A Stable and Efficient Loop Tiling Algorithm

Chung-Hsing Hsu and Ulrich Kremer*

Department of Computer Science
Rutgers University

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 is 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 stablize its effectiveness by avoiding "pathological" array sizes.

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 algorithms 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 algorithms 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

*C-H. Hsu, email: chunghsu@cs.rutgers.edu; U. Kremer (*corresponding author*), email: uli@cs.rutgers.edu, address: Department of Computer Science, Hill Center, Busch Campus, Rutgers University, Piscataway, NJ 08855

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*[11] 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* [17, 16]. 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 [25].

Loop tiling [38, 39, 22, 8, 26, 32, 34, 20, 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-conflicts*, i.e., cache conflicts among array elements of the same tile, or *cross-conflicts*, i.e., cache conflicts among 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 self-conflicts, and cross-conflicts are minimized [22, 9, 8, 37, 32, 6]. Some work has been done to quantify the total number of conflict misses [35, 14, 15, 12, 11].

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, 22, 2] which result in poor choices of tile sizes. Array padding [1, 23, 24, 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.

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, 21]. In this paper we discuss a new tile selection algorithm.

Unlike some other tiling algorithms, our new algorithm does not require the number of padding choices to be fixed a priori. Instead, all pad sizes are evaluated in increasing order until a *good* tile has been found. Most previous algorithms define good tiles to be those which eliminate self-conflicts, maximize cache utilization, and minimize cross-conflicts. In contrast, the new algorithm redefines good tiles as those which eliminate TLB misses, have *good* cache utilization, and *few* conflict misses. In other words, our new algorithm optimizes different performance factors, and uses different levels of approximations for these performance factors.

Experimental results for matrix multiply and LU factorization kernels 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 other words, the new algorithm suggests that more precise and costly tile selection and array padding may not be justified by the resulting performance improvement since such improvement may also be achieved by much simpler and therefore less expensive models. The key to developing our new algorithm was to identify critical performance factors and to evaluate the properties of their approximations models. In summary, the contributions of the paper are as follows:

- A case study showing that more precise and costly optimizations 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 paper is organized as follows. Section 2 presents a more detailed description of the tested algorithms, followed by a discussion of our new algorithm in Section 3. Our experimental testbed is described in Section 4, together with a discussion of experimental results and a summary of the findings. In addition, several performance factors are identified and discussed. Conclusions and future work are presented in the last section.

2 Tile Selection Algorithms

Most tile selection algorithms choose a tile from the set of maximal tiles that have no self-conflicts the one that minimizes a (program-dependent) cost model **cost** [19], i.e.,

$$\mathbf{h} \times \mathbf{w} \leftarrow \arg\min\{\mathbf{cost}(h \times w) : h \times w \in \mathcal{M}(C, l, n)\}$$

where M(C,l,n)} denotes the set of maximal tiles that have no self conflicts for a given cache capacity C, cache line size l, and array size n. The notations used in this paper are summarized in Table 1. A tile is *maximal* if neither its height nor its width may be increased without causing self-conflicts [32]. A maximal tile does not guarantee to have the largest tile area among all tiles that have no self-conflicts. As has been done by most researchers, we focus our attention on two-dimensional arrays, array sizes which are too large to fit into the cache entirely, and rectangle tile shapes.

Various algorithms use different cost models $\mathbf{cost}(h \times w)$ to quantify the effects of cross-conflicts and cache utilization. As observed by researchers previously [22, 8, 27, 32, 19], the cost model can be interpreted as the "tension" between tile shape and tile area. In general, extremely tall tiles are argued to have low cache utilization, and extremely wide tiles are argued to introduce severe TLB misses. Square tiles are favored since they minimize cross-conflicts. However, square tiles may encounter low cache utilization in some cases.

The computation of $\mathcal{M}(C, l, n)$ has been described in [8, 32, 6]. Algorithm **tli** [6] is *precise*, i.e., the tiles generated are guaranteed to be maximal, for arbitrary l. Algorithm **euc** [32] is precise only when $l = 1$. Algorithm **tss** [8] is the first to propose using Euclidean GCD algorithm [13] to compute $\mathcal{M}(C, l, n)$. However, it is imprecise even when $l = 1$.

The essence of computation by **euc** for $\mathcal{M}(C, l, n)$ can be described as $\mathcal{E}(C, n) \equiv \{h_i \times \min(w_i, n) : i \geq 1\}$

Notation	Interpretation
n	size for a two-dimensional array $n \times n$
C	cache capacity
l	cache line size
a	cache associativity
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
$\mathcal{M}(C, l, n)$	the set of maximal tiles that have no self-conflicts
$\mathcal{E}(C, n)$	the set of Euclidean tile sizes
Δ	pad size

Table 1: List of notations used in the paper. All values are defined in terms of array elements.

$\mathcal{M}(C = 2048, l = 4, n = 127)$	$\mathcal{E}(C = 2048, n = 127)$	**euc**
{ 127 × 12 ,	{ 127 × 16 ,	{ 124 × 16 ,
113 × 16 ,	16 × 113 ,	13 × 113 ,
16 × 124 ,	15 × 127 ,	12 × 127 }
1 × 127 }	1 × 127 }	

Table 2: An example illustrating various approximations of $\mathcal{M}(C, l, n)$, the set of maximal tiles that have no self-conflicts. The reported sets are for $C = 2048$, $l = 4$, and $n = 127$.

where

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

This computation is precise when the array size is a multiple of cache line size, i.e., $n \equiv 0 \pmod{l}$. **euc** reduces height to be $h - l + 1$, slightly under-utilizing the cache, to take longer cache line into account [32]. Table 2 gives an example illustrating various approximations of $\mathcal{M}(C, l, n)$. The complete specification of **euc** is

$$\mathbf{h}' \times \mathbf{w} \leftarrow \arg\min\{\frac{1}{h'} + \frac{1}{w} : \ h' = h - l + 1, h \times w \in \mathcal{E}(C, n)\} \qquad (\mathbf{euc})$$

euc's cost model $\frac{1}{h} + \frac{1}{w}$ favors square tiles. However, sometimes unfavorable sizes n constrain $\mathcal{E}(C, n)$ to only contain tiles of undesired shapes. For example, in Table 2, there is no tile whose shape is near square, and therefore **euc** is constrained to select 124×16 as its best choice. Since $\mathcal{E}(C, n)$ is sensitive to 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. As shown in Table 2, **eucPad** is able to find 61×31 as its best choice with pad size $\Delta = 5$. By substituting different approximations for $\mathcal{M}(C, l, n)$, we can evaluate the impact of the quality (preciseness) of approximations to $\mathcal{M}(C, l, n)$ on the tiling effectiveness. The complete specification of **eucPad** and its variants is as follows:

$$\mathbf{h}' \times \mathbf{w} \leftarrow \arg\min\{\frac{1}{h'} + \frac{1}{w} :\ h' = h - l + 1, h \times w \in \mathcal{E}(C, n + \Delta), 0 \leq \Delta \leq 8\} \qquad (\mathbf{eucPad})$$

$$\mathbf{h} \times \mathbf{w} \leftarrow \arg\min\{\frac{1}{h} + \frac{1}{w} :\ h \times w \in \mathcal{E}(C, n + \Delta), 0 \leq \Delta \leq 8\} \qquad (\mathbf{maxPad})$$

$$\mathbf{h} \times \mathbf{w} \leftarrow \arg\min\{\frac{1}{h} + \frac{1}{w} :\ h \times w \in \mathcal{M}(C, l, n + \Delta), 0 \leq \Delta \leq 8\} \qquad (\mathbf{tliPad})$$

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-conflicts. This algorithm considers different cache line sizes.

3 A New Algorithm

Both **eucPad** and **datPad** are based on the assumption that the *best* tile is among those which eliminate self-conflicts, minimize cross-conflicts, and maximize cache utilization. We propose a new algorithm arguing that

- The best tile is among those which have *few* conflict misses and *good* cache utilization.

Also, **eucPad** and **datPad** do not take TLB misses into account. We argue that, as other researchers have done before [8, 26], TLB misses are an important performance factor that needs to be considered since a TLB miss costs more than a cache miss and may cause cache stall.

- The best tile eliminates TLB misses.

Finally, we argue that the fixed-amount pad choices strategy proposed by **eucPad** can select "bad" tiles in some cases.

- The padding choices shouldn't be fixed a priori.

Having the above desired properties in mind, we propose a new tile selection algorithm **newPad**. The new algorithm is greedy in that it increases pad size until a *good* tile size can be found with the current pad size. If there are more than one good tile sizes with a pad choice, the algorithm will select the one that minimizes the cost model. The skeleton of **newPad** is shown in Figure 1.

A good tile size, as discussed above, is the one which eliminates TLB misses, have few cache misses and good cache utilization. In this paper, we provide an implementation of what a good tile size might look like, as shown in Figure 2. To guarantee no TLB misses within a tile, we allow only a part of TLB entries to be touched by the entire tile. To

$$
\begin{array}{l}
\Delta \leftarrow 0 \\
\textsf{repeat} \\
\quad \mathbf{h} \times \mathbf{w} \leftarrow \arg\min\{\mathbf{cost}(h \times w) : \mathbf{good}(h \times w),\ h \times w \in \mathcal{E}(C, n + \Delta)\} \\
\quad \Delta \leftarrow \Delta + 1 \\
\textsf{until}\ \ \mathbf{h} \times \mathbf{w} \text{ exists}
\end{array}
$$

Figure 1: The Skeleton of new tiling algorithm **newPad**

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

Figure 2: The implementation of **newPad** ($\alpha = \beta = 0.75$)

---

guarantee good cache utilization, we only consider those tiles whose areas are sufficiently large. Finally, to guarantee few cache misses, the restriction on the tile shape is imposed. Specifically, we do not want the tile shape to be "too far away" from the "optimal" shape l (explained below).

For linear algebra codes, the memory access pattern within a tile are usually due to several array references. Following the strategies used by [22, 8] to estimate the effect of cross-conflicts within a tile in terms of footprint overlap, we get **euc**'s cost model $\frac{h+w}{h*w}$. Tile $h \times w$ interferes with two regions of sizes $h \times 1$ and $1 \times w$. As a result, the square tiles are favored over rectangle ones with the same area [32]. Unfortunately, the effect of large cache line sizes ($l > 1$) is not considered in this model. To capture cache line effects more precisely, we introduce the new model $\frac{h/l+w}{(h/l)w} = \frac{l}{h} + \frac{1}{w}$, which takes cache line size into account. A result of using the new cost model is that the optimal tile shape, instead of square, becomes $\frac{h}{w} = l$, a *rather long* tile size. Recent study has observed this phenomenon already [19]. Since tile shape implicitly estimates the effect of cross-conflicts, we propose to restrict the shape of the selected tile not too far away from the optimal shape in the hope that the tile introduces limited cross-conflicts. Since **newPad** selects tiles that are most likely tall, negative performance effects due to TLB constraints can be avoided.

We do not claim that our formulation of $\mathbf{good}(h \times w)$ is the *optimal* choice. However, we believe that the identified performance factors (TLB misses, tile area, and tile shape) are critical, and our modeling of these factors are at a sufficient level of accuracy. In addition, it

Algorithm	Maximal	Pad choices
org	No tiling	
euc[32]	almost	0
eucPad[32]	almost	0-8
tliPad[6]	guaranteed	0-8
maxPad	almost	0-8
newPad	almost	unlimited
datPad[28]	guaranteed	unlimited

Table 3: Tiling Heuristics used for our experimental study

euc	**eucPad**	**tliPad**	**maxPad**	**newPad**	**datPad**
124 × 16 (0)	61 × 31 (5)	32 × 63 (3)	32 × 63 (3)	98 × 16 (3)	44 × 44 (55)

Table 4: The final selections of various tile selection algorithms for the example in Table 2. The values in parentheses are the final pad sizes.

is interesting to note that **datPad** can be considered as a specialized case of our new algorithm by providing more strict definitions of good tile area and shape. Experimental results show that less strict definitions achieve comparable magnitude and stability of performance improvements with *significantly smaller* pad sizes. Finally, we want to mention that for any given tile size $h \times w$, there always exists a pad size that make it having no self-conflicts, for example, $\Delta = iC + h$ [21]. As a result, **newPad** will terminate for all cases.

Table 3 summarizes the set of algorithms tested in our experimental study. The final tile selections computed by these algorithms for the example in Table 2 are listed in Table 4. For **newPad**, we assume $P = 1024$. For **datPad**, we assume that tile 44×44 is desired.

4 Experimental Evaluation

To compare different heuristics, we varied the array size n from 100 to 1100 double-precision data elements with a step size of 4 elements. For each heuristic, two linear algebra benchmark kernels [1], namely matrix multiplication (`mm`) and LU-factorization without pivoting (`lu`), are executed on three target architectures with different cache organizations[2] The dif-

[1]These benchmarks have been used by many published papers [36, 22, 8, 32, 28] to evaluate the effectiveness of their tile selection algorithms.

[2]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 at the first-level cache, provided that the cache next level is much larger [33]. Therefore, we target the first-level cache.

ferent machine configurations are shown in Table 6. 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. All experimental results are represented in terms of MFLOPS.

We are concerned with the performance of each tiling heuristic in terms of execution time improvements and improvement stability. Figure 3 shows the achieved MFLOPS rate (y-axes) for each of the tiling heuristics and target machines for our 250 input array sizes ranging from 100 to 1100 by a step of 4 (x-axes). Table 7 summarizes the results by reporting average performance over all problem sizes. The following observations can be made:

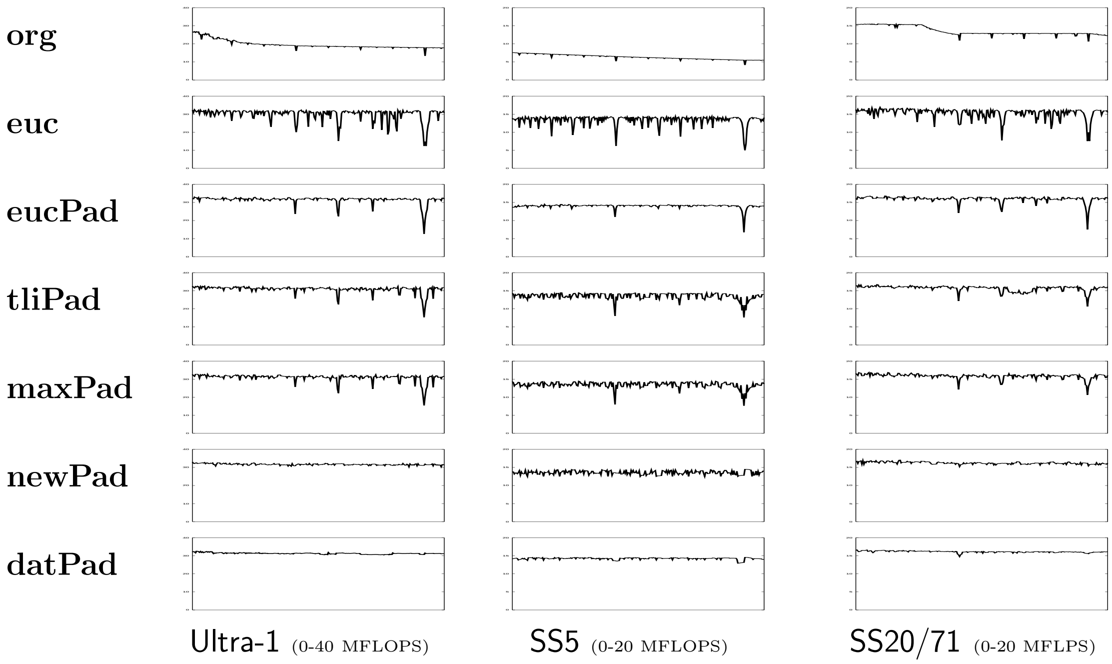
- **euc** has a significant amount of pathological cases which degrade performance improvement (note the downward peaks in **euc**).
- Array padding can effectively stablize the performance improvement (for instance, compare **eucPad** against **euc**).
- **maxPad** and **tliPad** have similar performance in both execution time speedup and stability.
- **eucPad** has similar performance improvement but slightly better stability, compared with **maxPad** and **tliPad**.
- **newPad** and **datPad** have similar speedup and stability. Both algorithms have a significantly better stability than **eucPad**, **tliPad**, and**maxPad**.

In summary, **newPad** has a stability and performance superior to padding algorithms **eucPad**, **tliPad**, and **maxPad**, and similar to **datPad**. The speedup of tiles generated by **newPad** over tiles generated by **eucPad**, **tliPad**, and **maxPad** are in the range of -12% to 173%. With respect to **datPad**, this speedup is in the range of -12% to 6%. However, as will be discussed below, **datPad** has a significantly higher overhead in terms of data space (7×) and execution time (760×). The range of performance improvements of **newPad** in terms of speedups over the other algorithms is presented in Table 5. In the following we will address several issues related to the new algorithm, its experimental results, and some possible extensions.

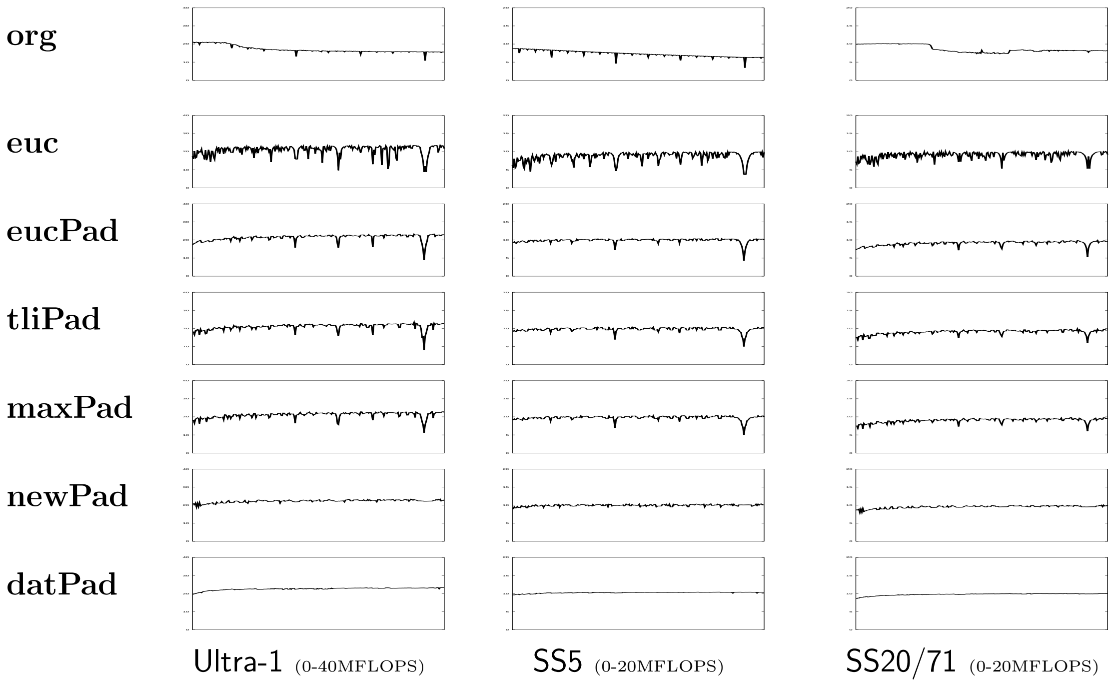
4.1 Cost-Benefit Trade-Offs

We have observed that **newPad** and **datPad** have similar effectiveness in stabilizing performance improvement due to loop tiling, at the cost of padding dummy array elements. 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 8. Compared to **eucPad**, the deviation by **newPad** is a slightly higher. It is the price to be paid for "unlimited" padding.

Another dimension of the costs of a tile selection algorithm is its execution time. Table 9 shows the execution times of the different algorithms in microseconds. The reported numbers are based on our C implementation of the algorithms, and were obtained by measuring the



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 3: Experimental Results

	Ultra-1	SS5	SS20/71
matrix multiplication (`mm`)			
org	20 to 137	72 to 250	2 to 45
euc	-4 to 155	-11 to 188	-3 to 115
eucPad	-6 to 146	-12 to 89	-4 to 115
tliPad	-4 to 102	-11 to 169	-3 to 52
maxPad	-4 to 102	-9 to 69	-3 to 52
datPad	-3 to 5	-12 to 2	-4 to 6
LU-factorization (`lu`)			
org	-10 to 105	1 to 192	-20 to 33
euc	-6 to 152	0 to 177	-10 to 80
eucPad	-3 to 150	-5 to 135	-2 to 79
tliPad	-2 to 173	-4 to 101	0 to 58
maxPad	-4 to 97	-4 to 101	-1 to 57
datPad	-7 to 6	-8 to 3	-11 to 2

Table 5: Range of relative performance improvements of **newPad** in terms of % of *speedup* over other algorithms.

---

average execution times (gcc -O, 143MHz UltraSparc-1) over all 250 input array sizes used in our experiments.

eucPad, **maxPad** and **newPad** are based on the Euclidean GCD algorithm and are therefore fast to compute. **datPad** performs memory-based simulation to find the smallest pad size that ensures no self-conflicts in the selected tile. **tliPad** uses a clever "simulation" strategy by searching the tiles as a computation-based incremental process. Our data shows that on average **eucPad**, **maxPad** and **newPad** all take approx. 80 microseconds, i.e., have comparable performance. **tliPad** takes about 3.68 seconds and **datPad** about 0.057 seconds, making these algorithms 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 algorithms does not know a priori the final tile and pad size, and and enumerates all possible choices to determine the best one. In contrast, **datPad** has a predetermined tile shape and can thus can quickly compute the largest tile size with this shape and reject inappropriate pad sizes.

The preliminary experimental data suggests 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.

Machine	CPU		Data Cache			TLB	
	Type	MHz	C	l	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 6: Different target architectures

Performance Factors for Tiling

Set associativity is not considered as a parameter in **eucPad** and its variants. On the other hand, **datPad** includes it to adjust the effective cache size (i.e., use $\frac{a-1}{a}C$ instead). The experiments showed that set associativity does not have significant impact on the effectiveness of tiling algorithms.

Tile shape is considered by most algorithms, either implicitly through the cost model or explicitly through candidate tiles. In general, tile shape is program-dependent. In the experiments, **euc** (and also **eucPad**) generates the same tile size (and thus the same tile shape) for both `mm` and `lu` for a particular array size. In the contrary, **datPad** argues that the optimal shape for them are $\frac{h}{w} = 1$ and $\frac{h}{w} = l$ respectively. However, the experimental results seem to indicate that the "optimum" of tile shape is not that critical for program performance, at least for the two measured benchmarks.

It *does not* imply the tile shape is not important at all. Previous studies have suggested that square-like tile shape is favored [32]. For example, Esseghir proposed a tile selection algorithm [9] which chooses the "overly skinny" tile shape but is observed to have poor performance improvement [8, 32, 19, 28]. On the other hand, "overly fat" tile shape introduces TLB thrashing effect and thus degrades the performance [8, 26, 19]. **newPad** skews the sqaure-like tile shape to take into account longer cache line size. It also restricts the shape of the selected tile not to be "overly skinny" nor "overly fat".

Tile area is another performance factor implicitly taken into account by many algorithms. It is relevant to the loop overhead. **eucPad** and its variants consider the tiles in $\mathcal{E}(C, n)$ having good tile area. The selected tile is not necessarily the *largest* in its area among all tiles which have no self-conflicts. And experimental results show that it works fine as well.

TLB thrashing is an important performance factor to be considered [8, 26], which are not taken into account by both **eucPad** and **datPad**. Mitchell et al. proposed a tile selection algorithm which finds tiles that have no TLB misses and few cache misses, and minimize the (program-dependent) cost model [26]. Their work focus on using effective cache size [22, 37] to reduce the impact of cache misses on low-associativity caches. In contrast, **newPad** works directly on low-associativity caches.

Finally, we want to point out that **euc**'s cost model $1/h + 1/w$ may occasionally choose a square-like tile with very low cache utilization. Consider two tiles, $\frac{C}{2} \times 2$ and 4×4. **euc** will favor 4×4 which in turns has very low cache utilization.

	Ultra-1	SS5	SS20/71
matrix multiplication (`mm`)			
org	19.59 (2.23)	6.35 (0.65)	13.58(1.17)
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)
maxPad	30.96 (1.99)	13.45 (0.89)	15.84 (0.70)
newPad	31.69 (0.43)	13.58 (0.52)	16.24 (0.30)
datPad	31.30 (0.40)	14.24 (0.28)	16.12 (0.23)
LU-factorization (`lu`)			
org	17.02 (1.86)	7.29 (0.83)	8.62 (0.96)
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)
maxPad	21.08 (1.58)	9.78 (0.57)	9.00 (0.58)
newPad	22.26 (0.76)	9.97 (0.25)	9.54 (0.37)
datPad	22.61 (0.67)	10.26 (0.17)	9.79 (0.28)

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

5 Conclusions and Future Work

In the paper 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 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

	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)
maxPad	3.98 (2.75)	3.96 (2.98)
newPad	4.96 (8.43)	3.30 (7.21)
datPad	66.58 (46.68)	19.81 (16.54)
LU-factorization (`lu`)		
datPad	51.76(35.40)	19.43(15.89)
all others	same as for `mm`	

Table 8: Analysis of space overhead due to padding. All values are represented in terms of final pad sizes in the leading array dimension. The values in the parenthesis are standard deviations.

eucPad	80
tliPad	3.68×10^6
maxPad	80
newPad	80
datPad	57×10^3

Table 9: Computational costs of tiling algorithms in *microseconds* averaged over all 250 input array sizes used in the experiments.

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 [18].

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. 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: A compiler framework for analyzing and tuning memory behavior. *To appear in ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[12] 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.

[13] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, Reading, Massachusetts, Second edition, 1994.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *1999 ACM International Conference on Supercomputing*, 1999.

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

[22] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, Calif., April 1991.

[23] 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.

[24] 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.

[25] 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.

[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, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[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.