2001

Effective Use of the Level-Two Cache for Two Cache for Skewed Tiling (Extended Version)

Yonghong Song

Zhiyuan Li
Purdue University, li@cs.purdue.edu

Report Number:
01-006
EFFECTIVE USE OF THE LEVEL-TWO CACHE FOR SKEWED TILING
(EXTENDED VERSION)

Yonghong Song
Zhiyuan Li

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #01-006
April 2001
Effective Use of The Level-Two Cache for Skewed Tiling  
(Extended Version) *

Yonghong Song Zhiyuan Li  
Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907  
{songyh,li}@cs.purdue.edu

Abstract

Tiling is a well-known loop transformation technique to enhance temporal data locality. In our previous work, we have developed a skewed tiling technique for relaxation codes, which requires to apply loop skewing before loop tiling. In this paper, we study how to effectively use the level-two cache for skewed tiling through a tile-size selection algorithm, STS. Particularly, we address two questions: (1) when to focus on enhancing locality for the L2 cache instead of the L1 cache, and (2) how to improve the L2 cache locality such that the overall performance can be improved. We address the first question by developing an execution cost model which incorporates both the L1 and the L2 cache misses. We address the second question by applying inter-array padding to minimize cross-interference misses.

We compare STS with several previously known algorithms. For certain test cases, STS is significantly better than those previous algorithms because it effectively exploits the L2 cache locality. For other cases, STS achieves comparable results because it also effectively exploits the L1 cache locality. For two well-known SPEC benchmarks with different inputs on two different machines, we also compare our inter-array padding algorithm with a previously-proposed padding algorithm. Our padding algorithm is significantly better.

1 Introduction

Memory access latency has become a key performance bottleneck on modern microprocessors. An important approach to reducing the average latency is to exploit data locality on the cache memories. Tiling is a well-known compiler technique to enhance data locality such that more data can be reused before they are replaced from the cache [23]. Tiling transforms a loop nest by combining strip-mining and loop interchange. Loop skewing and loop reversal are often used to enable tiling [20]. Figure 1 shows SOR relaxation as an example. Figure 1(a) shows the original loop nest in SOR, Figure 1(b) shows the tiled SOR in which loop J is skewed with respect to loop T (1-D tiling), and Figure 1(c) shows the tiled SOR in which both loops J and I are skewed with respect to loop T (2-D tiling). In this paper, we call tiling enabled by loop skewing skewed tiling. Much of previous work on tiling applies to perfectly-nested loops only [6, 20, 21, 23]. Since very

---

*This work is sponsored in part by National Science Foundation through Grants CCR-9975309, ITR/ACI-0082834 and MIP-9610379, by Indiana 21st Century Fund, by Purdue Research Foundation, and by a donation from Sun Microsystems, Inc.
Few programs arc known to have perfectly-nested loops, we recently proposed a new skewed-tiling technique to tile a class of imperfectly-nested loops [15, 16], including typical loops that perform iterative relaxation computations [15, 16].

Performance of a tiled loop nest can vary dramatically with different tile sizes [7]. How to select a proper tile size is hence an important issue. All previous publications on tile-size selection tacitly assume non-skewed tiling [2, 4, 7, 10, 14, 22]. Particularly, they assume that each tile repeatedly accesses the same set of data. This is certainly not true for skewed tiling. Furthermore, they consider only the L1, not the L2, cache misses.

Like the SOR code in Figure 1, many relaxation codes can be tiled with either 1-D or 2-D tiling. If the L1 cache is the only target for locality enhancement, then 2-D tiling will invariably be chosen over 1-D tiling, as is done in previous works. Our main claim in this paper is that the overall performance can be improved by including the L2 cache in the cost model. Moreover, under such a more comprehensive cost model, 1-D tiling may be chosen over 2-D tiling because it results in fewer L2 cache misses. We support our claim by making the following analytical and experimental contributions:

• We develop an execution cost model and utilize it in a tile-size selection algorithm called STS [18]. Unlike previous algorithms, our model incorporates two performance factors, namely cache misses (on both the L1 and L2 caches) and loop overhead, into a single performance-estimation model. Moreover, when estimating the number of cache misses, the effect of loop skewing is taken into account. The loop overhead is incorporated to avoid small tile heights ($B_2$ in Figure 1(c)), for which loop overhead could be significant. The choice between 1-D and 2-D tiling and the choice of tile size is determined by a comparison between the different execution costs.

• We adopt an inter-array padding [11] technique to reduce L2 cache misses due to interferences, thus enhancing the L2 cache locality.

• We report experimental results of three relaxation kernels which can be tiled at two loop levels. We compare our tile-size selection algorithm, STS, with previous algorithms. We measure the execution time, as well as the L1 and L2 cache misses, on two machines. Our results show that, for two programs, where the L1 cache locality can be exploited in most cases, STS achieves comparable results. For one program, where the L2 cache locality is exploited in most cases, STS is significantly better.

• We also report experimental results of two SPEC benchmarks which can be transformed at one loop level only. For these two programs, some competing tile-size selection algorithms no longer apply because they always generate square tile sizes. Comparing to the remaining applicable algorithms, STS is superior because it utilizes inter-array padding to improve the performance on the L2 cache. We compare our inter-array padding algorithm with another padding algorithm, GroupPad [13], which targets the L1 cache. We conclude that our inter-array padding is more appropriate for the STS. We also show that the results obtained by STS are within 5% of the optimal (among a range of tile sizes) except in one test case where STS underperforms the optimal by 13%.

This paper is a heavily-revised version of our previous technical report [17] in the following aspects:

• Unlike [17], we drop the TLB consideration in STS. We justify our decision in Sections 6.4 and 6.6.
DO \( T = 1, ITMAX \)
DO \( J = 2, N - 1 \)
DO \( I = 2, N - 1 \)
\( A(I, J) = (A(I, J) + A(I + 1, J) + A(I - 1, J) + A(I, J + 1) + A(I, J - 1))/5 \)
END DO
END DO
END DO

(a) Before transformation  (b) After skewing and "1-D" tiling  (c) After skewing and "2-D" tiling

Figure 1: An example of tiling: SOR relaxation.

- Unlike [17], we drop the consideration of software pipelining effects in STS because (1) the assumption we made in [17] that each load instruction takes one cycle to execute in a software-pipelined loop is not valid in most cases, and (2) the software pipelining only plays a marginal role in tile-size selection, i.e., without considering it, the performance does not change much. We justify our decision in Section 6.4.

- Because of the above algorithm-level change, our experimental results are heavily changed. In this paper, we further evaluate the impact of loop overhead and give some deeper discussions.

In the rest of the paper, we first compare with previous work in Section 2. We present a background in Section 3. We then present our memory cost model in Section 4. We model the execution time and present our tile-size selection algorithm in Section 5. In Section 6, we experimentally compare our algorithm with previous algorithms. Finally, we conclude in Section 7.

2 Previous Work

2.1 Competing Tile-Size Selection Schemes

Several tile-size selection algorithms have been proposed before STS, which are listed below.

- TLI by Chame and Moon [1] uses an execution cost model which includes the capacity misses and the cross-interference misses. TLI enumerates a range of tile sizes which are free of self-interference misses and selects the one with the smallest execution cost as the optimal tile size.

- TSS by Coleman and McKinley [2] uses a GCD algorithm to select a number of candidate tile sizes. Among such candidates, the one resulting in the largest array footprint is chosen as the optimal. Rivera and Tseng [14] present a variation of TSS.

- LRW by Lam et al. [7] chooses the largest square tile size which is free of self-interference misses.

- DAT by Panda et al. [11] chooses the maximum square tile size which are free of capacity misses. It then applies inter-array padding to minimize cross-interference misses.

When computing the tile size, these algorithms consider the L1, but not the L2, cache misses. They also ignore the effect of loop skewing even if skewed tiling is applied. By assuming non-skewed
Table 1: Comparison between various tile-size selection algorithms

<table>
<thead>
<tr>
<th></th>
<th>LRW</th>
<th>TSS</th>
<th>TLI</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Skewing</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LI data cache</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>L2 cache</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Dominant Array</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Padding</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Tile dimensions</td>
<td>2</td>
<td>1,2</td>
<td>1,2</td>
<td>1,2</td>
<td>2</td>
</tr>
<tr>
<td>Tile shape</td>
<td>squ.</td>
<td>rect.</td>
<td>rect.</td>
<td>rect.</td>
<td>squ.</td>
</tr>
</tbody>
</table>

tiling, previous algorithms can be applied to more general loop structures and array reference patterns than STS. On the other hand, STS considers the effect of skewed tiling when considering the L1 cache misses. Moreover, STS counts L2 cache misses, as well as the loop overhead, such that a decision on loop levels for tiling can be correctly made.

Of all arrays in a tiled loop, LRW, TSS and TLI identify a dominant array and choose the tile size to eliminate certain kinds of cache misses due to the dominant array. All other arrays are ignored. In contrast, both DAT and STS consider all arrays. Among all tile-size selection techniques, only DAT and STS utilize padding, a data transformation technique to eliminate certain interference misses [13]. All tile-size selection algorithms discussed in this paper can be applied to loop nests which can be tiled at two loop levels. In other words, they can be applied to two-dimensional tiles. LRW and DAT require a square tile shape. In contrast, STS, TSS and TLI permit rectangular tiles. For loop nests which can be tiled at one loop level only, LRW and DAT no longer apply. However, STS, TSS and TLI still apply in such cases because as far as tile-size selection is concerned, an one-dimensional tile can be viewed as a degenerate case of a two-dimensional rectangular tile, where the tile height equals to the trip count of the inner loop.

Table 1 summarizes the characteristics of various algorithms discussed here, where “squ.” stands for “square” and “rect.” for “rectangular”.

### 2.2 Other Previous Work

Rivera and Tseng present several intra-array padding algorithms, which increase the array column sizes, to eliminate cache conflict misses [13, 14]. Their algorithms, however, cannot be applied after LRW, TLI and TSS are applied because the array column size is used in these three tile-size selection algorithms. Rivera and Tseng do not give an algorithm to determine the intra-array padding size before tiling [14]. In [13], Rivera and Tseng also present an inter-array padding algorithm, GroupPad, to exploit group reuses across the outer loops. Their padding algorithm targets the L1 cache. We will compare our padding algorithm with theirs for the tiled codes in Section 6.5.

Ghosh et al. estimate cache misses, given a tile size, for a perfect loop nest [4]. They also informally discuss a tile-size selection scheme using matrix multiplication as the example. No formal algorithm is presented, however. They do not discuss the estimation of cache misses for imperfectly-nested loops. Therefore, we are not able to compare with their method in our experiments.

Ferrante et al. present an algorithm to estimate the number of distinct cache lines over a perfect loop nest [3]. Temam et al. derive an analytical method to estimate the number of self-interference misses [19]. Mckinley et al. present a simple cost model to estimate the number of cache misses [9]. These methods do not consider the effect of loop skewing.

Manjikian and Abdelrahman use cache partitioning to scatter arrays evenly in the cache, such
that cross-interference misses are minimized [8]. We use a different padding scheme which seems more suitable for our algorithm.

3 Background

In this section, we first define our program model and a few key parameters. We then discuss the issues of the memory hierarchy.

3.1 Skewed Tiling

Most of previous research on tiling addresses perfectly-nested loops only [6, 20, 21, 23]. After tiling, the loops remain perfectly-nested. In our recent work [15, 16], we perform tiling on a class of imperfectly-nested loops. Figure 2(a) shows a representative loop nest before tiling, where the T-loop body consists of m perfectly-nested loops. The depth of each perfectly-nested inner loop is at least two. The loop bounds $L_{ij}$ and $U_{ij}$, $1 \leq i \leq m$, $j = 1, 2$, are $T$-invariant. We assume that the iteration space determined by $J$ and $I$ remains unchanged over different $T$-loop index values. For simplicity of presentation, we also assume that cache-line spatial locality is already fully exploited in the innermost loops except on the loop boundaries. Figure 2(b) shows the code after tiling the $J_i$ loops only (1-D tiling), and Figure 2(c) shows the code after tiling both $J_i$ and $I_i$ loops (2-D tiling). In Figures 2(b) and 2(c), the iteration subspace defined by all $J_i$ and $I_i$ loops is called a tile. Loop $T$ is called the tile-sweeping loop, and loops $JJ$ and $II$ are called the tile-controlling loops [20]. Each combination of $JJ$ and $II$ defines a tile traversal. Two tiles are said to be consecutive within a tile traversal if the difference of the corresponding $T$ values equals 1.

Let $\gamma_1 = \min\{L_{ij}|1 \leq i \leq m\}$, $\gamma_2 = \max\{U_{ij}|1 \leq i \leq m\}$, $\eta_1 = \min\{L_{ij}|1 \leq i \leq m\}$ and $\eta_2 = \max\{|U_{ij}|1 \leq i \leq m\}$. We call $S_1$ and $S_2$ the skewing factors corresponding to $J_i$ and $I_i$ loops respectively. (The skewing factors are also called the slope in our previous work [15, 16].) If $S_1 = 0$, then loop skewing is not applied before tiling at the $J_i$ level. In this paper, we are interested only in skewed tiling at least at the $J_i$ level, thus $S_1 > 0$. $B_1$ is called the tile width and $B_2$ is called the tile height. $B_1$ and $B_2$ are called the tile size collectively. These parameters are used to define the bounds of the tile-controlling loops. For reference, Table 2 lists all the symbols used in this paper and their brief descriptions.

In this paper, we assume the data dependences permit both 1-D and 2-D tiling. Choosing between 1-D vs. 2-D tiling will depend on the estimate of cache misses and loop overhead. For simplicity, we assume all arrays are of two dimensions with the same column sizes. (We assume column-major storage.) Lower dimension variables can be ignored due to their lesser impact on cache misses in relaxation programs which we are interested in. Let $n_a$ be the number of two dimensional arrays in the given tiled loop nest. Within the innermost loop $I_i$, $1 \leq i \leq m$, of the untiled program in Figure 2(a), we assume array subscript patterns of $A_k(I_i+a, J_i+b)$, $1 \leq k \leq n_a$, where $a$ and $b$ are known integer constants.

Although we restrict our program model to be either 1-D or 2-D tiling and restrict all arrays to be of two dimensions, such restrictions can be relaxed. We can have n-D tiling by extending our tiling technique in [16]. We can also allow high-dimensional arrays by extending our memory cost model and execution cost model. However, such extension to high-dimensional tiling and arrays seems unnecessary for the applications we currently have met.
DO \( \text{T} = 1, \text{ITMAX} \)
DO \( \text{J}_1 = \text{L}_1, \text{U}_1 \)
DO \( \text{J}_2 = \text{L}_2, \text{U}_2 \)
END DO
END DO
END DO

DO \( \text{J}_m = \text{L}_m, \text{U}_m \)
DO \( \text{I}_m = \text{L}_m, \text{U}_m \)
END DO
END DO
END DO
END DO

DO \( \text{J} = \text{L} + \text{J} \ast \text{H} \)
DO \( \text{J} = \text{L}_1, \text{U}_1 \)
DO \( \text{I} = \text{L}_2, \text{U}_2 \)
END DO
END DO
END DO
END DO

END DO

Figure 2: The program model before and after tiling

Table 2: Description of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{n}_1 )</td>
<td>The minimum lower bound of all ( \text{J}_j ) loops</td>
</tr>
<tr>
<td>( \text{n}_2 )</td>
<td>The minimum lower bound of all ( \text{I}_j ) loops</td>
</tr>
<tr>
<td>( \text{N} )</td>
<td>The tile width</td>
</tr>
<tr>
<td>( \text{n}_m )</td>
<td>The number of arrays in the given loop nest</td>
</tr>
<tr>
<td>( \text{C}_1 )</td>
<td>The L1 cache size in the number of data elements</td>
</tr>
<tr>
<td>( \text{C}_2 )</td>
<td>The L2 cache size in the number of data elements</td>
</tr>
<tr>
<td>( \text{ITMAX} )</td>
<td>The trip count for the tile sweeping loop</td>
</tr>
<tr>
<td>( \text{m} )</td>
<td>The L1 cache miss penalty</td>
</tr>
<tr>
<td>( \text{m} )</td>
<td>The L2 cache miss penalty</td>
</tr>
<tr>
<td>( \text{n}_3 )</td>
<td>The number of group reuses which generate L1 cache misses</td>
</tr>
<tr>
<td>( \text{n}_4 )</td>
<td>The number of group reuses which generate L2 cache misses</td>
</tr>
</tbody>
</table>

3.2 Memory Hierarchy

The memory hierarchy includes registers, cache memories at one or more levels, the main memory and the secondary storage [5].

For simplicity of presentation, we consider two levels of caches in this paper, namely the L1 and L2 caches, which are common in current practice. The L1 cache has several parameters, namely the cache size \( \text{C}_1 \), the cache block size \( \text{C}_b \) and the set associativity \( \text{C}_a \). \( \text{C}_1 \) and \( \text{C}_b \) are measured in the number of data elements. Similarly for L2 cache, the cache size, cache block size and set associativity are \( \text{C}_2 \), \( \text{C}_b \) and \( \text{C}_a \) respectively. The cache misses can be divided into three classes [5]: compulsory misses, capacity misses and conflict misses. Conflict misses can be attributed to self-interference misses of the same array and to cross-interference misses between different arrays.

4 A Memory Cost Model

In this section, we want to estimate the number of cache misses incurred by executing the loop nest in our program model after tiling.
Let $S_o$ represent the iteration space defined by $\gamma_1 \leq J_i \leq \gamma_2$ and $\eta_1 \leq I_i \leq \eta_2$ in Figure 2(a). (For simplicity, we also regard $S_o$ as the original iteration space defined by $J_i$ and $I_i$ loops in Figure 2(a), as if all $J_i$ loops have the same loop bounds and all $I_i$ loops have the same loop bounds.) $S_o$ is illustrated in Figure 3(a) by the rectangle enclosed by the solid lines with the height $\eta$ and the width $\gamma$. Within each tile traversal, we define the base tile to be a tile with $T = 1$ and an advanced tile to be a tile with $T > 1$. The dashed-lines in Figure 3(a) separate the base tiles of different tile traversals. The two shaded areas illustrate two different tile traversals, $tt_1$ and $tt_2$, where each shaded rectangle with solid-line boundaries represents an advanced tile. When the tile-sweeping loop $T$ increases the index by 1, the tiles can only overlap partially.

The cache misses incurred by one tile traversal can be partitioned into those within the base tile and those within the advanced tiles. Note that only those base tiles and advanced tiles overlapping with $S_o$ will be executed, thus only they can contribute to the cache misses. In Figure 3(a), the base tile in the tile traversal $tt_1$ resides outside $S_o$, while the base tile in $tt_2$ resides within $S_o$.

We make the following two assumptions in our estimation of the number of cache misses:

- **Assumption 1**: There exist no cache reuse between different tile traversals.
- **Assumption 2**: $B_1 \ll \gamma$ and $B_1 \ll (ITMAX-1) \ast S_1$.

Assumption 1 is reasonable if $ITMAX$ is large, since it will be very likely for a tile traversal to overwrite cache lines whose old data could have been reused in the next tile traversal. Assumption 2 is reasonable because a large $B_1$ can easily cause an overflow in the target cache, or result in a small $B_2$ which may generate a large loop overhead. As explained later in Section 5.1, STS incorporates loop overhead in execution cost estimation to prevent a very small $B_2$. If the tile size $(B_1, B_2)$ is chosen properly, there should be exactly one cache miss for each cache line accessed within a tile traversal. To be more specific, the following two properties should hold:

- **Property 1**: No capacity and self-interference misses are generated within a tile traversal.
- **Property 2**: No cross-interference misses are generated within a tile traversal.

In Section 5.2, we shall discuss how to preserve the above properties. For now, we assume they hold.

We estimate cache misses with the following four options:

- **Option 1**: 2-D tiling with the L1 cache targeted. The tile shape is illustrated in Figure 2(c). The footprint of each tile fits in both the L1 cache and the L2 cache. Inter-array padding is
applied to satisfy Property 2 on the L1 cache. For this option, both Properties 1 and 2 are satisfied on both the L1 cache and the L2 cache.

- **Option 2**: 1-D tiling with the L1 cache targeted. This is similar to Option 1, except for the different tile shape (see Figure 2(b)). Both Properties 1 and 2 are satisfied on both the L1 and the L2 caches.

- **Option 3**: 2-D tiling with the L2 cache targeted. The tile shape is illustrated in Figure 2(c). The footprint of each tile fits in the L2 cache, but not necessarily in the L1 cache. Inter-array padding is applied to satisfy Property 2 on the L2 cache. For this option, both Properties 1 and 2 are satisfied on the L2 cache, but not necessarily on the L1 cache.

- **Option 4**: 1-D tiling with the L2 cache targeted. This is similar to Option 1, except for the different tile shape (see Figure 2(b)). Both Properties 1 and 2 are satisfied on the L2 cache, but not necessarily on the L1 cache.

4.1 Estimation of Cache Misses for Option 1

We first show how to compute the number of L1 cache misses caused by an advanced tile. Let $W$ represent the size of the data set accessed by the original loop nest in terms of the number of data elements. The average size of the data accessed by one tile is estimated to be $D = \frac{W}{\eta_1} * B_1 B_2$.

Figure 3(b) shows two consecutive tiles, $t_{3}$ and $t_{4}$, within a tile traversal, assuming that both tiles reside within $S_0$. The iteration subspace of $t_{4}$ is produced by shifting the iteration subspace of $t_{3}$ upwards by $S_2$ iterations and to the left by $S_1$ iterations. The L1 cache misses in $t_{4}$ either occur in Region $A B C D$ or in Region $D E F G$. The total estimated L1 cache misses equal to $(S_1 B_2 + S_2 B_1 - S_1 S_2) * \frac{W}{\eta_1 C_{b1}}$. (This estimate may not be exact because data accessed at the lower border of Region $D E F G$ may or may not be in the cache already.)

We now show how to accumulate the number of L1 cache misses for all the tile traversals with the same $J J$ value. Figure 3(c) illustrates the idea. For a particular $J J$ value, let $t_1$, $t_2$, $t_3$ and $t_4$ be the base tiles of four tile traversals, and let $t'_1$, $t'_2$, $t'_3$ and $t'_4$ be the corresponding advanced tiles when $T$ increases by 1. In this particular illustration, the number of L1 cache misses caused collectively by $t_i$ ($1 \leq i \leq 4$) equals to the sum of the number of L1 cache misses caused by each individual $t_i$, that is, $\frac{W B_i}{\eta_1 C_{b1}}$. Note that only the tiles overlapping with $S_0$ can contribute to L1 cache misses. Similarly, the number of L1 cache misses caused by the advanced tiles $t'_i$ ($1 \leq i \leq 4$) equal to the sum of the number of L1 cache misses caused by individual $t'_i$, that is, $\frac{W B_i}{\eta_1 C_{b1}} + 2(B_1 - S_1)S_2 * \frac{W}{\eta_1 C_{b1}}$.

In general, the number of L1 cache misses caused by the advanced tiles with the same $J J$ value equal to $\frac{W}{\eta_1 C_{b1}} + \tau(B_1 - S_1)S_2 * \frac{W}{\eta_1 C_{b1}}$, where $\tau$ is the number of base tiles in $S_0$ for a particular $J J$ estimated as

$$\tau = \begin{cases} \left\lfloor \frac{W}{B_2} \right\rfloor & \text{if } 1 \leq B_2 < \eta + S_2 * (ITMAX-1) \\ 0 & \text{if } B_2 = \eta + S_2 * (ITMAX-1) \end{cases}$$

The value $\eta + S_2 * (ITMAX-1)$ is the maximum height of the iteration space after tiling. Any $B_2$ value greater than or equal to $\eta + S_2 * (ITMAX-1)$ results in no tiling at the $I_1$ loop level.

With Assumptions 1 and 2, we can then accumulate L1 cache misses corresponding to different $J J$ values by considering three different cases:

- **Case 1**: $\gamma = (ITMAX-1) * S_1$.
  This case is illustrated by Figure 4(a). In this case, the tile traversals defined by $J J \leq \gamma_1 + \gamma - NSTEP$ will not execute to the $ITMAX$th $T$-iteration. The tile traversal defined by $\gamma_1 + \gamma - NSTEP < J J \leq \gamma_1 + \gamma$ is the first to reach the $ITMAX$th $T$-iteration. The tile
traversals defined by $JJ > \gamma_1 + \gamma$ will start executing at $T > 1$. During the execution, the tile traversals defined by $JJ = \gamma_1$ will incur L1 cache misses of $\frac{WB_1}{\gamma C_{B_1}}$. The tile traversals defined by $JJ = \gamma_1 + B_1$ will incur L1 cache misses of $\frac{WB_1}{\gamma C_{B_1}} * 2 + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}}$. Hence, we have the following:

- The L1 cache misses in all the tile traversals defined by $JJ \leq \gamma_2 - B_1$ amount to $\frac{WB_1}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma - B_1}{B_1} \rfloor) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma - 2B_1}{B_1} \rfloor)$.

- The L1 cache misses in all the tile traversals defined by $\gamma_2 - B_1 < JJ \leq \gamma_2$ amount to $\frac{WB_1}{\gamma C_{B_1}} * \lfloor \frac{\gamma_2}{B_1} \rfloor + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}} * \lfloor \frac{\gamma_2 - B_1}{B_1} \rfloor$.

- The L1 cache misses in all the tile traversals defined by $\gamma_2 < JJ$ amount to $\frac{WB_1}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma - B_1}{B_1} \rfloor) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma - 2B_1}{B_1} \rfloor)$.

Adding up the three numbers of the above, the total L1 cache misses in the tiled loop nest approximate $\frac{w_{S_1}(ITMAX-1)}{\gamma C_{B_1} B_1} + \frac{w_{S_2}(ITMAX-1)}{\eta C_{B_1}}$.

**Case 2: $\gamma < (ITMAX-1) * S_1$.**

This case is illustrated by Figure 4(b). Similar to the computation in Case 1, we have the following:

- The L1 cache misses in all the tile traversals defined by $JJ \leq \gamma_2$ amount to $\frac{WB_1}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma}{B_1} \rfloor) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma - B_1}{B_1} \rfloor)$.

- The L1 cache misses in all the tile traversals defined by $\gamma_2 < JJ \leq (ITMAX-1) * B_1 + \gamma_1$ amount to $\frac{WB_1}{\gamma C_{B_1}} * \lfloor \frac{\gamma}{B_1} \rfloor * \lfloor \frac{(ITMAX-1) * S_1 - \gamma}{B_1} \rfloor + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}} * \lfloor \frac{\gamma}{B_1} \rfloor * \lfloor \frac{(ITMAX-1) * S_1 - \gamma}{B_1} \rfloor$.

- The L1 cache misses in all the tile traversals defined by $(ITMAX-1) * B_1 + \gamma_1 < JJ$ amount to $\frac{WB_1}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma}{B_1} \rfloor) + \tau(B_1 - S_1)S_2 * \frac{B_1}{S_1} * \frac{W}{\gamma C_{B_1}} * (1 + 2 + \ldots + \lfloor \frac{\gamma - B_1}{B_1} \rfloor)$.

Adding up the three numbers of the above, the total L1 cache misses in the tiled loop nest approximate $\frac{w_{S_1}(ITMAX-1)}{\gamma C_{B_1} B_1} + \frac{w_{S_2}(ITMAX-1)}{\eta C_{B_1}}$.

**Case 3: $\gamma > (ITMAX-1) * S_1$.**

Similar to Case 2, the total L1 cache misses in the tiled loop nest approximate $\frac{w_{S_1}(ITMAX-1)}{\gamma C_{B_1} B_1} + \frac{w_{S_2}(ITMAX-1)}{\eta C_{B_1}}$. 

Figure 4: Calculating cache misses under different scenarios

(a) $TT$ (b) $TT$ (c) $TT$
Combining the above three cases and plugging in the estimate of \( \tau \), the total number of L1 cache misses is approximately

\[
\frac{WS_1(ITMAX-1)}{C_{b1}B_1} + \frac{WS_2(ITMAX-1)}{C_{b2}B_2}.
\]  
(1)

Similarly, with Properties 1 and 2 standing, the number of L2 cache misses for 2-D tiling is approximately

\[
\frac{WS_1(ITMAX-1)}{C_{b2}B_1} + \frac{WS_2(ITMAX-1)}{C_{b2}B_2}.
\]  
(2)

### 4.2 Estimation of Cache Misses for Option 2

Similar to the derivation in Section 4.1, for 1-D tiling with the L1 cache targeted, the total number of L1 cache misses is approximately

\[
\frac{WS_1(ITMAX-1)}{C_{b1}B_1}.
\]  
(3)

The total number of cache misses for the L2 cache is approximately

\[
\frac{WS_1(ITMAX-1)}{C_{b2}B_1}.
\]  
(4)

### 4.3 Estimation of Cache Misses for Option 3

Similar to the derivation in Section 4.1, for 2-D tiling with the L2 cache targeted, the total number of cache misses for the L2 cache is approximately

\[
\frac{WS_1(ITMAX-1)}{C_{b2}B_1} + \frac{WS_2(ITMAX-1)}{C_{b2}B_2}.
\]  
(5)

To estimate the number of L1 cache misses, we need to consider both the temporal reuses across different tiles and the group reuses [21] within the same tile. With the L2 cache targeted, the L1 cache temporal locality may not be exploited between different tiles. The locality due to certain group reuses, however, may be exploited on the L1 cache. For example in Figure 1(c), there exist group reuses between array references \( A(I+1,J) \) and \( A(I,J) \), between \( A(I,J-1) \) and \( A(I,J) \), etc. All the group reuses with a reuse distance [21] less than the L1 cache block size are very likely to be exploited due to spatial locality. (The reuse between \( A(I,J) \) and \( A(I+1,J) \) is an example.) Therefore, we assume that all such group reuses will result in L1 cache hits.

For group reuses with a distance greater than the L1 cache block size, we need to separate two cases. If \( n_a = 1 \), these group reuses may still result in cache hits since no cross-interference misses occur. For example, the group reuse between \( A(I,J-1) \) and \( A(I,J) \) in Figure 1(c) is very likely to generate L1 cache hit for array reference \( A(I,J) \). If \( n_a > 1 \), some of these group reuses may not generate cache hits because of cross-interference misses. Note that inter-array padding is applied to exploit temporal locality for the L2 cache across different tiles. The padding sizes computed by inter-array padding may not guarantee the L1 cache locality due to those group reuses.

Suppose \( n_a > 1 \). Let \( g \) be the number of group reuses with a distance greater than the L1 cache block size. If we assume all \( g \) group reuses result in cache hits, the total number of cache misses for the L1 cache will be

\[
ITMAX \times \frac{W}{C_{b1}}.
\]  
(6)
If we assume all \( g \) group reuses result in cache misses, the total number of cache misses for the L1 cache will be

\[
(1 + \frac{g}{n_a}) \cdot ITMAX \frac{W}{C_{b1}}. \tag{7}
\]

In this paper, if \( n_a > 1 \), we choose to assume that half the \( g \) group reuses generate L1 cache misses. Let \( \alpha \) be the number of group reuses which will result in cache misses. Based on the above discussion, we have the following:

\[
\alpha = \begin{cases} 
0 & \text{if } n_a = 1 \\
0.5 \cdot g & \text{otherwise}
\end{cases}
\]

The number of L1 cache misses can hence be estimated as

\[
(1 + \frac{\alpha}{n_a}) \cdot ITMAX \frac{W}{C_{b1}}. \tag{8}
\]

### 4.4 Estimation of Cache Misses for Option 4

Due to the different options presented above, for 1-D tiling with the L2 cache targeted, the total number of cache misses for the L1 cache is approximately

\[
(1 + \frac{\alpha}{n_a}) \cdot ITMAX \frac{W}{C_{b1}}. \tag{9}
\]

The total number of L2 cache misses is approximately

\[
\frac{WS_1(ITMAX-1)}{C_{b2}B_1}. \tag{10}
\]

### 5 Tile-Size Selection

In this section, we first present an execution cost model for tiling with a given tile size, based on both the number of cache misses and the loop overhead. We then present our tile-size selection algorithm, STS. Furthermore, we develop an execution cost model for GroupPad [13] and theoretically compare it with inter-array padding. Finally, we give a running example to go through our algorithm.

#### 5.1 An Execution Cost Model for Tiling

Loop tiling introduces loop overhead. To decide between 1-D tiling and 2-D tiling, the overhead of the tiled \( I_i \) loops in Figure 2(c) needs to be measured. Let \( n_1 \) be the sum of the static number of instructions for the computation of all the \( I_i \) loop bounds \((1 \leq i \leq m)\). The \( I_i \) loop overhead due to 2-D tiling in terms of the dynamic count of instructions, is measured approximately by

\[
n_1 \frac{ITMAX \cdot \gamma \cdot \eta}{B_2}. \tag{11}
\]

Let \( n_2 \) be the sum of the static number of instructions for the computation of all the \( J_i \) loop bounds \((1 \leq i \leq m)\). The loop overhead due to tiled \( J_i \) loops can be measured by

\[
n_2 \frac{ITMAX \cdot \gamma}{B_1}. \tag{12}
\]
Let \( n_3 \) be the sum of the static number of instructions in the \( I_i \) \((1 \leq i \leq m)\) loop bodies. The dynamic instruction count for the \( I_i \) loop bodies is

\[
n_3 \times \text{ITMAX} \times \gamma \times \eta. \tag{13}
\]

From (11) and (13), if \( n_1 \) and \( n_3 \) are approximately equal, then a small \( B_2 \) will introduce large loop overhead.

Similar to the classification for estimating cache misses, we have four different cases for execution cost estimation. By summing up the cache misses and the loop overhead in Section 4 and in the above respectively, we can model the execution cost as follows:

- **Execution Cost Estimation for Option 1:**

\[
p_1 \left( \frac{w_{S_1}(\text{ITMAX}-1)}{C_{b_1}B_1} + \frac{w_{S_2}(\text{ITMAX}-1)}{C_{b_2}B_2} \right) + p_2 \left( \frac{w_{S_1}(\text{ITMAX}-1)}{C_{b_2}B_1} + \frac{w_{S_2}(\text{ITMAX}-1)}{C_{b_2}B_2} \right) + n_1 \frac{\text{ITMAX} \times \gamma \times \eta}{B_2} + n_2 \frac{\text{ITMAX} \times \gamma}{B_1}. \tag{14}
\]

- **Execution Cost Estimation for Option 2:**

\[
p_1 \frac{w_{S_1}(\text{ITMAX}-1)}{C_{b_1}B_1} + p_2 \frac{w_{S_1}(\text{ITMAX}-1)}{C_{b_2}B_1} + n_2 \frac{\text{ITMAX} \times \gamma}{B_1}. \tag{15}
\]

- **Execution Cost Estimation for Option 3:**

\[
p_1 \left( 1 + \frac{\alpha}{n_a} \right) \frac{\text{ITMAX} \times \frac{W}{C_{b_1}}}{B_1} + p_2 \left( \frac{w_{S_1}(\text{ITMAX}-1)}{C_{b_2}B_1} + \frac{w_{S_2}(\text{ITMAX}-1)}{C_{b_2}B_2} \right) + n_1 \frac{\text{ITMAX} \times \gamma \times \eta}{B_2} + n_2 \frac{\text{ITMAX} \times \gamma}{B_1}. \tag{16}
\]

- **Execution Cost Estimation for Option 4:**

\[
p_1 \left( 1 + \frac{\alpha}{n_a} \right) \frac{\text{ITMAX} \times \frac{W}{C_{b_1}}}{B_1} + p_2 \frac{w_{S_1}(\text{ITMAX}-1)}{C_{b_2}B_1} + n_2 \frac{\text{ITMAX} \times \gamma}{B_1}. \tag{17}
\]

Note that \( B_2 \) and \( S_2 \) do not appear in formulas (15) and (17). For 1-D tiling, targeting the L1 cache will produces a smaller \( B_1 \) than targeting the L2 cache. From the above formulas, 1-D tiling is preferable to 2-D tiling under two circumstances: 1) if the skewing factor \( S_2 \) is so large that the tile height \( B_2 \) in 2-D tiling must be maximized in order to reduce the cache misses; and 2) \( n_1 \eta \) is so large that \( B_2 \) must be maximized in order to reduce the loop overhead. In either case, it is simply preferable that 2-D tiling degenerates to 1-D tiling.

Other than the above observation, the optimal tile size is not immediately clear from the formulas, due to the constraints of Properties 1 and 2. In the next, we develop our STS algorithm.

### 5.2 Tile-Size Selection Algorithm

In this section, we first discuss how to preserve Properties 1 and 2. We then present our tile-size selection algorithm.
Procedure `EnumFPSizc(C_s, C_b, N)`

for \( F_2 \leftarrow 1 \) to \( N \) do
  \( F_1 \leftarrow 1 \)
  \( t \leftarrow (F_1 \cdot N) \mod C_s \)
  while \( ((F_2 + C_b - 1) \leq t \leq (C_s - F_2 - C_b + 1)) \)
    Record \((F_1, F_2)\)
    \( F_1 \leftarrow F_1 + 1 \)
    \( t \leftarrow (F_1 \cdot N) \mod C_s \)
  end while
end for

(a)

Figure 5: Procedure `EnumFPSizc` and an illustration of utilizing portions of the cache by a single tile.

5.2.1 Preserving Property 1

First, we discuss how to eliminate self-interference misses within a single tile. For any array \( A_i \), let \( R \) be the minimum rectangular array region which contains all the \( A_i \) elements referenced within a tile \( t \). We say that \( A_i \)'s footprint size within tile \( t \) is \((F_1, F_2)\), where \( F_1 \) and \( F_2 \) are the numbers of columns and rows in \( R \) respectively. We call \( F_1 \) (\( F_2 \)) the array footprint width (height) for \( A_i \) within tile \( t \). Reversely, given a footprint size of \( A_i \), the tile size can also be computed. Given the subscript patterns and the loop bounds, such a computation is straightforward and we omit the details. For the example of SOR (Figure 1(c)), assuming the array footprint size for \( A \) to be \((\kappa_1, \kappa_2)\), the loop tile size should be \((\kappa_1 - 2, \kappa_2 - 2)\). For array \( A_i \), if the footprint height \( F_2 \) is greater than the distance between the locations of two columns in the cache, then the columns accessed within the tile will conflict in the cache, creating self-interference misses [1]. More precisely, we have the following lemma:

Lemma 1 Given array footprint size \((F_1, F_2)\) for any \( A_i \) \((1 \leq i \leq n_u)\), a cache of size \( C_s \) and cache line size \( C_b \), if there exist no self-interference misses, then the distance between the starting cache locations of any two columns of \( A_i \) within \( F_1 \) consecutive columns is either no smaller than \( F_2 \), or no greater than \( C_s - F_2 \). Conversely, there exist no self-interference misses if the distance between the starting cache locations of any two columns of \( A_i \) within \( F_1 \) consecutive columns is either no smaller than \( F_2 + C_b - 1 \), or no greater than \( C_s - F_2 - C_b + 1 \).

Proof Obvious. 

Given a directly-mapped cache of size \( C_s \) and cache line size \( C_b \), and given an array column size \( N \), procedure `EnumFPSizc` in Figure 5(a) enumerates all the footprint sizes \((F_1, F_2)\) which incur no self-interference misses, according to Lemma 1. We say that a footprint size \((F_1, F_2)\) of \( A_i \) is maximal if increasing either \( F_1 \) or \( F_2 \) will introduce self-interference misses for \( A_i \). In general, the maximal footprint size for array \( A_i \) is not unique. According to `EnumFPSizc`, the maximal footprint sizes for all arrays are the same if they have the same array column sizes. Our tile-size selection scheme will enumerate all array footprint sizes which are free of self-interference misses until the sizes become maximal. The scheme estimates and compares the execution cost for different \((F_1, F_2)\) in order to get the optimal tile size.

Next, suppose the cache is not directly-mapped, and assume an LRU replacement policy.
array footprint

Figure 6: An illustration of padding to eliminate cross-interferences

We show that the parameter \( C_s \) in procedure \( \text{EnumFPSize} \) should not be the whole cache size. Otherwise, self-interference misses will occur when the execution proceeds from one tile to the next. For clarity, instead of arguing formally for the general cases, we illustrate the cases of 2-way and fully-associative caches. Figure 5(b) shows two consecutive tiles \( t1 \) and \( t2 \). Suppose \( C_s \) equals the whole cache size in procedure \( \text{EnumFPSize} \) and suppose the footprint size of \( t1 \) is maximal. Tile \( t1 \) accesses the cache from the least-recently referenced data segment to the most-recently referenced data segment in the memory, in the order of \( D1, D2, D3 \) and \( D4 \) which are separated by solid lines. If the cache associativity is \( C_{s1} = 2 \), then \( D2 \) and \( D4 \) will map to the same cache sets. The data accessed in the blank rectangle \( A \) will replace segment \( D2 \). If the cache is fully associative, \( D1 \) will be replaced. However, part of the old data in segment \( D2 \) (or \( D1 \)) could have been reused by tile \( t2 \). One solution to avoid the replacement of useful data is to reduce the footprint size within \( t1 \) such that only a portion of the cache is used to compute the maximal footprint size in \( \text{EnumFPSize} \). Figure 5(c) shows the case for two-way set-associative cache. In this way, the data accessed in Regions A and C will replace the cache segment \( D2 \) and part of segment \( D1 \), whose old data are not reused by \( t2 \). The reusable data in \( D3 \) will be kept in the cache. Using the above idea, we let \( C_s = \frac{C_{s1} - 1}{C_{s1}} \cdot C_{s1} \) in procedure \( \text{EnumFPSize} \), for 2-way and fully-associative caches. The cases of other associativities are more complex, and they will not be discussed in this paper.

To eliminate capacity misses, the footprint size of each array \( A_i \) can only be \( (\frac{F_1}{n_a}, F_2) \), a fraction of \( (F_1, F_2) \). Here, we choose to partition columns instead of rows, in order to preserve spatial locality. Assume that \( (B_1^{(i)}, B_2^{(i)}) \), \( 1 \leq i \leq n_a \), is the tile size such that the footprint size for array \( A_i \) within a single tile is \( (\frac{F_1}{n_a}, F_2) \). For 2-way and fully-associative caches, we choose the tile size for the tiled loop as \( (B_1, B_2) = (\min_i B_1^{(i)}, \min_i B_2^{(i)}) \). For directly-mapped caches, we choose \( (B_1, B_2) = (\min_i B_1^{(i)} - S_1, \min_i B_2^{(i)} - S_2) \). One can prove that for directly-mapped, 2-way and fully-associative caches, Property 1 holds under the above treatment. For other set-associative caches, procedure \( \text{EnumFPSize} \) needs to be revised.

5.2.2 Preserving Property 2

We apply inter-array padding to eliminate cross-interference misses within a tile traversal. For simplicity of presentation, we assume that the array subscript patterns of one particular array \( A_k \) cover all the array subscript patterns for all the other arrays \( A_i \), \( i \neq k \). The discussion in this section can be easily extended if such an assumption does not hold. Using inter-array padding, we let the starting addresses for array \( A_i(1 \leq i \leq n_a) \) map to the same location in the cache as the starting address of the \( (\frac{F_1}{n_a} \times (i - 1)) \)th column of array \( A_1 \). With such padding, cross-interference misses are eliminated within a single tile between \( A_i \) and \( A_j \) \( (1 \leq i, j \leq n_a, i \neq j) \).

When the execution goes from one tile to the next, if the cache is directly-mapped, the newly accessed data for \( A_i \) will map to cache locations previously unused in the tile traversal. If the cache is not directly-mapped, the newly accessed data for \( A_i \) will map to cache locations which are
Input: $S_1, S_2, C_{a1}, C_{a2}, C_{b1}, C_{b2}, \alpha_1, \alpha_2, \alpha, N$ (see Table 2).
Output: The size $(B_1, B_2)$ and the transformed array declaration.

Procedure:

/* $M$ is the minimum execution cost */
$M \leftarrow \infty$
if ($C_{a1} = 1$) then $C_{a1} \leftarrow \frac{C_{a1} - 1}{2} C_{a1}$ end if
if ($C_{a2} = 1$) then $C_{a2} \leftarrow \frac{C_{a2} - 1}{2} C_{a2}$ end if
ComputeTileSize-2D($C_{a1}, C_{a2}, C_{b1}$) /* Option 1 */
ComputeTileSize-1D($C_{a1}, C_{b1}$) /* Option 2 */
ComputeTileSize-2D($C_{a2}, C_{b2}, C_{b1}$) /* Option 3 */
ComputeTileSize-1D($C_{a2}, C_{b2}$) /* Option 4 */
Apply inter-array padding [17].
Return $(B_1, B_2)$.

Procedure ComputeTileSize-1D($C_{a}, C_{b}$)
/* Targeting the cache with the size $C_{a}$. $(TB_1, TB_2)$ is a temporary tile size. */
Select the maximum tile width $k$ such that the footprint of one tile can fit in the cache with size $C_{a}$.
$TB_1 \leftarrow k - S_1$
$TB_2 \leftarrow \eta + S_2 \times (TMAX-1)$
Compute the execution cost, $TM$, based on (15) for Option 2 or (17) for Option 4.
if ($TM < M$) then $B_1 \leftarrow TB_1, B_2 \leftarrow TB_2, M \leftarrow TM$ end if

Procedure ComputeTileSize-2D($C_{a1}, C_{a2}, C_{b1}$)
/* Targeting the cache with the size $C_{a}$. $(TB_1, TB_2)$ is a temporary tile size. */
for $F_2 \leftarrow C_{b1}$ to $N$ do
$F_1 \leftarrow 1$
$t \leftarrow (F_1 \times N) \mod C_{a}$
while $((F_1 + C_{b1} - 1) \leq (C_{a} - F_1 - C_{b1} + 1))$ do
Convert array footprint size ($F_1, F_2$) to loop tile size $(TB_1, TB_2)$ [17].
if ($C_{a2} = 1$) then $TB_1 \leftarrow TB_1 - S_1, TB_2 \leftarrow TB_2 - S_2$ end if
if ($TB_1 > 0$ and $TB_2 > 0$) then
Compute the execution cost, $TM$, based on (14) for Option 1 or (16) for Option 3.
if ($TM < M$) then $B_1 \leftarrow TB_1, B_2 \leftarrow TB_2, M \leftarrow TM$ end if
end if
$F_1 \leftarrow F_1 + 1$
$t \leftarrow (F_1 \times N) \mod C_{a}$
end while
end for

Figure 7: Tile-size selection algorithm - STS

either previously unused or will not be referenced again within the current traversal. Therefore, cross-interference misses are also eliminated within a tile traversal. Figure 6 illustrates an example for $F_1 = 4$ and $n_0 = 2$, where the cache is directly mapped. Here, assuming the starting address for array $A_1$ to be 0, the padded number of data items, $x$, between arrays $A_1$ and $A_2$ can be determined from

$$\text{size}(A_1) + x = (2 \times N), \mod C_{a1}. \quad (18)$$

We are ready to present our tile-size selection algorithm in the next section.

5.2.3 Algorithm STS

Algorithm STS in Figure 7 selects the tile size by interleaving the operations in procedure EnumFP-Size with the applications of Formulas (17) and (14) which compute the execution cost. We require $B_2$ to be no smaller than the cache line size $C_{b1}$. However, we do not require $B_2$ to be a multiple of $C_{b1}$, since such a requirement does not have much benefit when execution proceeds from one tile to the next.

STS makes the decision between 1-D and 2-D tiling based on their execution cost. For 1-D tiling, ComputeTileSize-1D tries to find tile width $B_1$ such that Properties 1 and 2 are preserved on the
L2 cache and that Formula (17) is minimized. For 2-D tiling, ComputeTileSize-2D enumerates all tile sizes which are free of self-interference misses. The tile size with the lowest execution cost is selected. Between 1-D and 2-D tiling, the scheme with the lower execution cost is chosen.

Here, we want to specially mention the conversion from array footprint size \((F_1, F_2)\) to loop tile size \((TB_1, TB_2)\) in Procedure ComputeTileSize-2D in Figure 7. We have

\[
TB_1 = \left\lfloor \frac{F_1}{n_a} \right\rfloor - c,
\]

where \(n_a\) is the number of arrays within the T-loop body and \(c\) is a small nonnegative constant to adjust loop tile size in order to avoid capacity misses. If \(n_a\) is large, it is very likely that \(TB_1\) is no greater than \(S_1\). As the result, the \((F_1, F_2)\) pair will bear no effect on the final tile-size selection.

### 5.3 Inter-Array Padding and GroupPad

In the STS algorithm in Figure 7, after all four options are evaluated, one optimal tile size with the associated targeted cache is determined. Inter-array padding is then applied to satisfy Property 2 on the targeted cache.

For 1-D tiling, if the array column size is large, it is often difficult to exploit the L1 temporal locality. For example in Figure 1(b), if \(N\) is greater than the L1 cache size, the L1 temporal locality across the loop \(T\) cannot be exploited because no legal tile size can be found satisfying Properties 1 and 2 on the L1 cache. For 1-D tiling, if Option 4 is chosen, then the tile size should clearly be chosen to make the footprint fit the L2 cache. We need to justify, however, why we choose to perform padding for the L2, but not the L1, cache. Inter-array padding can preserve Property 2 on the L2 cache. However, due to the large footprint for the L1 cache, inter-array padding will not be able to help preserve the group reuses for the L1 cache. Alternatively, we can apply another padding algorithm, GroupPad, proposed by Rivera and Tseng [13], to exploit the L1 cache locality. The padding sizes generated by the two algorithms are often not the same, unfortunately. Suppose that, after applying GroupPad, the locality due to group reuses can be fully exploited on both the L1 and the L2 cache misses. On the other hand, no temporal locality across different tiles is exploited. For 1-D tiling with the L2 cache targeted, we will have an execution estimation, after GroupPad, as follows:

\[
p_1 \times (ITMAX \times \frac{W}{C_{b1}}) + p_2 \times (ITMAX \times \frac{W}{C_{b2}}) + \frac{ITMAX \times \gamma}{B_1},
\]

In order to exploit the L2 cache locality, \(B_1\) is to be chosen as large as possible. On the other hand, both \(\alpha\) and \(S_1\) are small. Comparing (17) with (20), it is thus clear that inter-array padding is preferred in the case of Option 4.

### 5.4 A Running Example

We now take SOR (Figure 1) as an example to show how STS works, assuming the following parameters: \(N = 1000\), \(ITMAX = 1050\), \(C_{b1} = 4096\), \(C_{b1} = 4\), \(C_{a1} = 2\), \(C_{a2} = 128 \times 1024\), \(C_{b2} = 16\), \(C_{a2} = 2\), \(n_1 = n_2 = 15\), \(p_1 = 6\), \(p_2 = 30\), and \(\alpha = 0\). Based on the array subscripts and the loop bounds, we have \(S_1 = S_2 = 1\), \(\gamma = \eta = 999\) and \(W = N \times N = 1000000\).

In the following, we show the steps of STS.

- Since \(C_{b1} = 2\), \(C_{a1} = \frac{C_{b1}}{2} = 2048\).
- Since \(C_{b2} = 2\), \(C_{a2} = \frac{C_{b2}}{2} = 64 \times 1024\).
Table 3: Machine parameters

<table>
<thead>
<tr>
<th>Processors</th>
<th>( C_{a1} )</th>
<th>( C_{b1} )</th>
<th>( C_{a2} )</th>
<th>( C_{b2} )</th>
<th>( p_1 )</th>
<th>( p_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra II</td>
<td>2K</td>
<td>2</td>
<td>256K</td>
<td>8</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>R10K</td>
<td>4K</td>
<td>2</td>
<td>512K</td>
<td>16</td>
<td>9</td>
<td>48</td>
</tr>
</tbody>
</table>

- ComputeTileSize-2D\((C_{a1}, C_{b1}, C_{a2})\) is called. After this call, we will have: \( B_1 = 38 \), \( B_2 = 43 \) and \( M = 541048960 \).

- ComputeTileSize-1D\((C_{a1}, C_{b1})\) is called. During this call, no legal tile sizes can be generated because of large \( N \). Therefore, we still have the same optimal tile size and execution cost as in the previous step.

- ComputeTileSize-2D\((C_{a2}, C_{b2}, C_{a2})\) is called. Although this call is able to generate legal tile sizes, after this call, the tile size and execution cost in the previous step remain optimal.

- ComputeTileSize-1D\((C_{a2}, C_{b2})\) is called. This call generates \( TB_1 = 63 \), \( TB_2 = 2048 \) and \( TM = 1608043520 \). Since \( TM > M \), the tile size and execution cost in the previous step remain optimal.

- Until this point, STS has chosen 2-D tiling with the L1 cache targeted. The optimal tile size is \((38, 43)\).

- No inter-array padding is applied since \( n_a = 1 \).

6 Experimental Evaluation

In this section, we use experimental results from several benchmark programs to compare different tile-size selection algorithms. In Section 6.1, we describe how we set up the experiments. In Section 6.2, we examine the results from benchmarks whose loops can be tiled at two levels. All tile-size selection algorithms are compared for these benchmarks. We evaluate the impact of loop overhead in Section 6.3. In Section 6.4, we justify our decision which drops the consideration of software pipelining and the TLB, compared with [17]. In Section 6.5, we examine the results from two programs, tomatv and swim, from the well-known SPEC benchmark suites. The loops in these two programs can be tiled at one level only, allowing a comparison among STS, TSS and TLI only. We also compare our inter-array padding algorithm with the GroupPad [13] for the tiled codes. Further, we exhaustively test different tile sizes to see how close STS is from the optimal tile size. In Section 6.6, we explain several subtle points concerning the results.

6.1 Experimental Setup

In this section, we use experimental results from several benchmark programs to compare different tile-size selection algorithms. In Section 6.1, we describe how we set up the experiments. In Section 6.2, we examine the results from benchmarks whose loops can be tiled at two levels. All tile-size selection algorithms are compared for these benchmarks. We evaluate the impact of loop overhead in Section 6.3. In Section 6.4, we justify our decision which drops the consideration of software pipelining and the TLB, compared with [17]. In Section 6.5, we examine the results from two programs, tomatv and swim, from the well-known SPEC benchmark suites. The loops in these two programs can be tiled at one level only, allowing a comparison among STS, TSS and TLI only. We also compare our inter-array padding algorithm with the GroupPad [13] for the tiled codes. Further, we exhaustively test different tile sizes to see how close STS is from the optimal tile size. In Section 6.6, we explain several subtle points concerning the results.

6.1 Experimental Setup

We have implemented a stand-alone version of STS based on Figure 7. We have previously implemented our skewed tiling algorithm in Panorama [16], which is a Fortran source-to-source compiler. However, some special parameters to STS, for example, \( n_1 \) and \( n_2 \), can be easily obtained only within a compiler backend. Therefore, we have not integrated the STS into Panorama. Currently, we obtain these special parameters by examining the assembly code of the original program.
We apply STS to three numerical kernels, SOR, Jacobi and Livermore Loop No. 18 (LL18), and two SPEC benchmarks, tomcatv and swim. These benchmarks are chosen because they require skewed tiling. We use reference inputs for tomcatv and swim. For SOR, Jacobi and LL18, we declare $N \times N$ double precision arrays, with randomly chosen $N$ based on a random number generator [12] with the following formula

$$z_{n+1} = (16807z_n) \mod 2147483647.$$  

Assuming that the array sizes under consideration range from $r_0$ to $r_1$, we select 200 array sizes, $a_n$, such that

$$a_n = r_0 + (z_n \mod (r_1 - r_0)), 1 \leq n \leq 200.$$  

We use $z_1 = 9$ in all our experiments. Note that it would be too time-consuming to exhaustly test all array sizes within the range in our experiments.

We run the test programs on a SUN Ultra II uniprocessor workstation and on one MIPS R10K processor of an SGI Origin 2000 multiprocessor, with the tile sizes selected by five different algorithms, namely, STS, TLI [1], TSS [2], LRW [7] and DAT [11]. In order to handle several equally-important arrays, we make an obviously necessary modification on the original TSS and LRW algorithms such that the value of the initial tile size will meet the working set constraint. We also modify the TLI algorithm such that only the cache size divided by the number of equally-important arrays is used to compute the tile sizes which are free of self-interference misses. For 1-D tiling, we also extend TSS and TLI to consider the L2 cache if the L1 cache size is too small for locality enhancement in that case. If any algorithm decides to choose the whole array column as the tile height, then we let $B_2 = \eta + S_2 \ast (ITMAX-1)$ and tile the $J_i$ loops only (Figure 2(b)).

Table 3 lists the machine parameters for the Ultra II and the R10K, assuming the size of an array element of 8 bytes. The main memory size for the Ultra II is 128M bytes, and it is 16G bytes for the R10K, in which 1G bytes are local to the processor. To accommodate the competition between instructions and data in the L2 cache, we only tries to utilize 95% of the total L2 cache capacity. We use the machine counters on both machines to measure the cache miss rate.

On the R10K, the untiled codes are compiled using the native compiler with the "-O3" optimisation switch set. On the R10K, we found that compiling the tiled code with the "-O2" switch can sometimes run faster than that with the "-O3" switch, regardless of the tile-size selection schemes. Therefore, we compile the tiled code with "-O2" or "-O3" depending on which produces shorter execution time. For all the tile-size selection schemes, we switch off loop tiling for the native compiler on the R10K when we compile the tiled source programs (with for both 1-D and 2-D tiling). We switch off prefetching on the R10K when we compile 2-D tiled source codes since prefetching may increase cross-interference misses for smaller tile height $B_2$. We also switch off common block reorganization since the tile size selection algorithms already take care of memory layout. On the Ultra II, both the untiled and the tiled codes are compiled using the native compiler with the "-fast -xchip=ultra2 -xarch=v8plusa -simple=2" optimization switch, which is recommended by the vendor.

### 6.2 Loops Which Can Be Tiled at Two Levels

In this subsection, we compare different tile-size selection schemes through three numeric kernels, SOR, Jacobi and LL18.

#### The SOR kernel

The SOR kernel is a perfectly-nested loop as shown in Figure 1(a). We fix $ITMAX$ to 1050 and randomly choose 200 array sizes ranging from 200 to 2000, i.e., $(r_0, r_1) = (200, 2000)$ in
Figure 8: Execution time of SOR for various schemes on the Ultra II

Figure 9: L1 cache miss rate of SOR for various schemes on the Ultra II

Figure 10: L2 cache miss rate of SOR for various schemes on the Ultra II
Figure 11: Execution time of SOR for various schemes on the R10K

Figure 12: L1 cache miss rate of SOR for various schemes on the R10K

Figure 13: L2 cache miss rate of SOR for various schemes on the R10K
Table 4: Speedup by STS and average cache miss rates over different schemes for SOR

<table>
<thead>
<tr>
<th></th>
<th>Ultra II</th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TLI</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup by STS</td>
<td>1.05</td>
<td>1.01</td>
<td>1.27</td>
<td>0.99</td>
<td>1.00</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>L1 Miss Rate</td>
<td>0.14</td>
<td>0.03</td>
<td>0.07</td>
<td>0.03</td>
<td>0.02</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>0.006</td>
<td>0.006</td>
<td>0.009</td>
<td>0.005</td>
<td>0.006</td>
<td>0.008</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R10K</th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TLI</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup by STS</td>
<td>1.30</td>
<td>1.02</td>
<td>1.09</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>L1 Miss Rate</td>
<td>0.113</td>
<td>0.024</td>
<td>0.012</td>
<td>0.006</td>
<td>0.031</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>0.057</td>
<td>0.030</td>
<td>0.043</td>
<td>0.085</td>
<td>0.007</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: Execution time of Jacobi for various schemes on the Ultra II

Equation (22). We have $S_1 = S_2 = 1$ and $\alpha = 0$. We have $n_1 = n_2 = 11$ on the R10K and $n_1 = n_2 = 22$ on the Ultra II. On the Ultra II, STS chooses 1-D tiling for 7 test cases and chooses 2-D tiling for 193 test cases. On the R10K, STS chooses 1-D tiling for 9 test cases and 2-D tiling for 191 test cases. All test cases of 1-D tiling on both machines target the L2 cache.

Table 4 summarizes the average speedup by STS over other schemes, average L1 and L2 cache miss rates for SOR on both the Ultra II and the R10K. The execution time is averaged by geometric mean, and the cache miss rates are averaged by arithmetic mean of cache miss rates for individual array size. On the Ultra II, STS performs slightly worse than TLI by 1%. On the R10K, it performs equally or better than all other schemes. On the Ultra II, TSS often chooses a rectangular tile size with a small $B_2$. Such a tile size introduces a high loop overhead, which slows down the program execution.

From Table 4, STS has the lowest average L1 cache miss rate, but it exhibits the worst L2 cache miss rate on the R10K. When combined with the low L1 cache miss rate, however, STS still outperforms the others on the R10K, with a largest speedup of 1.09 over TSS.

Figures 8 and 11 show the execution time for various schemes on the Ultra II and on the R10K respectively. Figures 9 and 10 show the L1 cache and L2 cache miss rates respectively on the Ultra II. Figures 12 and 13 show the L1 cache and L2 cache miss rates respectively on the R10K.

The Jacobi Kernel

The Jacobi kernel is imperfectly nested such that the loop $T$ contains two perfect nests ($m = 2$ in Figure 2(a)). The $T$-loop body contains two arrays after tiling [16]. We fix $TMAX$ to 500 and randomly choose 200 array sizes ranging from 200 to 2000. We have $S_1 = S_2 = 1$ and $\alpha = 1.0$. We have $n_1 = n_2 = 17$ on the R10K and $n_1 = n_2 = 28$ on the Ultra II. On the Ultra II, STS chooses 1-D tiling for 11 test cases and chooses 2-D tiling for 189 test cases. On the R10K, STS chooses
Figure 15: L1 cache miss rate of Jacobi for various schemes on the Ultra II

Figure 16: L2 cache miss rate of Jacobi for various schemes on the Ultra II

Figure 17: Execution time of Jacobi for various schemes on the R10K
1-D tiling for 19 test cases and 2-D tiling for 181 test cases. All test cases of 1-D tiling, except one on each machine, target the L2 cache.

Table 5 shows the average speedup by STS, average L1 and L2 cache miss rates for Jacobi on both the Ultra II and the R10K. On both machines, STS performs equally or better than all other schemes.

From Table 5, STS has the lowest average L1 cache miss rate except when compared with LRW on the R10K. However, STS achieves a good speedup over other schemes on both machines in most cases.

Figures 14 and 17 show the execution time of Jacobi for various schemes on the Ultra II and on the R10K respectively. Figures 15 and 16 show the L1 cache and L2 cache miss rates respectively on the Ultra II. Figures 18 and 19 show the L1 cache and L2 cache miss rates respectively on the R10K.

The LL18 Kernel

Similar to Jacobi, LL18 is also imperfectly nested and the loop \( T \) contains three perfect nests (\( m = 3 \) in Figure 2(a)). LL18 has 9 arrays, and the tiled version has 11 arrays after duplicating arrays \( ZR \) and \( ZZ \). Due to the relatively large number of arrays, the array sizes we used in SOR will produce extremely small tile sizes for all the tile-size selection schemes. Therefore, we reduce the array sizes and randomly choose 200 array sizes ranging from 200 to 500. We fix \( ITMAX \) to 300. We have \( S_1 = S_2 = 2 \) and \( \alpha = 5.0 \). We have \( n_1 = n_2 = 75 \) on the R10K and \( n_1 = n_2 = 87 \) on the Ultra II.
Figure 20: Execution time of LL18 for various schemes on the Ultra II

Figure 21: L1 cache miss rate of LL18 for various schemes on the Ultra II

Figure 22: L2 cache miss rate of LL18 for various schemes on the Ultra II
Figure 23: Execution time of LL18 for various schemes on the R10K

Figure 24: L1 cache miss rate of LL18 for various schemes on the R10K

Figure 25: L2 cache miss rate of LL18 for various schemes on the R10K
Table 5: Speedup by STS and average cache miss rates over different schemes for Jacobi

<table>
<thead>
<tr>
<th></th>
<th>Ultra II</th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TL1</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup by STS</td>
<td>5.14</td>
<td>1.35</td>
<td>2.07</td>
<td>1.22</td>
<td>1.00</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>L1 Miss Rate</td>
<td>0.60</td>
<td>0.12</td>
<td>0.24</td>
<td>0.24</td>
<td>0.06</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>0.15</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R10K</th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TL1</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup by STS</td>
<td>5.66</td>
<td>1.01</td>
<td>1.24</td>
<td>1.19</td>
<td>1.03</td>
<td>1.30</td>
<td></td>
</tr>
<tr>
<td>L1 Miss Rate</td>
<td>0.23</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.03</td>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>0.19</td>
<td>0.06</td>
<td>0.03</td>
<td>0.02</td>
<td>0.10</td>
<td>0.09</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Speedup by STS and average cache miss rates over different schemes for LL18

<table>
<thead>
<tr>
<th></th>
<th>Ultra II</th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TL1</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup by STS</td>
<td>2.04</td>
<td>2.35</td>
<td>2.83</td>
<td>2.18</td>
<td>1.00</td>
<td>2.35</td>
<td></td>
</tr>
<tr>
<td>L1 Miss Rate</td>
<td>0.43</td>
<td>0.21</td>
<td>0.28</td>
<td>0.32</td>
<td>0.47</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>0.11</td>
<td>0.03</td>
<td>0.06</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R10K</th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TL1</th>
<th>STS</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup by STS</td>
<td>1.83</td>
<td>2.11</td>
<td>2.11</td>
<td>1.72</td>
<td>1.00</td>
<td>1.80</td>
<td></td>
</tr>
<tr>
<td>L1 Miss Rate</td>
<td>0.17</td>
<td>0.07</td>
<td>0.09</td>
<td>0.12</td>
<td>0.21</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>L2 Miss Rate</td>
<td>0.12</td>
<td>0.04</td>
<td>0.07</td>
<td>0.01</td>
<td>0.04</td>
<td>0.02</td>
<td></td>
</tr>
</tbody>
</table>

Table 6 shows the average speedup by STS, average L1 and L2 cache miss rates for LL18 on both the Ultra II and the R10K. Note that \( S_2 = 2 \) suggests that we must have a large tile height \( B_2 \) to reduce the number of L1 cache misses for 2-D tiling. After tiling, the T-loop body contains 11 arrays. Such large number of arrays often make STS unable to eliminate the cache misses for 2-D tiling through padding (see Section 5.2.2).

STS correctly estimate the number of cache misses between 1-D and 2-D tiling, which is crucial to the determination of the final tile sizes. Out of 200 cases, STS chooses 1-D tiling in 186 cases on the Ultra II and in all 200 cases on the R10K. All test cases of 1-D tiling on both machines target the L2 cache. All the other tiling schemes either choose 2-D tiling or no tiling if they fail to generate the legal tile sizes. From Table 6, STS achieves a significant speedup over other schemes, ranging from 1.72 to 2.86 on both machines. Such a speedup is largely due to the more accurate modeling of the effects of loop skewing and due to the consideration of both L1 and L2 cache misses.

From Table 6, STS has the worst average L1 cache miss rates on both machines. This is expected because 1-D tiling tries to minimize the number of L2 cache misses. From Table 6, STS indeed has much smaller number of L2 cache miss rates than other schemes in most cases.

Figures 20 and 23 show the execution time of LL18 for various schemes on the Ultra II and on the R10K respectively. Figures 21 and 22 show the L1 cache and L2 cache miss rates respectively on the Ultra II. Figures 24 and 25 show the L1 cache and L2 cache miss rates respectively on the R10K.

Overall, for SOR and Jacobi, where the L1 cache locality can be exploited in most cases, STS achieves comparable results. For LL18 where the L2 cache locality is exploited in most cases, STS is significantly better.

6.3 Impact of Loop Overhead

In this subsection, we evaluate the impact of loop overhead through the benchmarks SOR, Jacobi and LL18. Table 7 shows the average speedup of STS over the schemes without consideration of
Figure 26: Comparison with and without loop overhead for SOR

Figure 27: Comparison with and without loop overhead for Jacobi

Figure 28: Comparison with and without loop overhead for LL18
Table 7: Speedup by STS over the STS without consideration of loop overhead

<table>
<thead>
<tr>
<th></th>
<th>Ultra II</th>
<th>R10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>1.60</td>
<td>1.60</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.60</td>
<td>1.60</td>
</tr>
<tr>
<td>LL18</td>
<td>1.60</td>
<td>1.60</td>
</tr>
</tbody>
</table>

Figure 29: Execution time reduction by STS compared with STS not considering loop overhead

loop overhead. Specifically, "STS" stands for the tiled codes with STS applied, "STS-loop" for the tiled codes with STS not considering loop overhead (i.e., considering cache misses only). Figure 26 shows the detailed results for SOR. Figure 27 shows the results for Jacobi. Figure 28 shows the results for LL18.

From Table 7, the impact of loop overhead seems marginal. In Figure 29, we further show the distribution of execution time reduction by STS, compared with STS not considering loop overhead. From Figure 29, in most cases, the performance difference remains in the range [-2%,2%]. However, out of total 1200 test cases on both machines, by considering loop overhead, STS improves the performance by equal to or greater than 10% in 71 cases, and hurts the performance by more than 10% in 38 cases.

Without considering loop overhead, STS tends to choose 2-D tiling targeting the L1 cache, even with a small $B_2$. Although such a small $B_2$ still tends to improve L1 cache locality, the corresponding loop overhead could offset that benefit. On the other hand, converting to 1-D tiling with the L2 cache targeted, sometimes caused by considering loop overhead, loses the ability to exploit L1 cache locality, and therefore may actually hurt the performance. Table 7 has shown the marginal gain (up to 4%) by considering loop overhead.

6.4 Impact of Software Pipelining and the TLB

In this subsection, we experimentally justify our decision to drop the consideration of software pipelining and the TLB, compared with [17].

Table 8 shows the speedup of STS over the STS with consideration of software pipelining. We can see that On the Ultra II, incorporating the software pipelining into the STS can improve the average performance for all three programs. However, the average performance drops for all three
Table 8: Speedup by STS over the STS with consideration of software pipelining

<table>
<thead>
<tr>
<th></th>
<th>STS</th>
<th>STS+sup</th>
<th>R10K</th>
<th>STS</th>
<th>STS+sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>1.00</td>
<td>0.94</td>
<td>SOR</td>
<td>1.00</td>
<td>1.03</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.00</td>
<td>0.97</td>
<td>Jacobi</td>
<td>1.00</td>
<td>1.03</td>
</tr>
<tr>
<td>LL18</td>
<td>1.00</td>
<td>0.98</td>
<td>LL18</td>
<td>1.00</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Table 9: Speedup of STS over the STS considering the TLB

<table>
<thead>
<tr>
<th></th>
<th>STS</th>
<th>STS+TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.00</td>
<td>0.96</td>
</tr>
<tr>
<td>LL18</td>
<td>1.00</td>
<td>1.11</td>
</tr>
</tbody>
</table>

programs on the R10K. The average performance gain with considering software pipelining is near to 0%.

Even with considering the TLB, the STS will generate the same tile size on the R10K [17]. However, it will generate smaller tile sizes on the Ultra II. Note that such smaller tile sizes will cause the under-utilization of the L2 cache. Table 9 shows the speedup of STS over the STS considering the TLB. We will further justify our decision in Section 6.6.

6.5 Loops Which Can Be Tiled at One Level Only

In this subsection, we evaluate the quality of STS for loops which can be tiled at one level only, using two SPEC benchmarks, tomcatv and swim. These two benchmarks can be tiled at one loop level only. Furthermore, the L2 cache should be targeted because of the large number of arrays within the loop body and the large array column size. We also evaluate the impact of padding through these two benchmarks.

tomcatv

We use two different reference inputs for tomcatv from SPEC92 and SPEC95 respectively. To verify whether STS produces nearly the best results, we run through a range of tile sizes, from 2

![Figure 30: Performance of tomcatv with different tile sizes on the Ultra II](image-url)
Figure 31: Performance of tommc3 with different tile sizes and without padding on the Ultra II

Figure 32: Performance of tommc3 with different tile sizes on the R10K

Figure 33: Performance of tommc3 with different tile sizes and without padding on the R10K
to three times of the size selected by STS, for each version of tomcatv. Figures 30(a) and (b) show the results on the Ultra II, where the vertical bar indicates the tile size selected by the STS. The original programs from SPEC92 and SPEC95 run 5 and 174 seconds respectively on the Ultra II, and 4.0 and 115.0 seconds respectively on the R10K. Figures 32(a) and (b) show the results on the R10K. Figures 34(a) and (b) show the results on the R10K. The results chosen by STS are at most 5% worse when compared with the optimal solutions for the enumerated tile sizes for both versions of the codes on both machines. To examine how padding affects the STS, we also run both versions of tomcatv without padding applied. Figures 31(a) and (b) show the results on the Ultra II, and Figures 33(a) and (b) show the results on the R10K. Except few cases, padded version runs significantly faster than unpadded version, which demonstrates the effectiveness of padding for STS.

**swim**

On the R10K, we use three different reference inputs for swim from SPEC92, SPEC95 and SPEC2000 respectively. On the Ultra II, however, because of the large data set size and the relative small main memory size, the SPEC2000 version of swim cannot be tiled with a positive tile size, i.e., it cannot be tiled profitably. Hence, on the Ultra II, we use two different reference inputs from SPEC92 and SPEC95 respectively. Similar to tomcatv, we choose the tile sizes from 2 to three times of

![Figure 34: Performance of swim with different tile sizes on the Ultra II](image)

![Figure 35: Performance of swim with different tile sizes and without padding on the Ultra II](image)
the size selected by STS for each version of swim. On the Ultra II, the original programs from SPEC92 and SPEC95 run 36 and 157 seconds respectively. On the R10K, the original programs from SPEC92, SPEC95 and SPEC2000 run 21.2, 91.9 and 1156 seconds respectively. Figures 34(a) and (b) show the results on the Ultra II, and Figures 36(a), (b) and (c) show the results on the R10K. When compared with the optimal solutions for the enumerated tile sizes for all versions of the codes on both machines, the results chosen by STS are within 5% worse, except that for SPEC92 swim on the Ultra II the performance degrades by 13%. The tiled swim requires several scalars expanded to 1-D arrays, which are ignored by our model (see Section 3). We suspect the performance degradation is due to the cross-interference misses between 2-D arrays and these ignored 1-D arrays. Figures 35(a) and (b) show the results on the Ultra II for unpadded versions of swim, and Figures 37(a), (b) and (c) show the results on the R10K. Similar to tomcatv, padded version runs faster than unpadded version in most cases for SPEC92 and SPEC95.

TLI and TSS can also be applied to tomcatv and swim by assuming the tile height to be the trip count of the maximum range of the inner loop bounds. They generate the same tile sizes as STS, however, without padding applied. The vertical bars in Figures 31, 33, 35 and 37 indicate the execution time for both TLI and TSS. From Figures 30 to 37, the versions with padding run significantly faster than the ones without padding, with a speedup of 1.08 to 2.14. Therefore, STS is superior to TLI and TSS for tomcatv and swim.

We also applied the GroupPad algorithm [13] to the tiled tomcatv and swim. Table 10 compares the results with our inter-array padding. It is clear that padding for the L2 cache (using our interarray padding) is better than padding for the L1 cache (using GroupPad) in the case of 1-D skewed tiling with the L2 cache targeted.
Table 10: Execution time comparison between inter-array padding and GroupPad in seconds

<table>
<thead>
<tr>
<th></th>
<th>tomos(PUST II)</th>
<th>swim(PUST II)</th>
<th>tomos(R10K)</th>
<th>swim(R10K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-array padding</td>
<td>SPEC92 SPEC95</td>
<td>SPEC92 SPEC95</td>
<td>SPEC92 SPEC95</td>
<td>SPEC92 SPEC95</td>
</tr>
<tr>
<td></td>
<td>4 82</td>
<td>25 70</td>
<td>3.5 72</td>
<td>10.8 59.6</td>
</tr>
<tr>
<td>GroupPad</td>
<td>3.6 117</td>
<td>72 20.0</td>
<td>685</td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Summary of speedup of STS over other schemes

<table>
<thead>
<tr>
<th></th>
<th>ORG</th>
<th>LRW</th>
<th>TSS</th>
<th>TLI</th>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra II</td>
<td>2.25</td>
<td>1.63</td>
<td>1.95</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>R10K</td>
<td>2.38</td>
<td>1.30</td>
<td>1.42</td>
<td>1.27</td>
<td>1.22</td>
</tr>
<tr>
<td>Both</td>
<td>2.31</td>
<td>1.46</td>
<td>1.66</td>
<td>1.32</td>
<td>1.29</td>
</tr>
</tbody>
</table>

6.6 Discussion

In summary, Table 11 shows the normalized execution time for all 600 cases for SOR, Jacobi and LL18, where “Both” stands for both the Ultra II and the R10K. From Table 11, the major competitors of STS are DAT and TLI, where DAT performs a little better than TLI. However, because DAT ignores the L2 cache and only chooses square tile sizes, and TLI ignores the L2 cache and does not apply inter-array padding to minimize cross-interference misses, they do not perform as well as STS.

Comparison with LRW

One interesting point is related with LRW. Considering the combination of each benchmark (SOR, Jacobi and LL18) and each machine (Ultra II and R10K), LRW produces smaller average L1 cache misses in 3 out of 6 combinations compared with STS. However, this does not translate into large performance saving. We found that in general LRW produces smaller tile sizes than STS, which potentially introduces more loop overhead. For LL18, LRW has greater average L2 cache miss rates than STS since STS exploits locality for L2 cache in most of cases due to large number of arrays.

TLB

Unlike [17], we drop the TLB constraint in the STS algorithm such that the array footprint size within one tile may not fit in the TLB any more. As demonstrated in Section 6.4, such a TLB constraint may cause the L2 cache under-utilization, thus hurting the performance in many cases.

In our execution cost model, we did not consider the TLB misses. Although the TLB misses could be added to the execution cost, we consider the TLB misses are not important, when compared with the L1 and L2 cache misses, because of the following reasons:

- The current modern microprocessor often has a large TLB block size. For example, the Ultra II supports a block size of 8KB, 16KB, etc., and the R10K supports 32KB, 64KB, etc. The L2 cache block size, on the other hand, is much smaller.
- In general, the number of TLB entries are enough to execute one T-loop iteration without TLB thrashing.
Table 12: Recommendation of tiling schemes in different cases

<table>
<thead>
<tr>
<th></th>
<th>small $S_2$</th>
<th>large $S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>small $n_a$</td>
<td>2-D</td>
<td>1-D/2-D</td>
</tr>
<tr>
<td>large $n_a$</td>
<td>1-D</td>
<td>1-D</td>
</tr>
</tbody>
</table>

- For the relaxation codes we are targeting, the spatial locality is often fully exploited in the innermost loop body. That is, the array elements are often accessed with stride one in the innermost loop.

- Because of the above three factors, one TLB miss penalty can be amortized among a large number of TLB hits, which makes the TLB misses less important than L2 cache misses. For example, let us assume that each array element takes 8 bytes, the TLB block size is 32KB, the L2 cache block size is 128B and the array is accessed with spatial locality fully utilized. A TLB miss will be generated every 4K data references, while a L2 cache miss will be generated every 16 data references.

Miscellaneous

From Figures 4 to 6, the average L1 cache miss rates on the Ultra II are generally greater than those on the R10K. This is because the Ultra II has a smaller L1 cache size. Also from Figures 4 to 6, STS performs better on the Ultra II than on the R10K in most cases. Such results suggest that STS can have a bigger impact for smaller cache sizes.

Recommendation

Table 12 summarizes our general observation concerning 1-D vs. 2-D tiling. If the number of arrays in the T-loop body is large, 1-D tiling is recommended because it is difficult to satisfy Properties 1 and 2 (i.e., eliminating all interference misses). Otherwise, if the skewing factor $S_2$ is small, 2-D tiling is preferred. If $S_2$ is large, then either 1-D or 2-D may be applied, depending on the estimated execution cost. Both 1-D tiling and 2-D tiling can target either the L1 cache or the L2 cache, depending on the respective execution costs.

7 Conclusion

In this paper, we address the issues of when and how to exploit the L2 cache locality for skewed tiling. We present a tile-size selection algorithm, STS, based on an execution cost model which incorporates both the L1 and the L2 cache misses to determine when the L2 cache locality should be exploited. We apply inter-array padding to minimize the cross-interference misses so that the L2 cache locality can be achieved. The experimental results show that STS achieves an average speedup of 1.29 to 1.66 over previous algorithms for three test programs with various data sizes. For two SPEC benchmarks with different inputs, STS achieves a speedup of 1.08 to 2.14 over those applicable previous algorithms. The results produced by STS are shown to be within 5% of the optimal, except for one test case in which the STS underperforms by 13%. Overall, for skewed tiling, the experimental results favor STS over previously-proposed algorithms.

In our experiments, we found that turning on the compiler switch for prefetching for the tiled codes may degrade the performance. How to effectively combine tiling and prefetching seems an interesting future research topic.
References


