 × 16 × 176	2.47	112 × 16 × 176	2.47
doitgen	eucpad	8 × 14 × 16 × 150	3.26	5 × 14 × 24 × 128	3.21	4 × 68 × 16 × 150	2.49	4 × 68 × 16 × 150	2.49	5 × 127 × 24 × 128	2.66	5 × 127 × 24 × 128	2.66
	TSS	32 × 8 × 8 × 104	2.92	16 × 8 × 24 × 120	3.07	32 × 8 × 32 × 80	2.38	32 × 8 × 32 × 80	2.38	32 × 24 × 24 × 120	2.52	32 × 24 × 24 × 120	2.52
	Best Tile	128 × 32 × 560	4.82	128 × 48 × 496	3.83	128 × 48 × 2000	2.30	128 × 48 × 2000	2.30	128 × 192 × 496	1.82	128 × 192 × 496	1.82
jacobi-2d	eucpad	64 × 64 × 32	4.73	64 × 32 × 240	3.93	128 × 96 × 256	2.45	128 × 96 × 256	2.45	128 × 224 × 256	1.30	128 × 224 × 256	1.30
	TSS	128 × 32 × 560	4.82	128 × 48 × 496	3.83	128 × 48 × 2000	2.30	128 × 48 × 2000	2.30	128 × 192 × 496	1.82	128 × 192 × 496	1.82
	Best Tile	64 × 64 × 32	4.73	64 × 32 × 240	3.93	128 × 96 × 256	2.45	128 × 96 × 256	2.45	128 × 224 × 256	1.30	128 × 224 × 256	1.30
seidel-2d	eucpad	128 × 208 × 16	1.31	96 × 14 × 16	1.25	128 × 208 × 16	1.07	128 × 208 × 16	1.07	128 × 14 × 16	1.34	128 × 14 × 16	1.34
	TSS	64 × 192 × 16	1.32	32 × 64 × 16	1.28	32 × 64 × 16	1.15	32 × 64 × 16	1.15	64 × 208 × 16	1.3	64 × 208 × 16	1.3
	Best Tile	64 × 192 × 16	1.32	32 × 64 × 16	1.28	32 × 64 × 16	1.15	32 × 64 × 16	1.15	64 × 208 × 16	1.3	64 × 208 × 16	1.3

the conflict misses. For example, *eucpad* chooses a tile size of $(h,w) = (40,88)$ for the *matmul* kernel. The TSS algorithm, on the other hand, generates $(40,80)$ as one of the maximal tiles for the L1 cache. In other words, the TSS algorithm predicts pronounced interference misses at the tile size $(40,88)$, which is actually observed—the L1 cache misses incurred by the tile size $(40,88)$ are 1.4 times more than those incurred by the size $(40,80)$.

Table IV also compares the tile size chosen by the TSS algorithm with the “Best Tile.” To generate the *Best Tile*, we extensively ran all combinations of tiled codes with each tile dimension that is a multiple of cache line size, ranging from a cache line size (8 for double precision and 16 for single precision floating data) to $\sqrt{\frac{L2Cache\ Size}{size(DataType)}}$ (176 for double precision and 256 for single precision floating data). The reason for choosing this range of tile sizes is that the working set that results from these sizes is large enough to occupy the L2 cache, which is essential to compare with the TSS algorithm. The search space is chosen considering the L2 cache size on Sandy Bridge, as a large shared cache in Core microarchitecture would yield an unmanageable search space (the search space with the chosen limits already leads to 10,648 tiles for double-precision and 4,096 tiles for single-precision floating point data arrays). However, still, the *Best Tile* obtained for both microarchitectures clearly demonstrates the various factors impacting the tile size, and that the tile size generated by the TSS algorithm is close to the optimal tile size.

Table IV shows that for linear algebra and data mining kernels with 3D loop nests, the tile size chosen by TSS, like that of *Best Tile*, are rectangular and are such that the outermost and innermost intratile dimensions (I and J , respectively) are large. This favors both vectorization and data reuse in the L2 cache, as discussed in Section 4. Further, since the chosen tile size does not incur any interference misses, the performance achieved by TSS is always close to that achieved by the *Best Tile*. Since the tile sizes chosen are rectangular, in some cases the optimal tile size cannot be captured within the already large search space, and thus the tile size chosen by TSS could even outperform the *Best Tile*. This is particularly observed for the Core microarchitecture that has a considerably large L2 cache. From the table, an important observation can be made about the *Best Tile* sizes—the tile dimension corresponding to the innermost (vector) loop (96, for example, in *matmul*) is not the maximum possible length (176 for *matmul*) as would be expected to favor vectorization. The *Best Tile* instead conforms to the cost model used by TSS (i.e., $\min(\frac{1}{I} + \frac{1}{J})$), thus validating the importance of reusing the data in the L2 cache.

For the linear algebra and data mining kernels considered, PLuTo generates tiled code in which the goals of achieving data reuse in the L2 cache and efficient vectorization are concomitant. However, as discussed in Section 4.6, these goals conflict in the case of the tiled *strsm* benchmark, where the algorithm decides in favor of efficient vectorization and data reuse in only the L1 cache. In this case, the outermost loop is effectively left untiled by choosing the tile size for that loop to be the problem size, as shown in Table IV. Another important observation from Table IV is that although TSS performs significantly better than *eucpad* for most benchmarks, their performance is comparable for the *lu* and *trisolv* benchmarks. The reason for this is that both of these benchmarks have nonrectangular iteration spaces, and thus the reuse distance in different tiles varies as a function of their position in the iteration space. Our algorithm underestimates the tile size in such cases. However, since no conflict misses result from underestimating the tile size, our algorithm still achieves good performance in such cases.

Among the linear algebra kernels, *doitgen* has 3D arrays and a loop nest that is tiled in four dimensions. To estimate the tile size for *doitgen*, the framework first identifies the array references carrying reuse in the L2 cache. The TSS algorithm (modified for

Fig. 8. Comparison of TSS with other approaches that use 3D tiling.

3D arrays, as discussed in Section 5.1) generates a list of tile size triplets, all of which represent maximal tiles that do not incur interference. From this list, the triplet that maximizes reuse in the L2 cache according to the TCM/RR analysis is chosen as an estimate of the optimal tile size. The TCM/RR analysis suggests that for a tile size triplet, (P, Q, R) , the tile dimension corresponding to the innermost loop (R in this case) and the tile dimension corresponding to the outermost loop (the product $P * Q$ in this case) should both be large. Thus, the tile size triplet that satisfies these criteria is selected from the list. With P , Q , and R known, the remaining tile dimension, S , is computed by considering reuse in the L1 cache, as demonstrated earlier for the case of 3D loop nests through Figure 4.

Of the two stencil benchmarks considered, *seidel* is representative of nonvectorizable stencils and *jacobi* is representative of vectorizable stencils. For *seidel*, it can be observed that the analysis for data reuse in the L2 cache presented in Section 4.6 holds true—the *Best Tile* is rectangular with a larger tile height. The tile size corresponding to the outermost time loop is determined from the formula derived earlier in the same section, and the tile size determined by TSS performs close to the *Best Tile*. For *jacobi*, however, the TSS algorithm chooses a tile size to benefit from vectorization and intratile reuse in the L2 cache, at the cost of intertile reuse in the L2 cache. Results show that the TSS algorithm achieves performance comparable to the *Best Tile* in such cases as well. The difference in the optimal tile sizes for these two classes of stencils clearly validate both the impact of vectorization and reuse in the L2 cache on tile size selection.

7.1. Comparison with Other Approaches

Existing Compilers. Figure 8 plots normalized performance improvements achieved by the TSS algorithm, the default tiling in PLuTo, and the Best Cubic Tile (BCT) with respect to the Intel's C compiler. We show results for 9 of the 12 benchmarks considered on Sandy Bridge. The best cubic tiles have been chosen for comparison, as some auto-tuning frameworks [Whaley et al. 2001; Yuki et al. 2010] only consider cubic tiles to decide the best tiled code. Results show that the TSS algorithm outperforms the best cubic tile in all cases, achieving an average improvement of 9.7% and 20.4% for the two problem sizes. It is interesting to note that ICC could only tile the *matmul* benchmark (marked with an asterisk) among these 9 benchmarks, where it chose a cubic tile of size 128. Choosing such a tile size leads to a working set size that is midway between the L1 and L2 cache size. The rationale is that some data could be reused in the L1 cache, and if there are some misses, they could be serviced by the L2 cache. Although an intuitive approach, it cannot match the performance of the TSS algorithm since it only considers one level of data reuse, whereas as shown by our analysis, there are at least two (loop) levels at which data can be reused (loops kT and i for the *matmul*

Table V. Comparison between TSS and C+T

Benchmark	Scheme	N=2,000		N=4,000		N=256		N=1,024	
		Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)
matmul	TSS	$168 \times 32 \times 104$	4.1	$168 \times 32 \times 104$	4.28	$40 \times 7 \times 504$	3.73	$40 \times 7 \times 504$	3.52
	C+T	$50 \times 50 \times 50$	3.74	$50 \times 50 \times 50$	3.89	$64 \times 64 \times 64$	3.73	$64 \times 64 \times 64$	4.38
dsyrk	TSS	$168 \times 32 \times 104$	3.72	$168 \times 32 \times 104$	3.60	$40 \times 7 \times 504$	2.85	$40 \times 7 \times 504$	3.58
	C+T	$50 \times 50 \times 50$	3.02	$50 \times 50 \times 50$	2.80	$64 \times 64 \times 64$	2.72	$64 \times 64 \times 64$	3.71
dsyr2k	TSS	$120 \times 8 \times 88$	5.11	$120 \times 8 \times 88$	4.98	$16 \times 4 \times 504$	3.87	$16 \times 4 \times 504$	4.30
	C+T	$40 \times 40 \times 40$	4.14	$40 \times 40 \times 40$	3.75	$32 \times 32 \times 32$	3.30	$32 \times 32 \times 32$	2.59

benchmark). In case of PLuto, default cubic tiles of size 32 usually under-utilize the L1 cache and thus perform sub-optimally.

The DL/ML model - The authors in Shirako et al. [2012] have recently proposed a useful analytical model for bounding the search space of tile sizes. The authors compute the Minimum working set Lines (ML) and the number of Distinct Lines (DL) [Ferrante et al. 1992] as a function of the tile size. These ML and DL models provide the upper and lower bounds on the tile size to ensure intratile data reuse in the L1 cache. The authors further use the DL model to enable intertile data reuse in a higher level of cache. However, using these three bounds and only considering tile sizes in multiples of the cache line size, we get a search space of 7,931 tile sizes for the *matmul* benchmark on the Sandy Bridge processor used in our experiments. The size of the search space is similar for other benchmarks as well, which although much smaller than the original search space is nonetheless significantly large. Our work in this article accommodates cache set associativity to narrow down the search space to a list of maximal tile sizes, which maximally utilize the caches without causing interference. As a result, all maximal tiles obtained by the TSS algorithm (for linear algebra kernels) lie within the search space obtained from DL/ML, and thus the performance achieved is similar in both approaches. The TSS algorithm further uses a cost model derived from general principles underlying data reuse in linear algebra (and data mining) kernels to choose a single tile from the list of maximal tiles that minimizes cache misses.

For stencil codes, however, intertile data reuse does not happen in the same manner as in linear algebra codes. For example, for *seidel-2d* stencil code, the lower bound according to DL model on the Sandy Bridge processor (with 32KB L1 cache) is given as, $(H + T + 1) * (\lceil \frac{W+T}{CLS} \rceil + 1) \geq 512$, where W and H are tile width and height, respectively, and T is the tile size corresponding to the outermost time loop. Using this lower bound, the best tile size ($32 \times 64 \times 16$) for the problem size of 512 and various other close to the best tile sizes, including that chosen by TSS, lie outside the search space of DL/ML. This is because, in the optimal tile, the tile height should be less to avoid interference misses, and the tile width should be less to gain intertile reuse in the L2 cache on account of *shifting* tiles, as discussed in Section 4.5. The DL/ML model does not consider either and thus cannot adequately bound the search space. The tile size chosen by the TSS algorithm considers these factors affecting data reuse and achieves good performance.

Copy and Tile. Table V compares the TSS algorithm with Copy and Tile (C+T) for three out of six level-three BLAS kernels on Sandy Bridge. C+T emulates the *data copying* performed in ATLAS, where the innermost matrix is copied in a block-major layout as described in Whaley et al. [2001] to prevent conflict misses. After data copying, a cubic tile is chosen that maximizes the L1 cache usage based on the inequality derived in Yotov et al. [2005], which is a refinement over Whaley et al. [2001]. Further, we also ensure spatial locality for the matrices as in ATLAS and choose the tile size to be a multiple of the problem size to avoid complex clean-up code. We do not, however,

Table VI. Performance of TSS Algorithm in a Multithreaded Environment

Scheme	matmul ($N=2,000$)				dsyr2k ($N=2,000$)			
	4 threads		8 threads		4 threads		8 threads	
	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)
BCT	$64 \times 64 \times 64$	7.48	$64 \times 64 \times 64$	13.96	$64 \times 64 \times 64$	6.06	$64 \times 64 \times 64$	10.07
TSS	$128 \times 8 \times 80$	7.84	$128 \times 8 \times 80$	15.3	$40 \times 8 \times 40$	6.33	$40 \times 8 \times 40$	10.53
Best Tile	$72 \times 16 \times 120$	8.12	$64 \times 16 \times 96$	15.63	$72 \times 8 \times 72$	7.02	$96 \times 16 \times 96$	11.23

compare with ATLAS, as ATLAS also performs other optimizations such as register tiling and pipeline scheduling in addition to tile size selection.

Results in Table V show that the TSS algorithm achieves a performance improvement of (17%) over C+T for problem sizes that are not a power of 2 (2,000 and 4,000). This improvement is attributed to (1) data reuse in the L2 cache exploited by the tile size chosen by TSS, whereas C+T cannot capture this opportunity and results in a 4% higher L2 cache miss rate than TSS, and (2) since C+T only considers cubic tile sizes, it cannot take advantage of the large vector pipeline length made possible by considering rectangular tiles.

On the other hand, the performance of C+T and TSS is comparable for the pathological problem sizes (256 and 1,024), with the former even outperforming TSS in the case of matmul. This is because, to avoid interference for such problem sizes, the TSS algorithm chooses ‘thin’ tiles (e.g., the tile size chosen for *matmul* at $N = 1,024$ is $40 \times 7 \times 504$). This leads to less effective data reuse in the L2 cache, as compared to square tiles chosen by C+T. The reason that C+T could choose square tiles even for these pathological problem sizes stems from the fact that it is immune to the effects of problem size as a result of the copy optimization. Data copying, however, is a complex optimization and is not supported by production-quality compilers such as GCC and ICC. Since the TSS algorithm does not rely on copy optimization, it can be readily implemented in any production-quality compiler.

7.2. Performance of the TSS Algorithm in a Multithreading Environment

Table VI shows the performance of TSS algorithm using 4 and 8 threads on the Intel Xeon processor based on Sandy Bridge microarchitecture. For the purpose of evaluation, we ran the same 10,648 different tiled versions of *matmul* and *dsyr2k* benchmarks to obtain the “Best Tile.” In the table, we compare the performance of the TSS algorithm with the *Best Tile* and the BCT code.

On Sandy Bridge that allows two-way hyperthreading, 2 threads on a core share resources including private L1 and L2 caches on the core and bring in data simultaneously to the caches. This is incorporated in the TSS algorithm by reducing the effective set associativity by a factor of 2 or the number of threads sharing the cache. As a result, a smaller tile size is chosen by our algorithm when using 4 and 8 threads (run on 2 and 4 cores, respectively), performs comparable to the *Best Tile*, and outperforms the BCT code. A smaller tile size is similarly observed for the best cubic tiles using this two-way hyperthreading technology as compared to that observed when using a single thread per core, which further validates our proposed effect of multithreading on tile size selection. As a result of using smaller tiles instead of the same estimates of tile size for a single core, the TSS algorithm achieves an improvement of 28% and 7%, respectively, for *matmul* and *dsyr2k* benchmarks. A similar phenomenon is observed (data not shown), when multiple cores share the L2 cache on the Core2 processor using CMP technology.

8. RELATED WORK

In the past, the problem of optimal tile size selection has been attempted using two approaches: (1) through an analytical framework based on analyzing the interaction between the source program and the host architecture, and (2) through an extensive search at runtime to tune for tile size. In approach (1), there has been considerable work in the past [Lam et al. 1991; Esseghir 1993; Coleman and McKinley 1995; Ghosh et al. 1997; Chame and Moon 1999; Rivera and Tseng 1999; Hsu and Kremer 2004]. However, none of these works has proved to be effective for a range of programs. The authors in Lam et al. [1991], Esseghir [1993], and Ghosh et al. [1997] study the impact of problem size and cache parameters on the tile size to eliminate self-interference misses but do not consider cross-interference misses. On the other hand, the authors in Coleman and McKinley [1995] and Chame and Moon [1999] aim to minimize cross-interference misses as well. However, they assume the cache to be direct mapped and use probabilistic approaches for minimizing conflicts that leads to suboptimal performance, especially for the pathological problem sizes. The authors in Rivera and Tseng [1999], Panda et al. [1999], and Hsu and Kremer [2004] address this problem of robustness by using array padding for such pathological problem sizes. However, these works do not accommodate set-associative caches in their models, nor do they consider the option of exploiting data reuse in the multiple levels of cache. The authors in Chatterjee et al. [2001] use Presburger formulas to express cache misses and can consider moderate levels of associativity, but they cannot accommodate the high set associativity in modern caches.

Of all works in approach (2), the most prominent work is the ATLAS library generator [Whaley et al. 2001], which generates an optimized BLAS library on a given target platform. However, ATLAS relies on an extensive time-consuming search for this purpose. Other self-tuning library generators [Bilmes et al. 1997; Frigo 1999; Xiong et al. 2001] suffer from the same problem of large search space. Tiwari et al. [2009] combine the CHiLL [Chen et al. 2008] compiler framework with Active Harmony [Țăpuș et al. 2002] to manage the cost of this search space but rely on the user for the copy optimization. Yotov et al. [2005] improve upon ATLAS by showing that search is not necessary to generate optimized BLAS. However, they use the code generator of ATLAS, which relies on the ad hoc copy optimization to reduce interference and thus estimate the optimal tile size. The automatic tile size creation model of Yuki et al. [2010] avoids array copying but only considers cubic tiles to contain the data collection time. However, Goto and Van De Geijn [2008] and Datta [2009] show that for linear algebra and stencil codes, respectively, the optimal tile is not cubic, which is also corroborated by our work in this article.

Chen et al. [2005] and Shirako et al. [2012] combine an analytical model with empirical search to manage the search space. Chen et al. rely on copying to work with a reduced search space and use heuristics to traverse it. Shirako et al. propose a model to determine the upper and lower bounds on the search space using the existing DL model [Sarkar 1997] and their ML model, and consider data reuse in multiple levels of cache. However, the ML model assumes an optimal cache replacement policy, thus effectively ignoring the impact of set associativity in caches. In this article, we accommodate the impact of set associativity in our analysis of data reuse.

Recently, the auto-tuning framework has been improved to select the optimal tile size while in the production run (ETile [Tavarageri et al. 2011]) using parameterized tiled code [Baskaran et al. 2010]. ETile monitors a few loop iterations for a tile size to determine the optimality. However, this does not reduce the search space and may not give the best tile, as it is difficult to capture reuse in a few iterations, especially if data reuse is in the outermost loop.

9. CONCLUSION

This article revisits the problem of tile size selection in the context of modern processors. It proposes an analytical framework that unveils several factors that had not been considered by previous models. It particularly considers the data reuse in not just the L1 but also the L2 cache. In addition, it considers the interaction of the SIMD unit in modern processors with tile size selection. The TSS algorithm proposed in the article chooses rectangular tiles that benefit from both vectorization and data reuse in multiple levels of cache. Experimental results indicate that the TSS algorithm achieves significant improvement over the previous analytical models that did not consider these important factors on modern processors. In addition, results indicate that the tile size chosen by the TSS algorithm is comparable to exhaustive search in a constrained search space. Since the TSS algorithm estimates the optimal tile size at compile time, it can be readily incorporated within any production-quality compiler.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback that helped us improve the quality of our work. We would also like to acknowledge NSF grants CNS-0834599 and CCF-0708822 for supporting this work.

REFERENCES

- BASKARAN, M. M., HARTONO, A., TAVARAGERI, S., HENRETTY, T., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Parameterized tiling revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, 200–209.
- BASTOUL, C. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. 7–16.
- BILMES, J., ASANOVIC, K., CHIN, C.-W., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 1997 International Conference on Supercomputing (ICS'97)*. ACM, 340–347.
- BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08*. ACM, 101–113.
- CHAME, J. AND MOON, S. 1999. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 1999 International Conference on Supercomputing (ICS'99)*. ACM, 492–499.
- CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 286–297.
- CHEN, C., CHAME, J., AND HALL, M. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO'05*. 111–122.
- CHEN, C., CHAME, J., AND HALL, M. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Technical Report (2008), 08–897. University of Southern California.
- COLEMAN, S. AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, 279–290.
- COOPER, K. AND SANDOVAL, J. 2011. *Portable Techniques to Find Effective Memory Hierarchy Parameters*. Technical Report.
- DATTA, K. 2009. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. Technical Report. University of California at Berkeley.
- DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND DUFF, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 16, 1, 1–17.
- ESSEGHIR, K. 1993. *Improving Data Locality for Caches*. Ph.D. Dissertation. Rice University.
- FERRANTE, J., SARKAR, V., AND THRASH, W. 1992. On estimating and enhancing cache effectiveness. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'92)*. Springer, Berlin, 328–343.
- FRIGO, M. 1999. A fast Fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, 169–180.
- GHOSH, S., MARTONOSI, M., AND MALIK, S. 1997. Cache miss equations: an analytical representation of cache misses. In *Proceedings of the 1997 International Conference on Supercomputing (ICS'97)*. ACM, 317–324.

- GOTO, K. AND VAN DE GEIJN, R. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.* 35, 1, Article 4 (July 2008), 14 pages.
- HSU, C.-H. AND KREMER, U. 2004. A quantitative analysis of tile size selection algorithms. *Journal of Supercomputing* 27, 3, 279–294.
- KNIJNENBURG, P. M. W., KISUKI, T., AND O'BOYLE, M. F. P. 2003. Combined selection of tile sizes and unroll factors using iterative compilation. *Journal of Supercomputing* 24, 1, 43–67.
- LAM, M. D., ROTHBERG, E. E., AND WOLF, M. E. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*. ACM, 63–74.
- LIM, A. W., LIAO, S.-W., AND LAM, M. S. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices* 36, 103–112.
- PANDA, P. R., NAKAMURA, H., DUTT, N., AND NICOLAU, A. 1999. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers* 48, 2, 142–149.
- POUCHET, L.-N. 2013. Polybench Benchmark Suite. (2013). Available at <http://www-roc.inria.fr/~pouchet/software/polybench/>.
- RAMANUJAM, J. AND SADAYAPPAN, P. 1992. Tiling multidimensional iteration spaces for multicomputers. *J. Parallel and Distrib. Comput.* 16, 2, 108–120.
- RIVERA, G. AND TSENG, C.-W. 1999. A comparison of compiler tiling algorithms. In *Compiler Construction*. Springer, 168–182.
- SARKAR, V. 1997. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM Journal of Research and Development* 41, 3, 233–264.
- SARKAR, V. AND MEGIDDO, N. 2000. An analytical model for loop tiling and its solution. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*. 146–153.
- SHIRAKO, J., SHARMA, K., FAUZIA, N., POUCHET, L.-N., RAMANUJAM, J., SADAYAPPAN, P., AND SARKAR, V. 2012. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. Springer Verlag, Estonia.
- ȚĂPUȘ, C., CHUNG, I.-H., AND HOLLINGSWORTH, J. K. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'02)*. IEEE Computer Society Press, Los Alamitos, CA, 1–11.
- TAVARAGERI, S., POUCHET, L.-N., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2011. Dynamic Selection of Tile Sizes. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC'11)*.
- TEMAM, O., GRANSTON, E. D., AND JALBY, W. 1993. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC'93)*. 410–419.
- TIWARI, A., CHEN, C., CHAME, J., HALL, M., AND HOLLINGSWORTH, J. K. 2009. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS'09)*. 1–12.
- WHALEY, R. C., PETTIT, A., AND DONGARRA, J. J. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 3–35.
- WOLF, M. E. AND LAM, M. S. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*. ACM, 30–44.
- WOLFE, M. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (SC'89)*. ACM, 655–664.
- WONNACOTT, D. 2002. Achieving scalable locality with time skewing. *International Journal of Parallel Programming* 30, 3, 181–221.
- XIONG, J., JOHNSON, J., JOHNSON, R., AND PADUA, D. 2001. SPL: a language and compiler for DSP algorithms. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 298–308.
- YOTOV, K., LI, X., REN, G., GARZARAN, M. J. S., PADUA, D., PINGALI, K., AND STODGHILL, P. 2005. Is search really necessary to generate high-performance BLAS? *Proc. IEEE* 93, 2, 358–386.
- YUKI, T., RENGANARAYANAN, L., RAJOPADHYE, S., ANDERSON, C., EICHENBERGER, A. E., AND O'BRIEN, K. 2010. Automatic creation of tile size selection models. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'10)*. ACM, 190–199.

Received June 2013; revised August 2013; accepted October 2013