A Quantitative Analysis of Tile Size Selection Algorithms[*]

Chung-Hsing Hsu and Ulrich Kremer

Department of Computer Science

Rutgers University, Piscataway, NJ 08855

keywords: compiler optimizations, memory hierarchy optimization, loop tiling, array padding, performance models, quantitative case study

Abstract

Loop tiling is an effective optimizing transformation to boost the memory performance of a program, especially for dense matrix scientific computations. The magnitude and stability of the achieved performance improvements are heavily dependent on the appropriate selection of tile sizes. Many existing tile selection algorithms try to find tile sizes which eliminate self-interference cache conflict misses, maximize cache utilization, and minimize cross-interference cache conflict misses. These techniques depend heavily on the actual layout of the arrays in memory. Array padding, an effective data layout optimization technique, is therefore incorporated by many algorithms to help loop tiling stabilize its effectiveness by avoiding "pathological" array sizes.

[*]This work has been partially supported by NSF grant CCR-9808607. Authors' contact information: Chung-Hsing Hsu, Los Alamos National Laboratory, chunghsu@lanl.gov; Ulrich Kremer, Department of Computer Science, Rutgers University, uli@cs.rutgers.edu

In this paper we examine several such combined algorithms in terms of cost-benefit trade-offs, and introduce a new algorithm. The preliminary experimental results show that more precise and costly tile selection and array padding algorithms may not be justified by the resulting performance improvements since such improvements may also be achieved by much simpler and therefore less expensive strategies. The key issues in finding a good tiling algorithm are (1) to identify critical performance factors and (2) to develop corresponding performance models that allow predictions at a sufficient level of accuracy. Following this insight, we have developed a new tiling algorithm that performs better than previous algorithms in terms of execution time and stability, and generates code with a performance comparable to the best measured algorithm. Experimental results on two standard benchmark kernels for matrix multiply and LU factorization show that the new algorithm is orders of magnitude faster than the best previous algorithm without sacrificing stability and execution speed of the generated code.

1 Introduction

As the speed of modern microprocessors has increased much faster than the memory speed, the memory traffic has become a key performance factor. As a result, effectively keeping reused data in the cache reduces the memory traffic and thus improves program performance. Cache performance is usually evaluated in terms of cache misses (or miss rate) which, according to the causes, can be classified as either *compulsory misses* or *replacement misses*[12]. A compulsory miss occurs if the first access to a block is not in cache. If a block in the cache is evicted and later retrieved, the miss is called a replacement miss. Replacement misses can be sub-categorized as *capacity misses* and *conflict misses* [16, 15]. Most compiler optimizations for improving cache performance have focused on reducing capacity misses, conflict misses, or both. A recent study showed that both capacity misses and conflict misses are equally important in determining the cache performance [24].

Loop tiling [38, 39, 21, 8, 26, 32, 34, 19, 6] is a well-known compiler optimization that partitions the iteration space of a loop nest into *tiles* (or *blocks*) to avoid replacement misses of those array elements frequently referenced during the computation involving the tile. Early efforts have been to select the tile in such a way that its working set fits into the cache to eliminate capacity misses, and its size is maximized to minimize loop overhead [3]. Significant work has been done to quantify the size of the working set [10, 29, 7].

Recent work takes conflict misses into account as well. With respect to tiling, conflict misses are classified as either *self* conflict misses, i.e., misses due to array elements of the same tile, or *cross* conflict misses, i.e., misses due to elements of different tiles. In addition to eliminating capacity misses and maximizing cache utilization, the tile is selected in such a way that there are no (or few) self conflict misses, and cross conflict misses are minimized [21, 9, 8, 37, 32, 6]. Some work has been done to quantify the total number of conflict misses [35, 13, 14, 11, 12].

Unfortunately, the performance of a tiled program resulting from existing tiling heuristics shows a large amount of instability [32, 28]. Instability comes from the so-called *pathological* array sizes [4, 10, 21, 2] which result in poor choices of tile sizes. Array padding [1, 22, 23, 30, 31] is a compiler optimization that increases the array sizes and initial locations to avoid the pathological cases. It introduces space overhead but effectively stabilizes program performance. As a result, more recent research efforts have investigated the combination of both loop tiling and array padding in the hope that both magnitude and stability of performance improvements of tiled programs can be achieved at the same time [32, 28, 20].

Example: Figure 1 shows the original implementation of matrix multiplication and a version of tiled and padded program. The loop header `do kk=1,n,`w

in the tiled version indicates that the loop counter `kk` starts from `1` and ends when it hits `n` with increments of w. The loop order for the original version is determined by applying loop order optimizations as proposed by McKinley et al. [25]. The iteration space spanned by `k`-loop and `i`-loop is tiled. As a result, a tile size $h \times w$ of array `a` is reused across iterations of the `j`-loop. Careful selection of the tile size will realize the data reuse and thus improve the program performance.

```
real a(n,n),b(n,n),c(n,n)

do j=1,n
  do k=1,n
    do i=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
```

```
real a(n+Δ,n),b(n,n),c(n,n)

do kk=1,n,w
  do ii=1,n,h
    do j=1,n
      do k=kk,min(kk+w-1,n)
        do i=ii,min(ii+h-1,n)
          c(i,j)=c(i,j)+a(i,k)*b(k,j)
```

Figure 1: Tiled and padded matrix multiplication: What is the best choice of h, h, and Δ to exploit data reuse?

In this paper, a subset of published tile size selection algorithms are examined in a single framework – select from a set of candidate tile sizes the one that minimizes a particular cost model. These algorithms can be distinguished by their different choices of candidate tile sizes and cost models. The choices may affect the efficiency of an algorithm, the magnitude of the resulting performance improvement, and the stability of the improvement over different problem sizes.

Common performance factors in these choices are then identified. They are discussed in terms of criticality in precision to the effectiveness of performance optimization. Such information is crucial for any programmer or compiler writer who wants to use loop tiling to improve program performance. In addition, the insights into the performance characteristics of previous algorithms can be used to improve existing tiling algorithms, or develop new algorithms such as the new tile size and pad size selection algorithm discussed in this paper. The new algorithm gives priority to performance factors differently, and uses different levels of approximations for these performance factors. Specifically, most previous algorithms search for tile sizes that eliminate self conflict misses, maximize cache utilization, and minimize cross conflict misses. In contrast, the new algorithm searches for tile sizes that eliminate TLB misses, have a reasonable cache utilization, and generate few conflict misses.

To validate the findings, the new algorithm and a set of popular tile size selection algorithms are used to produce tiled codes which are then executed on real machines. Two benchmarks, matrix-matrix multiplication and LU factorization, are selected since all published tile size selection algorithms use them to demonstrate the effectiveness of their algorithms. Experimental results show that the new algorithm generates code with a performance and stability comparable to the best existing tiling algorithms, but at a much lower cost in terms of both space and execution time overheads.

In summary, the main contributions of the paper include

- A case study showing that more precise and costly models may not be justified by the resulting performance improvements since much simpler and faster models can achieve comparable effectiveness.
- A new tile selection algorithm that selects tiles very fast, and effectively boosts and stabilizes performance while using only small pad sizes.
- A discussion of several critical performance factors and possible approximation models.

The rest of the paper is organized as follows. Section 2 presents a more detailed description of the tested algorithms. Several performance factors are identified and discussed in Section 3. In addition, a new algorithm based on the findings is presented. A discussion of our experimental results and findings can be found in Section 4. Conclusions and future work are presented in the last section.

2 Tile Size Selection Algorithms

Most tile size selection algorithms choose from a set of candidate tile sizes ($\mathcal{T}$) the one (t) that minimizes a particular cost model, i.e.,

$$\arg_t \min\{\mathbf{cost}(t) |\ t \in \mathcal{T}\} \tag{1}$$

For simplicity, only square arrays of size $n \times n$ and rectangle tile sizes are considered in this paper. Since a rectangle tile size can be described in terms of its height (h) and width (w), the tile size selection problem is reduced to determine the values of h and w that minimizes the cost model:

$$\arg_{h\times w} \min\{\mathbf{cost}(h \times w) |\ h \times w \in \mathcal{T})\} \tag{2}$$

where

$$\mathcal{T} \subseteq \{h \times w |\ 1 \leq h, w \leq n\} \tag{3}$$

In the following, we refer to the cache size as C, cache block size as b, and cache associativity as a. All values are defined in terms of array elements, and the array is assumed to be stored in column-major order. Furthermore, we focus on the cases where the array size is too large to fit entirely in the cache ($C < n^2$), but its single column can completely fit in ($n \leq C$).

Many tile size selection algorithms concentrate on the tile sizes in $\mathcal{T}$ being non-conflicting. A tile size is *non-conflicting* if it does not generate any self conflict misses [32]. Whether or not a tile size is non-conflicting depends on

Notation	Interpretation
C	cache size
b	cache block size
a	cache associativity
n	the size of a two-dimensional array
P	page size
E	total number of entries in TLB
$h \times w$	a rectangle tile with height h and width w
$\mathbf{h} \times \mathbf{w}$	the selected tile size
$\mathcal{N}(C, b, a, n)$	the set of all non-conflicting tile sizes
$\mathcal{M}(C, b, a, n)$	the set of all maximal non-conflicting tile sizes
Δ	pad size

Table 1: The summary of notations used in defining the tile size selection algorithms. All values are defined in terms of array elements.

the cache organization (C, b, a) and the array size (n). As a result, the set of all non-conflicting tile sizes is denoted as $\mathcal{N}(C, b, a, n)$. For algorithm efficiency, almost all algorithms using non-conflicting tile sizes restrict their candidates to the maximal ones. A non-conflicting tile size is *maximal* if neither its height nor its width may be increased without causing self conflict misses. The set of maximal non-conflicting tile sizes is in general much smaller than the set of all non-conflicting tile sizes, i.e.,

$$\mathcal{M}(C, b, a, n) \subseteq \mathcal{N}(C, b, a, n) \tag{4}$$

These notations are summarized in Table 1.

Table 2 shows a number of tile size selection algorithms for matrix-matrix multiplication, extracted from the referenced papers. It can be seen that all presented algorithms differ in the set of candidate tile sizes and/or the cost

Algorithm	$\mathcal{T}$	$\mathbf{cost}(h \times w)$
ess [9]	$\{h \times w \mid h = n, h \times w \in \mathcal{M}(C,1,1,n)\}$	$C/(h * w)$
lrw [21]	$\{b \times b \mid b = \min(h,w), h \times w \in \mathcal{M}(C,1,1,n)\}$	$1/h + 1/w + (2h+w)/C$
tss [8]	$\{(\lfloor h/l \rfloor l) \times w \mid h * w + h + w \leq C, h \times w \in \mathcal{M}(C,1,1,n)\}$	$(2h+w)/(h*w)$
euc [32]	$\{(h - l + 1) \times w \mid h \times w \in \mathcal{M}(C,1,1,n)\}$	$1/h + 1/w$
moon [27]	$\{h \times w \mid h \times w \in \mathcal{M}(C,b,1,n)\}$	$1/h + 1/w + (h+w)/C$
tli [6]	$\{h \times w \mid h \times w \in \mathcal{M}(C,b,1,n)\}$	$1/h + 1/w + (h+w)/C + h*w/C^2$
wmc [37]	$\{h \times w \mid h = w, h * w \leq \alpha * C\}$	$C/(h * w)$
mhcf [26]	$\{h \times w \mid hw(1/h + 1/l + 1/n) \leq \alpha * C\}$	$(1/h + 1/w)(1/n + 1/l) + 2/(h*w)$
eucPad [32]	$\{(h - l + 1) \times w \mid h \times w \in \mathcal{M}(C,1,1,n+\Delta), 0 \leq \Delta \leq 8\}$	$1/h + 1/w$
tliPad	$\{h \times w \mid h \times w \in \mathcal{M}(C,b,1,n+\Delta), 0 \leq \Delta \leq 8\}$	$1/h + 1/w$
datPad [28]	$\{h \times w \mid h = w, h \equiv 0 \pmod{b}, h * w + h + w \leq C\}$ $\mathbf{h} \times \mathbf{w} \in \mathcal{N}(C,b,a,n+\Delta)\}$	$C/(h * w)$ Δ

Table 2: The candidate tile sizes and the cost models of different tile size (and pad size) selection algorithms for matrix-matrix multiplication. The second column $\mathcal{T}$ represents the set of candidate tile sizes considered by each algorithm. The third column $\mathbf{cost}(h \times w)$ indicates the cost model used by each algorithm for the tile size $h \times w$. Underlined algorithms are used for our quantitative comparison. Note that the original **mhcf** algorithm uses multi-level cost models while only one-level formulation is presented here for simplicity.

---

models they use. In general, most algorithms search for the largest tile sizes that generate the least amount of capacity misses and conflict misses. Sometimes, high cache utilization and low cache misses may not be achieved simultaneously. As a result, we can consider each algorithm as a different way to approximate cache utilization and number of cache misses, and to weigh between these two quantities.

To find tile sizes that have few capacity misses, many algorithms restrict their candidate tile sizes to be the ones whose working set can entirely fit in the cache (e.g., $h * w \leq C$). To model self conflict misses due to low associativity cache, some algorithms such as **wmc** and **mhcf** use the *effective* cache size ($\alpha * C$), while others explicitly find the non-conflicting tile sizes. Note that these non-conflicting tile sizes, by the definition, do not generate capacity misses, i.e.,

$$\mathcal{N}(C, b, a, n) \subseteq \{h \times w | \ h * w \leq C\} \tag{5}$$

In summary, the surveyed algorithms tend to focus their attention on tile sizes that eliminate both capacity misses and self conflict misses. The minimization of cross conflict misses and maximization of cache utilization are modeled in the cost function, usually in terms of the cross conflict miss *rate*.

For algorithms using explicitly the non-conflicting tile sizes, $\mathcal{M}(C, b, a, n)$ is approximated differently. Algorithms **moon** and **tli** find the *exact* set of all non-conflicting tile sizes with respect to cache block size b. In contrast, algorithms such as **tss** and **euc** approximate the set by always using $b = 1$. The approximation leads to a simple formula $\mathcal{M}(C, 1, 1, n) \equiv \{h_i \times \min(w_i, n) : i \geq 1\}$ such that

$$\begin{array}{lll} h_0 = C, & h_1 = n, & h_{i+2} = h_i \bmod h_{i+1} \\ w_0 = 1, & w_1 = \lfloor C/n \rfloor, & w_{i+2} = \lfloor h_i/h_{i+1} \rfloor * w_{i+1} + w_i \end{array}$$

The resulting algorithm is very fast, but the selected tile size is not guaranteed to be non-conflicting. In contrast, algorithms such as **tli** propose simulation-based methods to find the exact set for arbitrary b, which is computationally more expensive. Table 3 gives an example illustrating various approximations of $\mathcal{M}(C, b, a, n)$.

The desired tile shape (h/w) has been *explicitly* specified in algorithms such as **lrw**, **ess**, and **wmc**. Both **lrw** and **wmc** search for square tiles ($h = w$). In contrast, **ess** finds extremely tall tiles ($h = n$). Tile shape can be implicitly favored through the cost model. A more detailed discussion can be found in Section 3.

$\mathcal{M}(2048,4,1,127)$	$\mathcal{M}(2048,1,1,127)$	**euc**
$\{$ 127 $\times$ 12 ,	$\{$ 127 $\times$ 16 ,	$\{$ 124 $\times$ 16 ,
113 $\times$ 16 ,	16 $\times$ 113 ,	13 $\times$ 113 ,
16 $\times$ 124 ,	15 $\times$ 127 ,	12 $\times$ 127 $\}$
1 $\times$ 127 $\}$	1 $\times$ 127 $\}$	

Table 3: An example illustrating various approximations of $\mathcal{M}(C, b, a, n)$, the set of maximal tiles that have no self conflict misses. The reported sets are for $(C, b, a, n) = (2048, 4, 1, 127)$.

2.1 Array Padding Extension

Loop tiling alone is in general effective in improving program performance. However, there are certain array sizes that will introduce severe cache conflict misses and deteriorate the performance improvement. Such array sizes are usually near the power of two. Array padding [1, 22, 23, 30, 31] is a data layout transformation that sets a dimension in an array to a new size to reduce the number of conflict misses. For example, the transformed code in Figure 1 has the leading dimension of its array `a` padded with pad size Δ. With the appropriate value of Δ for each array size `n`, array elements in a tile are mapped to the different cache lines, and therefore avoid the performance degradation due to severe conflict misses. In short, with the help of array padding, loop tiling is guaranteed to be effective across all problem sizes. We say it has *stable* performance improvement.

From tile size selection perspective, the array sizes that cause severe conflict misses constrain the choice of candidate tile sizes when the non-conflictingness is a desired feature. Usually the set of candidate tile size for these array sizes are *ill-shaped*, i.e., either they are extremely long or extremely wide. It has been observed that ill-shaped tile sizes cannot boost program performance. As an example, consider the set of candidate tiles for **euc** when array size n is 127, as shown in Table 3. It can be seen that there is no tile whose shape is near square, and therefore **euc** is constrained to select 124×16 as its best choice. If the leading dimension of the array is increased with pad size $\Delta = 5$, then **euc** is able to find 61×31 as its best choice.

Since $\mathcal{M}(C, b, a, n)$ is sensitive to the array size n, Rivera and Tseng [32] proposed an extension **eucPad**, which allows to increase the array size by at most 8 elements in the hope of finding a "better" tile. By substituting different approximations for $\mathcal{M}(C, b, a, n)$, we can evaluate the impact of the quality (preciseness) of approximations to $\mathcal{M}(C, b, a, n)$ on the tiling effectiveness.

Algorithm **datPad** [28] takes a very different approach. Unlike **eucPad** which simply uses padding to enlarge the set of tiles for selection, **datPad** finds the largest tile with a specified (program-dependent) shape, and then uses padding to eliminate self conflict misses. This algorithm considers different cache block sizes.

3 Performance Factors and A New Algorithm

Tile *shape* (h/w) and cache utilization ($h*w/C$) are two important performance factors considered by many algorithms, either implicitly through the cost model or explicitly through candidate tiles. In addition, many have observed that there is a "tension" between tile shape and cache utilization [21, 8, 27, 32, 18]. Extremely wide tiles may introduce severe TLB thrashing. On the other hand, extremely tall or square tiles may have low cache utilization.

For example, **lrw** will select the non-conflicting tile size 4×4 for case $(C, b, a, n) = (2048, 1, 1, 512)$, which has a very low cache utilization of 0.78%. Algorithms such as **ess**, **tss** and **euc** allow rectangle tile shape in the hope to avoid the problem. For the same configuration, **euc** is able to select tile size 512×4 instead and boosts the cache utilization up to 100%. However, if array size n is changed to 516, **euc** will select tile size 16×127. Though the cache utilization is as high as 99.22%, this tile size requires 127 entries in the page table to keep it in the cache, given a page size of say 4KB. In contrast, **ess** selects tile size 516×3, an overly tall tile size. Though it will not introduce TLB thrashing, it has only 75.59% cache utilization. As a result, surveyed algorithms have different ways to weigh these performance factors.

Algorithms **lrw**, **ess**, and **datPad** give priority to tile shape over cache utilization. For **datPad**, the best tile shape is program dependent. For example, it determines that the best tile shape for matrix multiplication is square ($h/w = 1$) and the best tile shape for LU decomposition is $\frac{h}{w} = b$. In contrast, **euc** suggests the same tile shape for both benchmarks because they are all linear algebra codes.

Tile shape is sometimes implicitly preferred through the cost model. For example, **euc**'s cost model $1/h + 1/w$ favors square tiles over non-square ones with the same area. For linear algebra codes, the memory access pattern within a tile is usually due to several array references. To model the effect of tile $h \times w$ interfering with two regions of sizes $h \times 1$ and $1 \times w$, we can estimate the probability of cross conflict misses in terms of footprint overlap, as done in [21, 8]. As a result, **euc**'s cost model $\frac{h+w}{h*w}$ is derived. That is, a "good" tile shape not only avoids TLB thrashing and low cache utilization but also minimizes cross conflict miss rate.

TLB thrashing have been explicitly considered in **tss** and a version of **mhcf** with multi-level cost functions. The complicated version of **mhcf** finds tile sizes that has no TLB misses, few cache misses, and minimize the (program dependent) cost model [26]. It can be done implicitly as well. For example, the constraints of $h/w = 1$ and $h * w \leq C$ imply that $h \leq \sqrt{C}$. The value $\sqrt{C}$ may always be smaller than the page table size. We feel that TLB thrashing is a crucial performance factor since a TLB miss costs more than a cache miss and may cause cache stalls.

Finally, we want to mention that cache associativity has been taken into account by **datPad** but not other algorithms surveyed in this paper. Algorithm **datPad** uses cache associativity to adjust the effective cache size (i.e., $\alpha = \frac{a-1}{a}$). In our target machines, we carefully chose one that has a 4-way set associative cache to evaluate the importance of this performance factor.

3.1 A New Algorithm

Based on the discussion above, we propose in this paper a new algorithm **new-Pad** with the following three guidelines:

- The best tile size has *few* conflict misses and *good* cache utilization.
- The best tile size eliminates TLB misses.
- The padding choices should not be fixed a priori.

These guidelines reorder the priorities of performance factors. Specifically, instead of searching for tile sizes that eliminate self conflict misses, maximize cache utilization, and minimize cross conflict misses, as most previous algorithms do, the new algorithm looks for tile sizes that eliminate TLB misses, generate few conflict misses, and have a reasonable cache utilization. The new algorithm is greedy in that it increases pad size until a "qualified" tile size is found with the current pad size. If there are more than one such tile sizes, the algorithm will select the one that minimizes the cost model. The pseudo code of the algorithm is presented in Figure 2, with parameters **cost**$(h \times w)$ and **good**$(h \times w)$.

In Figure 3, a possible formulation of **good**$(h \times w)$ and **cost**$(h \times w)$ is presented. A good tile size allows only part of TLB entries to be touched by the entire tile to avoid TLB thrashing. In a way it restricts the possible values of w. In addition, its area should be sufficiently large to guarantee a reasonable cache utilization. Finally, its shape should not be "too far away" from the "optimal" shape $h/w = b$ (explained below).

```
$\Delta \leftarrow 0$
repeat
    $\mathbf{h} \times \mathbf{w} \leftarrow \arg\min\{\mathbf{cost}(h \times w) : \mathbf{good}(h \times w),\ h \times w \in \mathcal{M}(C, 1, 1, n + \Delta)\}$
    $\Delta \leftarrow \Delta + 1$
until $\mathbf{h} \times \mathbf{w}$ exists
```

Figure 2: The skeleton of new tiling algorithm **newPad**

The cost model of **newPad** is similar to **euc**'s cost model $\frac{h+w}{h*w}$, but **newPad** takes into account the effect of large cache block size ($b > 1$). As a result, the cost model becomes $\frac{h/b+w}{(h/b)w} = \frac{b}{h} + \frac{b}{w}$. A consequence of using the new cost model is that the optimal tile shape, instead of square, becomes $h/w = b$, a *rather long* tile shape. Since tile shape implicitly estimates the number of cross conflict misses, bounding the tile shape and making sure it comes from $\mathcal{M}(C, 1, 1, n + \Delta)$ will introduce limited conflict misses.

Among all presented algorithms, **newPad** is closest to **datPad**. As a matter of fact, **datPad** can be considered as a specialization of **newPad** by providing *more strict* definitions of a good tile shape and cache utilization. Both of them do not fix the possible pad sizes and may therefore avoid cases where the fixed-amount pad choice strategy can only select "bad" tile sizes. Algorithm **newPad** considers TLB thrashing crucial, while **datPad** and **eucPad** do not. Table 4 summarizes the different characteristics of the algorithms tested in our experimental study. The final tile selections computed by these algorithms for the example in Table 3 are listed in Table 4 as well.

We do not claim that our formulation of **good**$(h \times w)$ and **cost**$(h \times w)$ is the *optimal* choice. However, we believe that our model reflects the importance of the different performance factor correctly. In addition, for any given tile size $h \times w$, there always exists a pad size with no self conflict misses, for example,

Algorithm	Non-conflicting	Pad choices	Selected tile
org	No tiling		
ess	guaranteed	0	127 × 16 (0)
lrw	almost	0	16 × 16 (0)
euc[32]	almost	0	124 × 16 (0)
eucPad[32]	almost	0-8	61 × 31 (5)
tliPad[6]	guaranteed	0-8	32 × 63 (3)
datPad[28]	guaranteed	unlimited	44 × 44 (55)
newPad	almost	unlimited	98 × 16 (3)

Table 4: Tiling heuristics used for our experimental study. The rightmost column shows the final selections of various tile selection algorithms for the example in Table 3. The values in parentheses are the final pad sizes.

$\Delta = iC + h$ [20]. As a result, **newPad** will always terminate.

4 Experimental Evaluation

To compare different tile size selection algorithms, the array size n was varied from 100 to 1100 double-precision data elements with a step size of 4 elements. For each algorithm, two linear algebra benchmark kernels, namely matrix-matrix multiplication (`mm`) and LU-factorization without pivoting (`lu`), were executed on three target architectures with different cache organizations. The two benchmarks have been used by many published papers [36, 21, 8, 32, 28] to evaluate the effectiveness of their tile selection algorithms. For machines with multi-level caches (and thus multiple cache sizes), Rivera and Tseng have found that nearly all the benefits can be achieved by simply targeting the first-level cache, provided that the next level cache is much larger [33]. Therefore, the discussed algorithms assume tile size selection of the first-level data cache.

$$\begin{aligned}
\mathbf{cost}(h \times w) &= b/h + 1/w \\
\mathbf{good}(h \times w) &= \min(n/P, 1) * w \leq \beta * E (\text{no TLB misses}) \\
& \quad h * w \geq \alpha * C (\text{good cache utilization}) \\
& \quad |\, \mathbf{shape}(h \times w) - b| \leq (b+1)/2 (\text{few cache misses}) \\
\mathbf{shape}(h \times w) &= \begin{cases} h/w & \text{if } h \geq w \\ 2 - h/w & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3: The implementation of **newPad** ($\alpha = \beta = 0.75$)

Machine	CPU		Data Cache			TLB	
	Type	MHz	C	b	a	E	P
Ultra-1	UltraSparc	143	16KB	32B	direct	64	8KB
SS5	MicroSparc-II	110	8KB	16B	direct	64	4KB
SS20/71	SuperSparc-II	75	16KB	32B	4-way	64	4KB

Table 5: Different target architectures

Each kernel was compiled by SUN's SparcCompiler 5.0 `f77` compiler with the `-O` switched on, and executed in five runs. The minimum execution time, excluding the time for data initialization, was then reported. This way, we can minimize abrupt noises and file cache effects. The loop orders for the untiled versions were determined by applying loop order optimizations as proposed by McKinley et al. [25]. All experimental results are represented in terms of MFLOPS. The three target architectures are shown in Table 5.

4.1 Performance and Stability

Figure 4 and Figure 5 plot the achieved MFLOPS rates (y-axes) for each of the algorithms and target machines for our 251 input array sizes ranging from 100 to 1100 by a step of 4 (x-axes). We are concerned with the quality of each algorithm in terms of its performance improvement magnitude and stability. Table 7 summarizes the results by reporting average performance and the standard deviation over all problem sizes. The following observations can be made:

- **lrw** and **euc** have similar performance improvement, which is better than **ess**.
- **euc** has a significant number of pathological cases which degrade performance improvement (note the downward peaks in **euc**).
- Array padding can effectively stabilize the performance improvement (for instance, compare **eucPad** against **euc**).
- Compared with **tliPad**, **eucPad** has similar performance improvement but slightly better stability.
- **newPad** and **datPad** have similar performance improvement and stability. The stability of both algorithms is significantly better than **eucPad** and **tliPad**.

In summary, the above algorithms have similar performance improvement magnitude. While array padding helps stabilize the performance improvement, only **datPad** and **newPad** have a consistent behavior over all tested array sizes.

4.2 Cost-Benefit Trade-Offs

newPad and **datPad** have similar effectiveness in stabilizing performance improvement due to loop tiling, at the cost of padding dummy array elements.

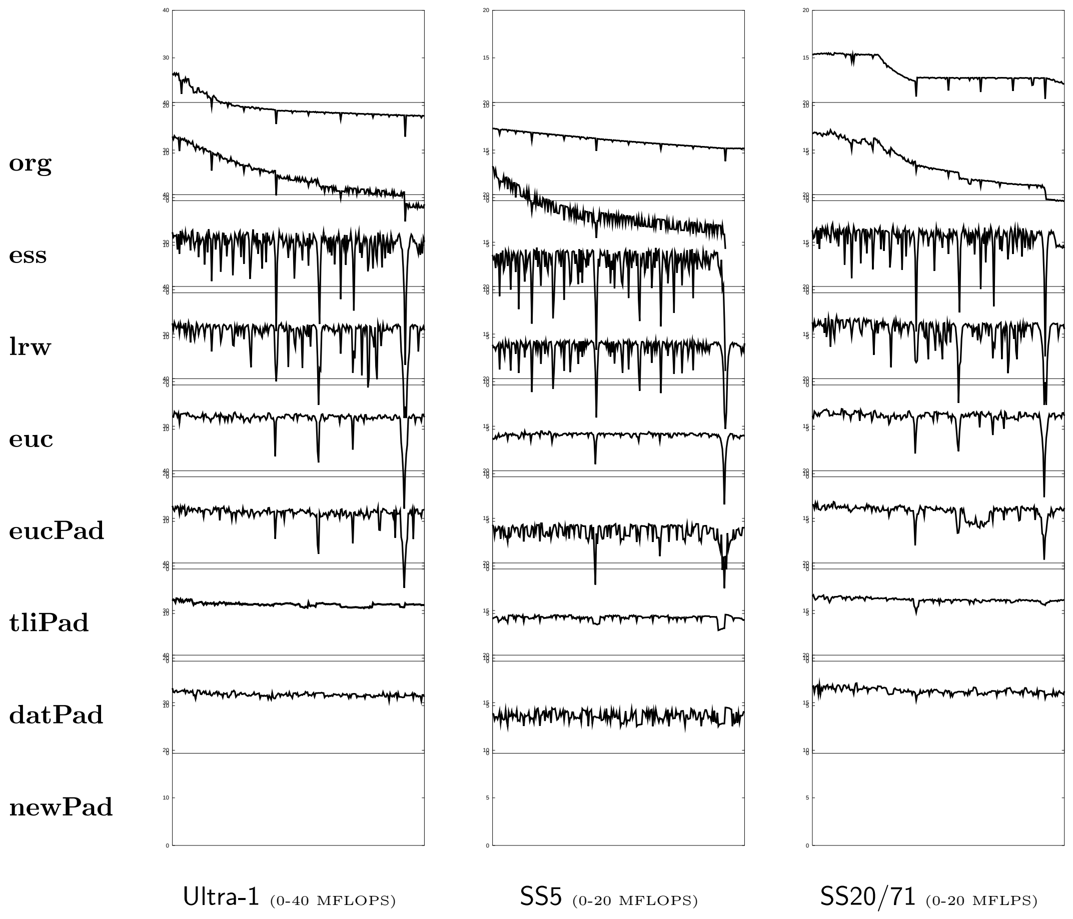
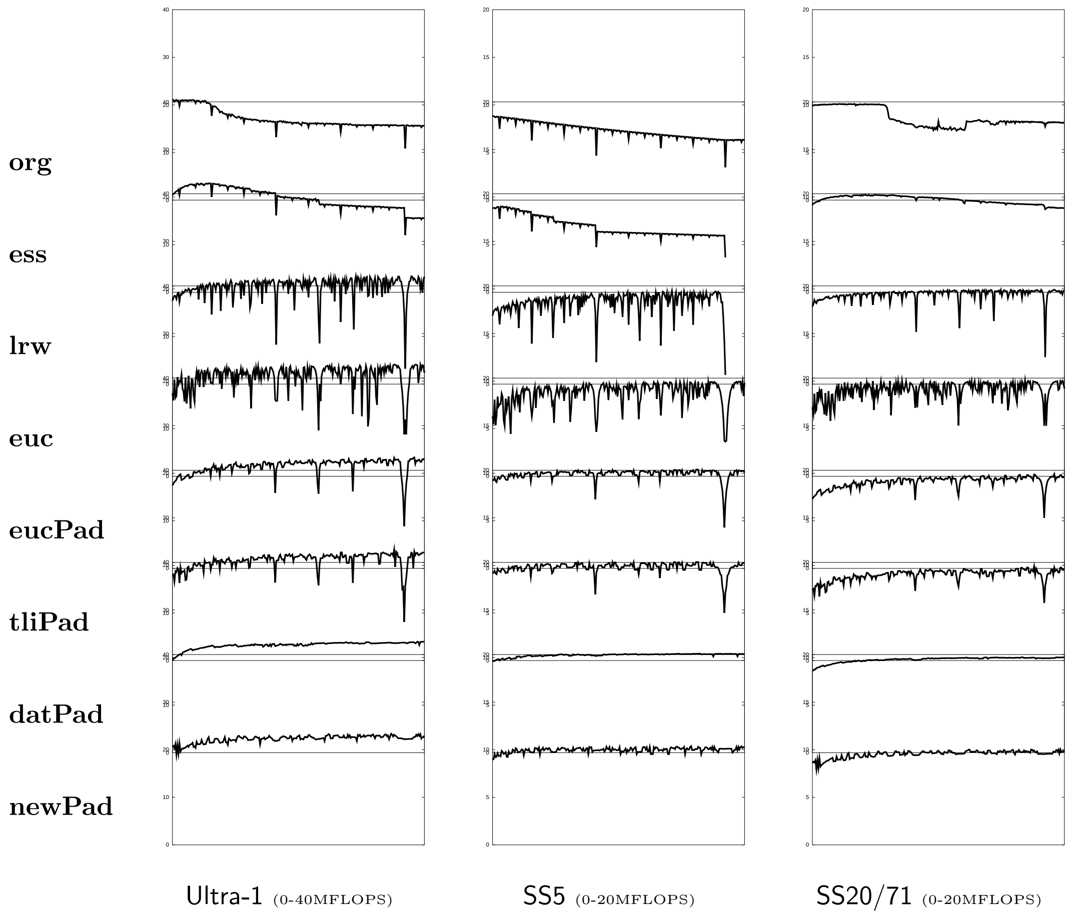


Figure 4: Matrix multiplication. The x-axes are problem sizes, the y-axes MFLOPS.



LU-factorization w/o pivoting. The x-axes are problem sizes, the y-axes MFLOPS.

Figure 5: Experimental Results

	Ultra-1,SS20/71	SS5
matrix multiplication (`mm`)		
eucPad	3.98 (2.73)	3.92 (3.00)
tliPad	3.89 (2.89)	3.94 (2.99)
datPad	66.58 (46.68)	19.81 (16.54)
newPad	4.96 (8.43)	3.30 (7.21)
LU-factorization (`lu`)		
datPad	51.76(35.40)	19.43(15.89)
all others	same as for `mm`	

Table 6: Analysis of space overhead due to padding. All values are represented in terms of final pad sizes in the leading array dimension. They are the average and standard deviation of pad sizes of 251 input array sizes. The values in the parenthesis are standard deviations.

With respect to the space overhead due to array padding, an analysis reveals that on average **newPad** introduces slightly more space overhead than **eucPad** but much less than **datPad**, as shown in Table 6. Compared to **eucPad**, the deviation by **newPad** is slightly higher. It is the price to be paid for "unlimited" padding.

Another dimension of the costs of a tile size selection algorithm is its execution time. Algorithms **eucPad** and **newPad** are based on the Euclidean GCD computation and are therefore fast to compute. **datPad** performs memory-based simulation to find the smallest pad size that ensures no self conflict misses in the selected tile. **tliPad** uses a clever "simulation" strategy by searching the tiles as a computation-based incremental process. Our data show that on average **eucPad** and **newPad** all take approximately 80 microseconds, **tliPad** takes 3.68 seconds, and **datPad** takes 0.057 seconds. That is, algorithms **tliPad** and

datPad are executed several orders of magnitude (6× and 3×, respectively) slower than the Euclidean GCD based algorithms. The long execution time of **tliPad** is due to the fact that the algorithm does not know a priori the final tile and pad size, and enumerates all possible choices to determine the best one. In contrast, **datPad** has a predetermined tile shape and thus can quickly compute the largest tile size with this shape and reject inappropriate pad sizes.

The preliminary experimental data suggest that more precise and costly tile selection and array padding may not be justified by the resulting performance improvements since such improvements can also be achieved by simpler, more approximate and therefore cheaper models. The quality of a tile selection algorithm is determined by its ability to identify critical performance factors and the degree in which such factors need to be approximated through performance models.

Experimental results show that less strict definitions achieve comparable magnitude and stability of performance improvements with *significantly smaller* pad sizes.

5 Conclusions and Future Work

We have presented several algorithms combining tile selection and array padding. A new tile selection algorithm has been introduced. The new algorithm and other published algorithms were evaluated in terms of the magnitude and stability of the performance improvement, the space overhead introduced by padding, and the time for tile selection. The experiments showed that the new algorithm generates tiles of comparable performance and stability as the best of our tested algorithms, but has a significant lower space overhead (factor of 7) and selection time (up to three orders of magnitude). We have found that the *cost-benefit balance* is a key in designing such an effective, yet inexpensive tile selection algorithm. We have observed that more precise and costly tile selection and array

	Ultra-1	SS5	SS20/71
matrix multiplication (`mm`)			
org	19.59 (2.23)	6.35 (0.65)	13.58(1.17)
ess	24.44 (4.01)	8.33 (1.65)	13.33 (2.17)
lrw	29.97 (3.39)	12.71 (1.87)	15.31 (1.64)
euc	29.97 (3.39)	13.40 (1.36)	15.34 (1.44)
eucPad	31.53 (2.09)	14.02 (0.62)	15.97 (0.87)
tliPad	30.90 (1.96)	13.54 (0.94)	15.69 (0.78)
datPad	31.30 (0.40)	14.24 (0.28)	16.12 (0.23)
newPad	31.69 (0.43)	13.58 (0.52)	16.24 (0.30)
LU-factorization (`lu`)			
org	17.02 (1.86)	7.29 (0.83)	8.62 (0.96)
ess	19.68 (2.18)	6.91 (1.04)	9.70 (0.41)
lrw	20.40 (2.34)	8.61 (1.20)	9.42 (0.69)
euc	20.85 (2.78)	8.70 (1.25)	9.04 (0.95)
eucPad	21.45 (1.70)	9.90 (0.59)	9.11 (0.60)
tliPad	21.03 (1.71)	9.79 (0.58)	9.02 (0.59)
datPad	22.61 (0.67)	10.26 (0.17)	9.79 (0.28)
newPad	22.26 (0.76)	9.97 (0.25)	9.54 (0.37)

Table 7: Average performance in MFLOPS. The values in parentheses are the standard deviation.

padding may not be justified by the resulting performance improvement since such improvements may also be achieved by much simpler and cheaper models.

A few performance factors for tile selection have been identified and discussed. We found that: (1) TLB thrashing effect is critical, (2) tile shape is important, but squareness is not critical, (3) tile area is important, but largest cache utilization is not critical, and (4) set associativity is not important. These observations hold for the two prevalent benchmark kernels `mm` and `lu`.

In the future, we are planning to perform experiments on a wider range of benchmark programs and underlying architectures. In addition, we are interested in investigating the impact of other program transformations such as tiling for multiple arrays and register tiling [5] on tile selection. Finally, we want to evaluate the possibility of *automatic* construction of effective, yet low-cost models [17].

References

[1] D. Bacon, J.-H. Chow, D.-C. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94 – Integrated Solutions*, pages 270–282, October 1994.

[2] D. Bailey. Unfavorable strides in cache memory systems. *Scientific Programming*, 4(2):53–58, Summer 1995.

[3] F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis. A quantitative algorithm for data locality optimization. In Robert Giegerich and Susan Graham, editors, *Code Generation: Concepts, Tools, Techniques*, pages 119–145. Berlin: Springer Verlag, 1992.

[4] D. Callahan and A. Porterfield. Data cache performance of supercomputer applications. In *Supercomputing '90*, pages 564–572, November 1990.

[5] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 349–357, December 1997.

[6] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *1999 ACM International Conference on Supercomputing*, June 1999.

[7] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *1996 ACM International Conference on Supercomputing*. ACM, May 1996.

[8] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, California, 18–21 June 1995.

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

[10] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *1991 Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, 1991.

[11] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *1997 ACM International Conference on Supercomputing*, pages 317–324, New York, July7–11 1997. ACM Press.

[12] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, July 1999.

[13] J. Harper, D. Kerbyson, and G. Nudd. Predicting the cache miss ratio of loop-nested array references. Technical Report CS-RR-336, Department of Computer Science, University of Warwick, Coventry, UK, December 1997.

[14] J. Harper, D. Kerbyson, and G. Nudd. Analytical modeling of set-associative cache behaviour. *IEEE Transactions on Computers*, 48(10):1009–1023, October 1999.

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, second edition, 1996.

[16] M.D. Hill and A.J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[17] C.-H. Hsu and U. Kremer. IPERF: A framework for automatic construction of performance prediction models. In *Workshop on Profile and Feedback-Directed Compilation (PFDC)*, Paris, France, October 1998.

[18] C.-H. Hsu and U. Kremer. Tile selection algorithms and their performance models. Technical Report DCS-TR-401, Department of Computer Science, Rutgers University, October 1999.

[19] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proceedings of the 13th International Conference on Supercomputing (ICS-99)*, 1999.

[20] F. Kuehndel. Software methods for avoiding cache conflicts. Technical Report CS-TR-98-16, University of Texas, Austin, September 1, 1998.

[21] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *4th International Conference on Architec-*

tural Support for Programming Languages and Operating Systems, pages 63–74, Santa Clara, Calif., April 1991.

[22] N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, 1995.

[23] N. Manjikian and T. Abdelrahman. Array data layout for the reduction of cache conflicts in loop nests. In Vincent Van Dongen, editor, *Proceedings of the High Performance Computing Symposium '95, Canada's Ninth Annual International High Performance Computing Conference and Exhibition*, July 1995.

[24] K. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, Massachusetts, October 1996. ACM Press.

[25] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[26] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6), December 1998.

[27] S. Moon and R. Saavedra. Hyperblocking: A data reorganization method to eliminate cache conflicts in tiled loop nests. Technical Report TR-98-671, Computer Science Department, University of Southern California, February 1998.

[28] P. Panda, H. Nakamura, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2), February 1999.

[29] W. Pugh. Counting solutions to presburger formulas: How and why. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 94.

[30] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998.

[31] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *1998 ACM International Conference on Supercomputing*, pages 353–360, New York, July 13–17 1998. ACM press.

[32] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, March 1999.

[33] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *Supercomputing '99*, November 1999.

[34] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 215–228, May 1999.

[35] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271, New York, NY, USA, May 1994. ACM Press.

[36] M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ont., June 1991.

[37] M. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *The 29th Annual International Symposium on Microarchitecture*, pages 274–286, December 2–4, 1996.

[38] M. Wolfe. Iteration space tiling for memory hierarchies. In Gary Rodrigue, editor, *the 3rd Conference on Parallel Processing for Scientific Computing*, pages 357–361, December 1989.

[39] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Co., 1996.