Tile Size Selection Using Cache Organization and Data Layout

Stephanie Coleman
scoleman@dsd.camb.inmet.com
Intermetrics, Inc., 733 Concord Ave.
Cambridge, MA 02138

Kathryn S. McKinley
mckinley@cs.umass.edu
Computer Science, LGRC, University of Massachusetts
Amherst, MA 01003

Abstract

When dense matrix computations are too large to fit in cache, previous research proposes tiling to reduce or eliminate capacity misses. This paper presents a new algorithm for choosing problem-size dependent tile sizes based on the cache size and cache line size for a direct-mapped cache. The algorithm eliminates both capacity and self-interference misses and reduces cross-interference misses. We measured simulated miss rates and execution times for our algorithm and two others on a variety of problem sizes and cache organizations. At higher set associativity, our algorithm does not always achieve the best performance. However on direct-mapped caches, our algorithm improves simulated miss rates and measured execution times when compared with previous work.

1 Introduction

Due to the wide gap between processor and memory speed in current architectures, achieving good performance requires high cache efficiency. Compiler optimizations to improve data locality for uniprocessors is increasingly becoming a critical part of achieving good performance [CMT94]. One of the most well-known compiler optimizations is *tiling* (also known as *blocking*). It combines strip-mining, loop permutation, and skewing to enable reused data to stay in the cache for each of its uses, *i.e.,* accesses to reused data are moved closer together in the iteration space to eliminate capacity misses.

Much previous work focuses on how to do the loop nest restructuring step in tiling [CK92, CL95, IT88, GJG88, WL91, Wol89]. This work however ignores the effects of real caches such as low associativity and cache line size on the cache performance of tiled nests. Because of these factors, performance for a given problem size can vary wildly with tile size [LRW91]. In addition, performance can vary wildly when the same tile sizes are used on very similar problem sizes [LRW91, NJL94]. These results occur because low associativity causes interference misses in addition to capacity misses.

Published in *SIGPLAN'95: Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

In this paper, we focus on how to choose the tile sizes given a tiled nest. As in previous research, our algorithm targets loop nests in which the reuse of a single array dominates. Given a problem size, the Tile Size Selection (TSS) algorithm selects a tile size that eliminates self-interference and capacity misses for the tiled array in a direct-mapped cache. It uses the data layout for a problem size, cache size, and cache line size to generate potential tile sizes. If the nest accesses other arrays or other parts of the same array, TSS selects a tile size that minimizes expected cross interferences between these accesses and for which the working set of the tile and other accesses fits in a fully associative LRU cache.

We present simulated miss rates and execution times for a variety of tiled nests that illustrate the effectiveness of the TSS algorithm. We compare these results to previous algorithms by Lam *et al.* [LRW91] and Esseghir [Ess93]. On average, TSS achieves better miss rates and performance on direct-mapped caches than previous algorithms because it selects rectangular tile sizes that use the majority of the cache. In some cases, it achieves significantly better performance. If the problem size is unknown at compile time, the additional overhead of computing problem-dependent tile sizes at runtime is negligible. We show that because TSS effectively uses the majority of the cache and its runtime overhead is negligible, copying is unnecessary and significantly degrades performance.

Section 2 compares our strategy to previous research. In Section 3, we briefly review the relevant terminology and features of caches, reuse, and tiling. Section 4 describes the tile size selection algorithm, TSS. It generates a selection of tile sizes without self-interference misses in a direct-mapped cache using the array size, the cache size, and the cache line size. It selects among these tile sizes to generate the largest tile size that fits in cache and that minimizes expected cross-interference misses from other accesses. Section 5 presents simulation and execution time results that demonstrate the efficacy of our approach and compares it to the work of Esseghir and Lam *et al.* [Ess93, LRW91].

2 Related Work

Several researchers describe methods for how to tile nests [BJWE92, CK92, CL95, IT88, GJG88, Wol89]. None of this work however addresses interference, cache replacement policies, cache line size, or spatial locality which are important factors that determine performance for current machines.

More recent work has addressed some of these factors for selecting tile sizes [Ess93, LRW91]. Esseghir selects tile sizes for a variety of tiled nests [Ess93]. His algorithm chooses the maximum number of complete columns that fit in the cache. This algorithm leaves one large gap of unused cache. All of his experiments were performed on the RS6000 (64K, 128 byte line, 4-way set associative). For this cache organization, Esseghir's strategy slightly out performs the TSS algorithm by a factor of 1.03. However, when compared to TSS or Lam *et al.* [LRW91] on an 8K cache with 1, 2, or 4-way set associative caches using matrices that are relatively large with respect to cache size (*e.g.,* 300×300), Esseghir's algorithm results in significantly higher miss rates. For example, TSS out performs it on the DEC Alpha (8K, 32 byte line, direct mapped) for matrix multiply by an average factor of 2 (Section 5).

Lam *et al.* present cache performance data for tiled matrix multiply and describe a model for evaluating cache interference [LRW91]. The model evaluates reuse for one variable, and quantifies self-interference misses for matrix multiply as a function of tile size. They show choosing a tile size that uses a fixed fraction of the cache performs poorly compared to tile sizes that are tailored for a problem and cache size. They present an algorithm which chooses the largest size for a square tile that avoids self interference based on array size. Square tiles use a smaller portion of the cache and result in higher miss rates and execution times when compared to the data-dependent rectangular tiles chosen by our algorithm (Section 5). TSS consistently improves execution times over Lam *et al.* by an average factor of 1.12 on the DEC Alpha and a smaller factor of 1.02 on the RS6000.

Esseghir, Lam *et al.*, and Temam *et al.* [TGJ93] all recommend copying as a method to avoid self-interference and cross-interference misses. Copying also requires knowledge of array sizes which may not be available until runtime. It makes performance much more predictable for varying tile sizes. However, computing the tile sizes at runtime with any of TSS, Esseghir, or Lam *et al.* has no noticeable impact on performance. TSS achieves significantly better performance than copying because it uses the majority of the cache, eliminates self interference, and minimizes cross interference (see Section 5).

3 Background

3.1 Cache Memory

Tiling can be applied to registers, the TLB, or any other level of the memory hierarchy. In this paper, we concentrate on tiling for the first level of cache memory. The cache is described by its size, line size, and set associativity [Smi82]. Unless otherwise indicated, we assume a direct-mapped cache. We divide cache misses into three categories.

Compulsory misses occur when a cache line is referenced for the first time. Without prefetching, these misses are unavoidable.

Capacity misses occur when a program's working set size is larger than the cache size and are defined with respect to a fully associative LRU cache. If a cache line containing data that will be reused is replaced before it is reused, a capacity miss occurs when the displaced data is next referenced. The miss is classified as a capacity miss only if it would occur in a fully LRU cache. Otherwise, it is classified as an interference miss.

Interference misses occur when a cache line that contains data that will be reused is replaced by another cache line. An interference miss is distinguished from a capacity miss because not all the data in the cache at the point of the miss on the displaced data will be reused. Intuitively, interference misses occur when there is enough room for all the data that will be reused, but because of the cache replacement policy data maps to the same location. Interference misses on arrays can be divided into two categories.

- **Self-interference** misses result when an element of the same array causes the interference miss.
- **Cross-interference** misses result when an element of a different array causes the interference miss.

3.2 Reuse

The two sources of data reuse are *temporal* reuse, multiple accesses to the same memory location, and *spatial* reuse, accesses to nearby memory locations that share a cache line. Without loss of generality, we will assume Fortran's column-major storage. Tiling only benefits loop nests with temporal reuse. We will also take advantage of spatial locality in tiled nests.

3.3 Tiling

Tiling reduces the volume of data accessed between reuses of an element, allowing a reusable element to remain in the cache until the next time it is accessed. Consider the code for matrix multiply in Figure 1(a) and its corresponding reuse patterns illustrated in Figure 2(a). The reference Y(J,K) is loop-invariant with respect to the I loop. Each iteration of the I loop also accesses one row each of X and Z. Therefore, $2*N + N^2$ elements are accessed per iteration of the I loop. Between each reuse of an element of Y there are N distinct elements of Z accessed on the J loop, N elements of the X array on the K loop, and N^2 - 1 elements

(a) Matrix Multiply

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

⇓ ⇓ ⇓

(b) Tiled Matrix Multiply

```
DO KK = 1, N, TK
    DO JJ = 1, N, TJ
        DO I = 1, N
            DO K = KK, MIN(KK+TK-1,N)
                R = X(K,I)
                DO J = JJ, MIN(JJ+TJ-1,N)
                    Z(J,I) = Z(J,I) + R * Y(J,K)
```

Figure 1:

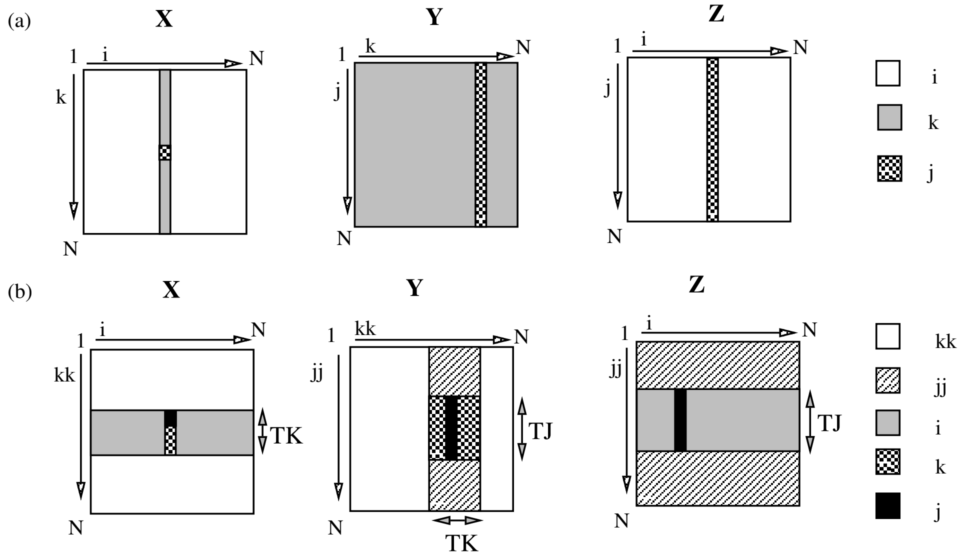

Figure 2: Iteration space traversal in (a) untiled and (b) tiled matrix multiply

of the Y array. If the cache is not large enough to hold this many elements, then the reusable Y data will be knocked out of the cache, and the program will have to repeat a costly memory access.

Previous research has focused on how to transform a nest into a tiled version to eliminate these capacity misses [CK92, CL95, IT88, GJG88, Wol89]. We assume as input a tiled nest produced by one of these methods and turn our attention the selection of tile sizes for the nest.

For example, tiled matrix multiply appears in Figure 1(b) and its corresponding reuse pattern in Figure 2(b). In the tiled nest, one iteration of the I loop now accesses only TK + TJ + TK*TJ elements. Between reuse of an element of Y, the J and K loops access TK distinct elements of X, TJ elements of Z, and TK * TJ elements of Y. We call the portion of an array which is referenced by a loop the *footprint* of that array reference [Wol89]. We call the number of times the same element is referenced by a loop the *reuse factor*. We call the innermost loop that has not been strip mined and interchanged the *target loop*, the I loop in matrix multiply; the target nest accesses a tile of data. Inspection of the array accesses, loop nesting, and loop bounds of the tiled nest determines the footprint and reuse factor [Wol89]. Table 1 illustrates these quantities for the version of tiled matrix multiply in Figure 1(b).

The largest tile with the most reuse on the I loop is the access to Y. We therefore target this reference to fit and stay in cache. We want to choose TK and TJ such that the TK×TJ submatrix of Y will still be in the cache after each iteration of the I loop and there is enough room in the cache for the working set size of the I loop, TK×TJ + TK + TJ.

	Reuse Factor			**Footprint**		
Array	I	K	J	I	K	J
X(K,I)	0	0	TJ	TK	1	1
Y(J,K)	N	0	0	TK*TJ	TJ	1
Z(J,I)	0	TK	0	TJ	TJ	1

Table 1: Reuse Factor and Footprint for Tiled Matrix Multiply

4 Tile Size Selection

Given a target array reference, we now show how to select a tile size for the reference.

4.1 Detecting and Eliminating Self Interference

In this section, we describe how to detect and eliminate self-interference misses when choosing a tile size. We compute a selection of tile sizes that exhibit no self interference and no capacity misses. Factors such as cross interference and working set size determine which size we select. We use the cache size, the line size, and the array column dimension. We only select tile sizes in which the column dimension is a multiple of the cache line size.

Consider the layout of a 200×200 array Y in a direct-mapped cache that can hold 1024 elements of Y as illustrated in Figure 3. Without loss of generality, we assume the first element of array Y falls in the first position of the cache. *Sets* are defined as groups of consecutive columns whose staring positions differ by N. The first set therefore consists of columns 1 through 6. Let *CS* be the cache size in elements, *CLS* the cache line size in elements, and *N* the column dimension (the consecutively stored dimension). The number of complete columns that fit in the

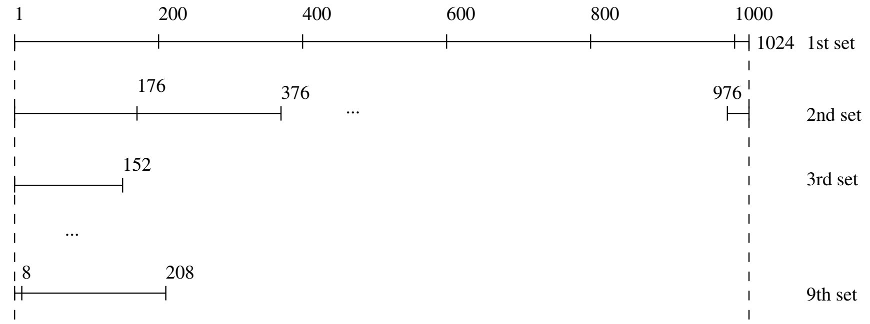

Figure 3: Column Layout for a 200×200 Array in a 1024 Element Cache

cache is simply

$$\textit{ColsPerSet} = \lfloor CS/N \rfloor. \tag{1}$$

For Figure 3, *ColsPerSet* = 5 for a tile of 200×5 (Esseghir selects this tile size [Ess93]). A 200×5 tile uses 97% percent of a 1024 element cache, but leaves a single contiguous gap. If N evenly divides CS, we also select this tile size. Otherwise, we look for a smaller column dimension with a larger row size that does not incur interference, combining to use a higher percentage of the cache.

We use the Euclidean algorithm [Kob87] to generate potential column dimensions. The Euclidean algorithm finds the *g.c.d.*(a, b) $a > b$, in $O(log^3(a))$ time. It computes

$$\begin{array}{lcl} a & = & q_1 b + r_1 \\ b & = & q_2 r_1 + r_2 \\ r_1 & = & q_3 r_2 + r_3 \\ & \cdots & \\ r_{k-1} & = & q_{k+1} r_k + r_{k+1} \end{array}$$

until a remainder divides a previous remainder. Remainders are always decreasing. For our purposes, $a = CS$ the cache size, and $b = N$ the column dimension. Each remainder is a potential column size. Given our example with $a = 1024$ and $b = 200$, Euclid generates the following.

$$\begin{array}{rcl} 1024 & = & 5 * 200 + 24 \\ 200 & = & 8 * 24 + 8 \end{array}$$

Since 8 divides 24, 8 = *g.c.d.*$(1024, 200)$ and it terminates.

We begin with an initial column size of $b = N$. We must reduce the column size to at least r_1 before additional columns will not incur interference. Look at the 6^{th} column's starting position in Figure 3. Even if we reduce the column size from 200 to 25, no additional columns will fit because when the 6^{th} column is of length 25 or greater it interferes with the first column. When the column size equals r_1, 24 in this example, it becomes possible to fit more columns. (When $r_1 > N - r_1$, only one more column fits with a column size of r_1 as opposed to N. Otherwise, the row size (the number of columns) increases by at least $\lfloor CS/N \rfloor$.)

The starting positions of the first and second set differ by *SetDiff* = $N - r_1$. The difference between subsequent sets will eventually become *Gap* = $N \bmod$ *SetDiff*. The row size is determined by the point at which the difference changes from r_1 to *Gap*. The algorithm for computing the row size for a column size which is a Euclidean remainder appears in Figure 4. The algorithm divides the cache into two sections: (1) the r_1 gap at the end of the first set and (2) the rest of the cache. It divides naturally because of the pattern in section 2. If an additional column starts a new set in section 2 and does not interfere with previous sets then at least an additional *ColsPerSet* will not interfere. A space of size Gap occurs at the end of the cache (in section 1, between the last column and the end of the cache) and eventually occurs between starting positions in section 2.

For each of section 1 and 2, we thus compute (a) the number of columns of *colSize* that fit between the starting addresses differing by *SetDiff*, and (b) the number of columns that fit in the spaces of size *Gap*. Since we only use Euclidean column sizes, *ColsPerGap* = ⌊ *Gap* / *colSize* ⌋.

For section 2 of the cache, *ColsPerN* is the number of *colSize* columns that fit between two complete columns of size N, dictated by *SetDiff*. *ColsPerSetDiff* is the number of columns of *colSize* that fit between columns that have starting positions with differences of *SetDiff*. Since columns are Euclidean numbers, it is the minimum distance allowed between starting positions of columns in different sets. *ColsPerGap* is the number that fit in the *Gap*. This pattern repeats *ColsPerSet* times. The number of columns that fit in section 2 of the cache is thus

(*ColsPerSetDiff* $*$ *ColsPerN* + *ColsPerGap*) $*$ *ColsPerSet*.

When *SetDiff* $< r_1$ patterns of columns fit in section 1, the pattern columns total *ColsPerSetDiff* $*$ ⌊ r_1 / *SetDiff* ⌋. The total is thus any pattern columns plus *ColsPerGap*.

Returning to the example in Figure 3, *ColsPerSet* = 5, $r_1 = 24$, *SetDiff* = 176, and *ColsPerN* = 1. Given a column size of 24, *ColsPerSetDiff* = 7 and *ColsPerGap* = 1. Thus *rowSize* = 7*1*5 + 1*5 + 0 + 1 = 41, for a tile size of 24×41.

4.2 Cache Line Size

To take advantage spatial locality, we choose column sizes that are multiples of the cache line size in terms of elements, *CLS*. We assume the start of an array is aligned on a cache line boundary.

```
Input:        CS: Cache Size, N: Column Dimension,
              colSize = r_k: Euclidean remainder
Output:       rowSize: max rows without interference
Invariants:   ColsPerSet = q_1 = ⌊CS/N⌋
              r_1 = CS mod N
              SetDiff = N − r_1
              ColsPerN = ⌊N/SetDiff⌋
              Gap = N mod SetDiff
procedure ComputeRows (colSize)
    if (colSize = N) return ColsPerSet
    else if (colSize = r_1 & colSize > SetDiff)
        return ColsPerSet + 1
    else
        ColsPerSetDiff = ⌊ SetDiff / colSize ⌋
        ColsPerGap = ⌊ Gap / colSize ⌋
        rowSize = ColsPerSetDiff * ColsPerN * ColsPerSet
            + ColsPerGap * ColsPerSet
            + ColsPerSetDiff * ⌊ r_1 / SetDiff ⌋ + ColsPerGap
        return rowSize
    endif
```

Figure 4: Row Sizes for Euclidean Column Sizes

After we find the row size, we simply adjust *colSize* as follows.

$$colSize = \begin{cases} colSize & \text{if } colSize \bmod CLS = 0, \text{or} \\ & \text{if } colSize = \text{column length} \\ \left\lfloor \frac{colSize}{CLS} \right\rfloor CLS & \text{otherwise} \end{cases}$$

If *colSize* is equal to the length of the column, we do not adjust it to a multiple of the line size.

4.3 Minimizing Cross Interference

In this section, we compute worst case cross-interference misses for tiled nests. We use footprints to determine the amount of data accessed in the tile and choose a tile size that has a working set size that fits in the cache and minimizes the number of expected cross-interference misses.

Consider again matrix multiply from Figure 1. On an iteration of the I loop (from Table 1), we access TK×TJ elements of Y, TJ elements of Z, and TK elements of X. If we completely fill up the cache with array Y, then every reference to Z causes a cross-interference miss on the next reference to the victimized element of Y. Another cross-interference miss will occur on the next reference to the same element of Z. There are potentially 2 * TJ cross interferences between Z and Y. Despite the fact that the element of X that is reused in the J loop is register-allocated, it still uses a cache line on its first access. This access causes a cross-interference miss on the next reference to any victimized element of Y. We ignore the cross-interference misses of Z caused by X because they happen very infrequently and are hard to predict. In the worst case, the number of cross-interference misses, *CIM*, that will occur as a result of a TK×TJ tile size in matrix multiply is

$$CIM = 2 * TJ + TK. \qquad (2)$$

The cross-interference rate, CIR, is thus

$$CIR = CIM/(TJ * TK), \qquad (3)$$

the number of cross-interference misses per element of Y in a single iteration of the I loop. To compute this rate in general, we use the footprints for the target nest for array accesses other than the target. Each footprint is then multiplied by 2 if they do not reduce to one on the next inner loop in the nest.

We also minimize expected cross interference by selecting tile sizes such that the working set will fit in the cache, *e.g.,* for matrix multiply the *working set size constraint* is

$$TJ * TK + TJ + 1 * CLS < CS. \qquad (4)$$

(Since X is register allocated, we reduce its footprint to *CLS*.) The left-hand side of equation 4 is exactly the amount of cache need for an fully-associative LRU cache. In general, the working set is simply the sum of the footprints for the target loop which accesses the tile.

We use the cross-interference rate and working set size constraint to differentiate between tile sizes generated by the algorithm described in Section 4.1. As the algorithm iterates, we select a new tile size without self interference if its working set size is larger than a previous tile size and still fits in cache and the new size has a lower *CIR* than the previous size. (Section 5 demonstrates that these tile sizes result in lower miss rates for direct-mapped caches.)

If the tile we select does not meet these constraints, we decrease *colSize* by *CLS* until it does. Both this phase and the self interference phase result in numerous gaps through out the cache, rather than one large gap. Because cross-interfering arrays typically map all over the cache, multiple gaps minimize the expected interference, *e.g.,* the columns of Z are more likely to map to one of many gaps rather than one larger gap.

4.4 Tile Size Selection Algorithm

The pseudo code for the TSS algorithm which completely avoids self interference, capacity misses, and minimizes expected cross interference appears in Figure 5. We begin by selecting the column dimension which is the maximum *colSize* and determine the maximum *rowSize* without self interference. In the pathological case, the column length evenly divides the cache and the algorithm terminates (this case often occurs with power of 2 array sizes). If the tile sizes are larger than the array or the bounds of the iteration space, there is no need to tile. In addition, if any of the dimensions of the tiles are larger than the bounds of the iteration space, the tiles are adjusted accordingly. For simplicity, these checks are omitted from Figure 5.

The *while* loop iterates finding potential tile sizes without self interference using Euclidean numbers as candidates for column dimensions. After determining the row size that will not interfere for the given column size, the column size is adjusted to a multiple of the CLS. If any newly compute tile size has a larger working set size that is also less than CS and for which the CIR is less than the previous best tile size, we set this tile

procedure TSS(CS, CLS, N, M)

Input: CS: cache size, CLS: cache line size,
N: column length, M: row length

Output: tile size = *bestCol*×*bestRow*

```
bestCol = oldCol = N
bestRow = rowSize = CS/N
colSize = bestCol mod N
while (colsSize > CLS & oldCol mod colSize ≠ 0 & rowSize < M)
    rowSize = computeRows (colSize)
    tmp = colSize adjusted to a multiple of CLS
    if ( WSet (tmp, rowSize) > WSet (bestCol, bestRow)
        & WSet (tmp, rowSize) < CS
        & CIR (tmp, rowSize) < CIR (bestCol, bestRow)
            bestCol = tmp
            bestRow = rowSize
    endif
    tmp = colSize
    colSize = oldCol mod colSize
    oldCol = tmp
endwhile
if necessary, adjust bestCol to meet the working set size constraint
end TSS
```

Figure 5: Tile Size Selection Algorithm

to be the best. If the initial tile size does not meet the working set size constraint and no subsequent tile size does either, we use the initial tile size, but reduce its column size by CLS until it meets the constraint.

4.5 Set Associativity

Set associativity does not affect the tile size picked by the algorithm for a particular cache size. As expected, increasing the set associativity usually decreases the miss rate on a particular tile size because more cross interferences, if they exist, are eliminated. Our results in Section 5 confirm this expected benefit from set associativity and illustrate that increasing set associativity causes the differences in miss rates between distinct tile sizes to become less extreme.

4.6 Translation Lookaside Buffer

A translation lookaside buffer (TLB) is a fast memory used for storing virtual to physical address mappings for the most recently reference page entries. All addresses referenced in the CPU must be translated from virtual to physical before the search for an element is performed. If the mapping for the element is not in the TLB, a TLB miss occurs, causing the rest of the system to stall until the mapping completes. A TLB miss can take anywhere from 30 to more than 100 cycles, depending on the machine and the type of TLB miss.

To avoid a TLB misses, tile sizes should enable the TLB to hold all the entries required for the tile. In general, the height (column in Fortran) of the tile should be much larger than the width of the tile. Since a TLB miss can cause a cache stall, ensuring that no TLB misses occur is more important than the cross and self-interference constraints. The tile size therefore needs to be constrained such that the number of non-consecutive elements accesses (*i.e.,* rows) is smaller than the number of page table entries in the TLB.

5 Results

For our experiments, we used the following tiled kernels on double precision (16 byte) two dimensional arrays.

mm	Matrix Multiply, tiled in 2 dimensions
sor	Successive Over Relaxation, tiled in 1 & 2 dimensions
lud	LU Decomposition, tiled in 1 & 2 dimensions
liv23	Livermore loop 23, tiled in 1 dimension

The point and tiled versions of SOR, LUD, and Livermore loop 23 appear in Appendix A. We used square arrays except for Livermore loop 23. The original Livermore loop 23 is 101×7. We increase both its bounds by a factor of 3 to preserve its shape and make it worthwhile to tile. Matrix multiply and Livermore loop 23 incur self and cross interference. SOR and LUD operate on a single array and thus by definition incur only self interference. Some of self interference for LUD results from non-contiguous accesses and we minimize it the same way we do cross interference for matrix multiply.

We selected problem sizes of 256×256 to illustrate the pathological case, and 300×300 and 301×301 to illustrate the effect a small change in the problem size has on selected tile sizes and performance. We used these relatively small sizes in order to obtain timely simulation results. For execution results, we added a problem size of 550×550. We expect, and others have demonstrated, more dramatic improvements for larger array sizes.

5.1 Simulation Results for Tiled Kernel

We ran simulations on these programs using the Shade cache simulator. We used a variety of cache parameters: a cache size of 8K and 64K; set-associativity of 1, 2, and 4; and a cache line size of 32, 64, and 128 [Col94]. Of these, we present cache parameters corresponding to the DEC Alpha Model 3000/400 (8K, 32 byte line, direct-mapped) and the RS/6000 Model 540 (64K, 128 byte line, 4-way) with variations in line size and associativity. We also executed the kernels on these machines.

In Table 2, we show simulated miss rates in an 8K cache for double precision arrays (16 byte) for the untiled algorithm and the version tiled with TSS. We present results for set associativities of 1, 2, and 4 and cache line sizes of 32 bytes and 128 bytes. TSS achieves significant improvement in the miss rates for most of these kernels. On average, it improves miss rates by a factor of 14. It improves SOR2D by a factor of 66.21 on a 8K, 4-way, 32 byte line cache. The improvement for 32 byte lines is higher, a factor of 21.8, than that for 128 byte lines, 6.8, because the longer line sizes benefit these kernels, all of which have good spatial locality. If the dramatic improvements of SOR2D are ignored, our algorithm improves miss rates by an average of 2.5 (3.3 on 32 byte lines and 1.7 on 128 byte lines).

Two more trends for tiled kernels are evident in this table. The first trend is to be expected: even though TSS selects tile sizes for direct-mapped caches, these tile sizes always improve their

Kernel	Sets	Line (bytes)	Untiled Size	Untiled Miss Rate	TSS Tile Size	TSS Miss Rate	TSS Improvement in Miss Rate
mm	1	32	300×300	2.208	16×29	0.613	3.60
mm	2	32	300×300	2.147	16×29	0.257	8.35
mm	4	32	300×300	2.164	16×29	0.197	10.98
mm	1	128	300×300	1.176	16×29	1.139	1.03
mm	2	128	300×300	0.458	16×29	0.274	1.67
mm	4	128	300×300	0.426	16×29	0.322	1.32
sor1D	1	32	300×300	1.214	300×86	1.204	1.01
sor1D	2	32	300×300	0.901	300×86	0.882	1.02
sor1D	4	32	300×300	0.927	300×86	0.880	1.05
sor1D	1	128	300×300	1.980	300×86	1.506	1.32
sor1D	2	128	300×300	0.992	300×86	0.262	3.79
sor1D	4	128	300×300	0.255	300×86	0.255	1.00
sor2D	1	32	300×300	1.214	86×3	0.076	15.97
sor2D	2	32	300×300	0.901	86×3	0.004	225.25
sor2D	4	32	300×300	0.927	86×3	0.004	231.75
sor2D	1	128	300×300	1.980	80×3	0.248	1.56
sor2D	2	128	300×300	0.992	80×3	0.176	5.63
sor2D	4	128	300×300	0.248	80×3	0.003	82.67
lud1D	1	32	300×300	1.482	300×2	0.746	1.99
lud1D	2	32	300×300	1.002	300×2	0.439	2.28
lud1D	4	32	300×300	0.990	300×2	0.398	2.49
lud1D	1	128	300×300	1.947	300×2	1.010	1.93
lud1D	2	128	300×300	0.366	300×2	0.166	2.21
lud1D	4	128	300×300	0.302	300×2	0.117	2.58
lud2D	1	32	300×300	1.537	16×29	0.471	3.26
lud2D	2	32	300×300	1.505	16×29	0.302	4.98
lud2D	4	32	300×300	1.498	16×29	0.271	5.52
lud2D	1	128	300×300	0.604	16×29	0.463	1.30
lud2D	2	128	300×300	0.405	16×29	0.203	2.00
lud2D	4	128	300×300	0.401	16×29	0.183	2.19
liv23	1	32	303×21	6.061	64×21	5.850	1.04
liv23	2	32	303×21	5.713	64×21	5.258	1.09
liv23	4	32	303×21	4.761	64×21	4.708	1.01
liv23	1	128	303×21	4.161	64×21	4.087	1.02
liv23	2	128	303×21	3.316	64×21	3.117	1.06
liv23	4	128	303×21	2.764	64×21	2.576	1.07
Average Improvement							14.0
Average Improvement for 32 byte lines							21.8
Average Improvement for 128 byte lines							6.8

Table 2: Miss rates in 8K cache for double precision element (16 byte) arrays

performance when associativity is higher. The second trend is that for the untiled kernels and direct-mapped cache, the 128 byte lines have higher miss rates than 32 byte lines for all but MM and LIV23. The interference from the larger line size plays an important role. For 4-way caches, all of the untiled kernels have lower miss rates for 128 byte lines because the increased set associativity has overcome the interference.

5.2 Comparing Algorithms

In both simulations and execution results, we compare our tile sizes to those chosen by Lam *et al.* [LRW91] and Esseghir's algorithms [Ess93]. We use the algorithms presented in their papers to compute tile sizes for the different cache organizations and data sets. LRW generates the largest square tiles without self interference. Esseghir chooses the column length, N for the column tile size and $\lfloor CS/N_c \rfloor$ for the row size. For 1 dimensional tiling, simply choosing the correct number of complete columns of size N suffices and as a result comparisons are uninteresting. We therefore consider matrix multiply, SOR2D, and LUD2D since they are tiled in 2 dimensions and the algorithms usually produce different tile sizes. We compare the results for 8K and 64K caches separately since they have slightly different behaviors.

5.2.1 8K Caches

Table 3 presents simulated miss rates for an 8K, 32 byte line cache with associativities of 1, 2, and 4 for TSS, LRW, Esseghir, and the untiled kernels. For each kernel, we present the selected tile size (the actual parameters to the tiled algorithm), the working set size (*WSet*), and the simulated miss rates. The working set size is presented to demonstrate the cache efficiency of the selected tile sizes. Notice the reduced tile sizes for SOR2D. The reduction accounts for the working set size of SOR2D which is two greater than the tile size.

LRW has lower miss rates than TSS only on LUD2D. TSS has lower miss rates on MM and lower or similar rates on SOR2D. On average, TSS improves miss rates by a factor of 1.83 over LRW, excluding arrays of size 256×256. (We exclude this case since padding is probably a better solution to pathological interference). TSS has consistently lower simulated miss rates than Esseghir, on average a factor of 6.66 (excluding arrays of size 256×256, including them 5.34). For example, on SOR2D 301×301 4-way, TSS improves miss rates by a factor of 50 over Esseghir. These results hold for larger line sizes as well [Col94].

TSS's lower simulated miss rates translate into better performance. Table 4 presents execution time results on the DEC Alpha (first level cache: 8K, direct-mapped, 32 byte line; second level cache: 512K, banked). We measured the execution times for TSS with and without computing the tile sizes at runtime. It made no measurable impact on performance.

The simulated miss rates and execution times for SOR2D do not always agree (TSS and LRW should be closer), nor do the miss rates and execution times for LUD2D when TSS is compared to Esseghir (TSS should be significantly better). We believe these inconsistencies result due to a combination of two factors: the difference in working set sizes and the Alpha's large second level cache (512K). TSS always has at least as good cache efficiency in terms of working set size as the other algorithms. Esseghir often uses too big of a working set, resulting in interference. LRW uses a small working set (often around 50%) because they are limited to square tiles. TSS may therefore get more benefit from the second level cache.

TSS always improves or matches execution time when compared to LRW or Esseghir's algorithm. On average, TSS improved over LRW by a factor of 1.05 when the pathological cases with array sizes of 256×256 are excluded. TSS improved over Esseghir on average by a more significant factor of 1.37, and by 2.01 on matrix multiply (again excluding array sizes of 256×256).

5.2.2 64K Caches

Tables 5 and 6 show the same type of results as the previous two tables, but for variants of the RS/6000 organization (64K, 4-way, 128 byte line). The simulated miss rates and execution times showed more variations than those for the 8K cache. These inconsistencies probably result because the simulator uses an LRU replacement policy and the RS6000 uses a quicker, but unpublished replacement policy.

TSS still achieves lower miss rates more often (a factor of 1.19 without 256×256 arrays), but Esseghir's algorithm has lower miss rates for most of the 4-way simulated results. Esseghir's algorithm out performs TSS and LRW in execution times on the RS6000 (for TSS, by a factor of 1.03). This result probably stems from set associativity and efficiency of the working set size. For this cache organization, Esseghir tends to encounter less interference because the working set sizes either fit in cache or are only slightly larger than the cache. When Esseghir encounters cross interference, the 4-way associativity is now more likely to over come it through the cache, rather than the tile sizes. For miss rates, LRW achieves lower miss rates TSS by a factor of 1.17 (excluding 256×256 arrays), but most of this comes from SOR2D. When SOR2D is excluded, TSS has lower miss rates by a factor of 1.10 (excluding 256×256 arrays). TSS however continues to out performs LRW on the RS/6000, as it did on the Alpha, but by less. In both Table 5 and 3, the square tile sizes use significantly less of the cache and are probably therefore not as effective.

5.3 Copying

To demonstrate that copying is unnecessary to achieve good per-

N	TSS	Tile-and-Copy	Speedup
256	1.68	2.92	1.74
300	1.88	3.28	1.75
301	1.82	3.52	1.93
550	11.85	20.15	1.76

Table 7: Matrix multiply execution times (seconds) on the Alpha

		Untiled		TSS			LRW			Esseghir		
Kernel	Sets	Size N×N	Miss Rate	Tile Size	WSet Size	Miss Rate	Tile Size	WSet Size	Miss Rate	Tile Size	WSet Size	Miss Rate
mm	1	256	2.325	170×2	512	1.905	2×2	8	3.778	256×2	770	4.116
mm	2	256	1.699	170×2	512	0.964	2×2	8	1.905	256×2	770	0.969
mm	4	256	1.679	170×2	512	0.878	2×2	8	1.876	256×2	770	0.894
mm	1	300	2.208	16×29	482	0.613	16×16	274	0.650	300×1	602	2.275
mm	2	300	2.147	16×29	482	0.257	16×16	274	0.277	300×1	602	1.617
mm	4	300	2.164	16×29	482	0.197	16×16	274	0.245	300×1	602	1.587
mm	1	301	2.308	28×17	506	0.600	17×17	308	0.596	301×1	604	2.275
mm	2	301	1.706	28×17	506	0.208	17×17	308	0.230	301×1	604	1.618
mm	4	301	1.680	28×17	506	0.177	17×17	308	0.213	301×1	604	1.587
sor2D	1	256	1.966	256×1	768	0.527	2×2	10	0.404	256×1	768	0.527
sor2D	2	256	1.943	256×1	768	0.477	2×2	10	0.253	256×1	768	0.477
sor2D	4	256	1.816	256×1	768	0.098	2×2	10	0.123	256×1	768	0.098
sor2D	1	300	1.214	86×3	440	0.076	14×14	256	0.076	300×1	900	0.409
sor2D	2	300	0.901	86×3	440	0.004	14×14	256	0.004	300×1	900	0.138
sor2D	4	300	0.927	86×3	440	0.004	14×14	256	0.004	300×1	900	0.166
sor2D	1	301	1.125	88×3	450	0.065	15×15	289	0.073	301×1	903	0.429
sor2D	2	301	0.901	88×3	450	0.003	15×15	289	0.003	301×1	903	0.165
sor2D	4	301	0.927	88×3	450	0.004	15×15	289	0.003	301×1	903	0.200
lud2D	1	256	3.605	170×2	512	2.312	2×2	8	0.971	256×2	770	2.381
lud2D	2	256	1.268	170×2	512	0.821	2×2	8	0.783	256×2	770	0.857
lud2D	4	256	1.266	170×2	512	1.107	2×2	8	0.816	256×2	770	1.172
lud2D	1	300	1.537	16×29	509	0.471	16×16	288	0.421	300×1	602	1.234
lud2D	2	300	1.499	16×29	509	0.302	16×16	288	0.274	300×1	602	1.183
lud2D	4	300	1.498	16×29	509	0.271	16×16	288	0.254	300×1	602	1.178
lud2D	1	301	1.537	30×12	402	0.398	17×17	323	0.389	301×1	604	1.233
lud2D	2	301	1.511	30×12	402	0.271	17×17	323	0.246	301×1	604	1.190
lud2D	4	301	1.504	30×12	402	0.249	17×17	323	0.226	301×1	604	1.188

Table 3: Miss rates and tile sizes in 8K (512 element) cache with 32 byte (2 element) lines for N×N arrays.

	Untiled		TSS		LRW		Esseghir		Speedup	
Kernel	N×N	Time	Tile Size	Time	Tile Size	Time	Tile Size	Time	LRW/TSS	ESS/TSS
mm	256	1.85	170×2	1.68	2×2	5.69	256×2	1.70	3.39	1.01
mm	300	3.36	16×29	1.88	16×16	1.88	300×1	3.47	1.00	1.85
mm	301	3.41	28×17	1.82	17×17	2.09	301×1	3.64	1.15	2.00
mm	550	26.03	18×27	11.85	18×18	12.95	512×1	25.67	1.09	2.17
sor2D	256	1.53	256×1	1.50	2×2	2.60	256×1	1.50	1.73	1.00
sor2D	300	2.35	88×3	2.07	16×14	2.18	300×1	2.10	1.05	1.01
sor2D	301	2.42	90×3	2.10	17×15	2.27	301×1	2.11	1.08	1.00
sor2D	550	9.81	38×12	7.06	18×16	7.42	512×1	8.87	1.05	1.26
lud2D	256	1.23	170×2	1.13	2×2	3.64	256×2	1.11	3.22	0.98
lud2D	300	1.88	16×29	1.76	16×16	1.76	300×1	1.60	1.00	0.91
lud2D	301	1.91	30×12	1.64	17×17	1.76	301×1	1.61	1.07	0.98
lud2D	550	13.48	18×27	10.32	18×18	10.36	512×1	11.92	1.00	1.15
Average Improvement									1.39	1.27
Average Improvement without 256×256 cases									1.05	1.37

Table 4: Execution Times in seconds on the DEC Alpha (direct-mapped, 8K, 32 byte lines)

		Untiled		TSS			LRW			Esseghir		
Kernel	Sets	Size N×N	Miss Rate	Tile Size	WSet Size	Miss Rate	Tile Size	WSet Size	Miss Rate	Tile Size	WSet Size	Miss Rate
mm	1	256	4.547	240×16	4088	0.566	16×16	280	0.647	256×16	4360	0.569
mm	2	256	4.441	240×16	4088	0.040	16×16	280	0.080	256×16	4360	0.045
mm	4	256	4.439	240×16	4088	0.036	16×16	280	0.029	256×16	4360	0.030
mm	1	300	0.534	88×41	3704	0.133	41×41	1730	0.193	300×13	4208	0.218
mm	2	300	0.419	88×41	3704	0.042	41×41	1730	0.043	300×13	4208	0.042
mm	4	300	0.419	88×41	3704	0.042	41×41	1730	0.030	300×13	4208	0.036
mm	1	301	0.509	112×26	3086	0.081	53×53	2870	0.172	301×13	4222	0.198
mm	2	301	0.524	112×26	3086	0.045	53×53	2870	0.026	301×13	4222	0.038
mm	4	301	0.419	112×26	3086	0.045	53×53	2870	0.023	301×13	4222	0.036
sor2D	1	256	0.080	256×14	4096	0.157	16×14	256	0.041	256×14	4096	0.157
sor2D	2	256	0.010	256×14	4096	0.005	16×14	256	0.001	256×16	4096	0.005
sor2D	4	256	0.001	256×14	4096	0.003	16×14	256	0.006	256×16	4096	0.003
sor2D	1	300	0.064	300×11	3900	0.161	41×39	1681	0.031	300×11	3900	0.161
sor2D	2	300	0.008	300×11	3900	0.003	41×39	1681	0.001	300×11	3900	0.003
sor2D	4	300	0.001	300×11	3900	0.003	41×39	1681	0.001	300×11	3900	0.003
sor2D	1	301	0.064	301×11	3913	0.161	53×51	2809	0.033	301×11	3913	0.161
sor2D	2	301	0.008	301×11	3913	0.003	53×51	2809	0.001	301×11	3913	0.003
sor2D	4	301	0.001	301×11	3913	0.003	53×51	2809	0.001	301×11	3913	0.003
lud2D	1	256	0.601	240×16	4096	0.245	16×16	288	0.251	256×16	4368	0.244
lud2D	2	256	0.307	240×16	4096	0.034	16×16	288	0.053	256×16	4368	0.030
lud2D	4	256	0.305	240×16	4096	0.038	16×16	288	0.053	256×16	4368	0.034
lud2D	1	300	0.330	88×41	3737	0.075	41×41	1763	0.070	300×13	4213	0.065
lud2D	2	300	0.319	88×41	3737	0.027	41×41	1763	0.030	300×13	4213	0.038
lud2D	4	300	0.318	88×41	3737	0.026	41×41	1763	0.028	300×13	4213	0.038
lud2D	1	301	0.333	112×26	3050	0.050	53×53	2915	0.080	301×13	4227	0.064
lud2D	2	301	0.321	112×26	3050	0.028	53×53	2915	0.029	301×13	4227	0.037
lud2D	4	301	0.522	112×26	3050	0.027	53×53	2915	0.028	301×13	4227	0.038

Table 5: Miss rates and tile sizes in a 64K (4096 element) cache, 128 byte (8 element) lines

	Untiled		TSS		LRW		Esseghir		Speedup	
Kernel	N×N	Time	Tile Size	Time	Tile Size	Time	Tile Size	Time	LRW/TSS	ESS/TSS
mm	256	2.08	240×16	1.79	16×16	2.15	256×16	1.76	1.20	0.98
mm	300	3.37	88×41	2.89	41×41	2.99	300×13	2.82	1.03	0.98
mm	301	3.42	112×26	2.87	53×53	2.97	301×13	2.89	1.03	1.01
mm	550	20.80	56×64	18.03	58×58	18.07	550×7	17.76	1.00	0.99
sor2D	256	2.27	256×14	2.21	16×14	2.30	256×16	2.21	1.01	1.00
sor2D	300	3.12	300×11	3.05	41×39	3.06	300×11	3.05	1.00	1.00
sor2D	301	3.14	301×11	3.05	53×51	3.06	301×11	3.05	1.00	1.00
sor2D	550	10.62	550×5	10.27	58×56	10.37	550×5	10.27	1.01	1.00
lud2D	256	1.27	240×16	0.82	16×16	1.04	256×16	0.79	1.27	0.96
lud2D	300	2.02	88×41	1.38	41×41	1.47	300×13	1.29	1.07	0.93
lud2D	301	2.06	112×26	1.35	53×53	1.45	301×13	1.28	1.07	0.95
lud2D	550	12.87	56×64	8.70	58×58	8.65	550×7	7.94	0.99	0.91
Average Improvement									1.06	0.98
Average Improvement without 256×256 cases									1.02	0.97

Table 6: Execution Times in seconds on the RS/6000 (64K, 4-way, 128 byte lines)

formance, we compared execution times for tiled matrix multiply using tile sizes chosen by TSS to execution times for code generated by Esseghir's Tile-and-Copy algorithm [Ess93]. These results appear in Table 7. The execution times for TSS include computing the tile sizes at runtime. Tile sizes for Esseghir's code were calculated using the formula $TS = \sqrt{CS/CLS} - 2$ where TS is the tile size. We also used Esseghir's algorithm to select rectangular tile sizes, but these execution times were longer. TSS significantly speeds up performance as compared to Esseghir's Tile-and-Copy algorithm. Because copying is an expensive run-time operation, tiling alone performs significantly better, even on the pathological $N = 256$ case where there is severe self interference.

6 Summary

This paper presents a new algorithm for choosing problem-size dependent tile sizes using the cache size and line size. Its runtime cost is negligible making it practical for selecting tile sizes and deciding when to tile whether or not array sizes are unknown at compile time. This algorithm performs better than previous algorithms on direct-mapped caches. It also performs well for caches of higher associativity when matrices are larger in comparison to the cache size. It obviates the need for copying. As compilers for scalar and parallel compilers increasingly turn their attention to data locality, tiling and other data locality optimizations will only increase in importance. We have shown the tile size selection algorithm to be a dependable and effective component for use in a compiler optimization strategy that seeks to use and manage the cache effectively.

Acknowledgements

We would like to thank Sharad Singhai for assisting in the initial experiments on this work and James Conant and Susan Landau for their insights on the Euclidean Algorithm.

References

[BJWE92] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis. A quantitative algorithm for data locality optimzation. In *Code Generation-Concepts, Tools, Techniques*. Springer-Verlag, 1992.

[CK92] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[CL95] S. Carr and R. B. Lehoucq. A compiler-blockable algorithm for QR decomposition. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[CMT94] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[Col94] S. Coleman. Selecting tile sizes based on cache and data organization. Master's thesis, Dept. of Computer Science, University of Massachusetts, Amherst, September 1994.

[Ess93] K. Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

[Kob87] Neal Koblitz. *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1987.

[LRW91] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[NJL94] J. J. Navarro, T. Juan, and T. Lang. Mob forms: A class of multilevel block algorithms for dense linear algebra operations. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 354–363, Manchester, England, June 1994.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[TGJ93] O. Temam, E. Granston, and W. Jalby. 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 Supercomputing '93*, Portland, OR, November 1993.

[WL91] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[Wol89] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, NV, November 1989.

A Tiled Kernels

A.1 SOR

Successive over relaxation is a five-spot stencil on a two dimensional array. The point algorithm with the best locality appears in Figure 6(a). The tiled versions for one and two dimensions appear in Figure 6(b) and (c), respectively.

(a) SOR

```
DO K = 1, N
    DO J = K+1, N
        DO I = 2, N-1
            A(I,J) = 0.2(A(I,J) + A(I+1,J) + A(I-1,J)
                    + A(I,J+1) + A(I,J-1)
```

⇓ ⇓ ⇓

(b) sor1d: SOR Tiled in One Dimension

```
DO K = 1, N
    DO II = 2, N-1, TI
        DO J = 2, N-1
            DO I = II, MIN(N-1,II+TI-1)
                A(I,J) = 0.2(A(I,J) + A(I+1,J) + A(I-1,J)
                        + A(I,J+1) + A(I,J-1)
```

⇓ ⇓ ⇓

(c) sor2d: SOR Tiled in Two Dimensions

```
DO JJ = 2, N-1, TJ
    DO K = 1, N
        DO II = 2, N-1, TI
            DO J = JJ, MIN(N-1, JJ+TJ-1)
                DO I = II, MIN(N-1, II+TI-1)
                    A(I,J) = 0.2(A(I,J) + A(I+1,J) + A(I-1,J)
                            + A(I,J+1) + A(I,J-1)
```

Figure 6:

A.2 Livermore Loop 23

Livermore Loop 23 is also a stencil kernel, but over 4 distinct arrays. The point and one dimensional tiled versions appear in Figure 7 respectively.

(a) Livermore Loop 23

```
DO J = 2, M
    DO K = 2, N
        QA= ZA(K,J+1)*ZR(K,J) +ZA(K,J-1)*ZB(K,J) +
            ZA(K+1,J)*ZU(K,J) +ZA(K-1,J)*ZV(K,J) +ZZ(K,J)
        ZA(K,J)= ZA(K,J) +fw*(QA -ZA(K,J))
```

⇓ ⇓ ⇓

(b) liv23: Livermore Loop 23 Tiled in One Dimension

```
DO KK = 2,N,TK
    DO J = 2,M
        DO K = KK,MIN(KK+TK-1,N)
            QA= ZA(K,J+1)*ZR(K,J) +ZA(K,J-1)*ZB(K,J) +
              ZA(K+1,J)*ZU(K,J) +ZA(K-1,J)*ZV(K,J) +ZZ(K,J)
            ZA(K,J)= ZA(K,J) +fw*(QA -ZA(K,J))
```

Figure 7:

A.3 LUD

LUD decomposes a matrix A into two matrices, L and U, where U is upper triangular and L is a lower triangular unit matrix. The point algorithm for LUD appears in Figure 8(a). The version of LUD tiled in one dimension in Figure 8(b) is taken from Carr and Kennedy [CK92]. The version tiled in two dimensions in Figure 8(c) is a modified version of the one Wolf and Lam use [WL91]. Their original version targeted C's row-major storage and we modified it to target Fortran's column-major storage.

(a) LUD

```
DO K = 1, N
    DO J = K+1, N
        A(J,K) = A(J,K)/A(K,K)
        DO I = K+1, N
            A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

⇓ ⇓ ⇓

(b) lud1d: LUD Tiled in One Dimension

```
DO KK = 1, N, TK
    DO K = KK, MIN(N,KK+TK-1)
        DO I = K+1, N
            A(I,K) = A(I,K)/A(K,K)
        DO J = K+1, N
            DO I = K+1, N
                A(I,J) = A(I,J) - A(I,K) * A(K,J)
    DO J = KK+TK, N
        DO I = KK+1, N
            DO K = KK, MIN(MIN(N,KK+TK-1),I-1)
                A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

⇓ ⇓ ⇓

(c) lud2d: LUD Tiled in Two Dimensions

```
DO JJ = 1, N, TJ
    DO II = 1, N, TI
        DO K = 1, N
            DO J = MAX(K+1,JJ), MIN(N,JJ+TJ-1)
                DO I = MAX(K+1,II), MIN(N,II+TI-1)
                    IF ((JJ≤K+1) & (K+1≤JJ+TJ-1) &
                        (J = MAX(K+1,JJ))
                        A(I,K) = A(I,K)/A(K,K)
                    END IF
                    A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

Figure 8: