Tile Selection Algorithms and Their Performance Models

Chung-Hsing Hsu and Ulrich Kremer*

Department of Computer Science
Rutgers University

DCS-TR-401

October 1999

Abstract

Loop tiling is an effective optimizing transformation to reduce the memory access cost of a program, especially for dense matrix computations. However, the success of loop tiling is heavily dependent on the appropriate selection of tile shapes and sizes. In this paper we examine several existing tile selection algorithms in a unified framework, and quantify their performance improvements for three dense matrix computation kernels and three target architectures. In addition, a new tiling algorithm is discussed that was inspired by the observed behavior of previous algorithms.

Four different quality metrics are introduced to measure the performance improvements of the algorithms over untiled versions of the three program kernels. The experiments showed that tile selection algorithms can be either very similar in performance, or significantly different depending on the chosen performance metric. For the measured test cases, our new selection algorithm had a better overall performance across the different performance metrics.

1 Introduction

As the speed of modern microprocessors has increased much faster than the memory speed, the traffic between the memory system and processor has become the key bottleneck for program performance. As a result, effectively keeping reused data in the cache is vital in achieving good program performance. Loop tiling is an optimization technique that partitions a loop's iteration space into uniform *tiles* or *blocks*. The goal of this optimization is to keep array elements with temporal locality (reuse) in cache by grouping array references into tiles. The tiles are then executed in sequence, allowing reuse across tile traversals.

*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

Several different tile selection algorithms for dense matrix codes have been proposed in the literature. Nearly all tile selection algorithms take as input a loop nest with a given loop order and a set of tiled loop iterations. The task of the tile selection algorithm is to determine the tile shapes in terms of width and height that have the best performance across all valid problem and array sizes. Published tile selection algorithms typically report experimental numbers for all or a subset of the following three dense matrix computation kernels: matrix multiply, LU factorization without pivoting, and successive over-relaxation (SOR). All three programs exhibit significant data reuse that can be exploited by loop tiling.

Figure 1 shows a possible input to a tile selection algorithm for matrix multiply. In this example, a block $w \times h$ of array `a` is reused across iterations of the `j`-loop. Careful selection of the block will realize the data reuse and thus improve the program performance.

```
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)
```

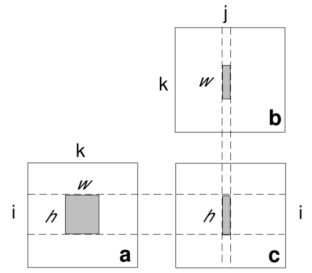


Figure 1: Tiled matrix multiply: What is the best choice of w and h to exploit data reuse?

The quality of a chosen tile width and height, i.e., the tile shape may depend on the resulting number of capacity and conflict misses across the memory hierarchy of the target machine, misses in the TLB, the ratio between exploited data reuse and loop overhead due to tiling, and other target system effects. It is important to note that most of these factors are problem size and array size dependent. Therefore, finding an efficient tile shape is a hard problem.

Due to the observation that in low associativity caches, *conflict misses* are a major source of cache misses in a loop nest [20], many cost models of tile selection algorithms are trying to quantify the cache conflicts, i.e., use the expected number of cache conflicts as their main criterion to evaluate the quality of a tile shape. Different hardware and software solutions are also proposed to avoid severe cache conflicts, in particular victim cache [16, 28], skewed cache [2], array padding [27, 25], array copying [19, 30], or combinations of these techniques.

In this paper, a subset of published tile selection algorithms are used to illustrate different quality aspects of tiling algorithms and their underlying performance models. Such quality metrics include average performance improvements and stability of performance improvements. The comparison is based on the measurement of the actual execution times of the selected algorithms for three two-dimensional dense matrix codes on three target architectures with different cache organizations. Problem and array sizes were chosen across the range of 100×100 to 1100×1100 data elements. For the experiments, the tile selection algorithms were applied by hand.

The presented quantitative comparison does not claim to be complete, neither in terms of the selected algorithms nor in the choice of possible quality metrics. The paper provides a framework to compare the performance of different tiling algorithms. Different quality metrics introduce different notions of precision and effectiveness, allowing a better understanding of the overall performance of a tiling algorithm and its associated performance prediction model. Such information is crucial for any compiler writer who wants to implement an optimization transformation such as loop tiling in her/his compiler. In addition, the insights into the performance characteristics of previous algorithms can be used to improve existing tiling algorithms. A new tile selection algorithms is discussed that was inspired by the observed behavior of previous algorithms. In addition, deficiencies in existing tiling algorithms have been discovered that should be addressed in future tiling algorithms. In summary, the main contribution of this paper are

- A quantitative comparison of several published tile selection algorithms. The comparison showed that depending on the selected quality metrics, the performance of the algorithms can be nearly equivalent or significantly different.
- A set of interesting quality metrics.
- A new tile selection algorithm. The new algorithms has a better overall performance for most performance metrics than previous algorithms, in particular the "standard" metric of average performance improvements.
- The identification of possible areas of further improvements to tiling algorithms.

The paper is organized as follows. Section 2 presents a comparison of published tile selection algorithms. Our quality metrics used to evaluate the performance of three representative tiling algorithms are discussed in Section 3. Experimental results are discussed in Section 4, including the details of our new algorithm and a summary of our findings. A discussion of conclusions and future work is presented in the last section of the paper.

2 Tile Selection Algorithms

The performance of a specific tile shape (width and height) depends on many factors, including the sizes of arrays referenced in the loop. Array sizes have a significant impact on the performance of a tile shape due to possible cache capacity and conflict misses. In order to deal with this complexity, tile selection algorithms use a "filter" approach to first select a set of candidate block shapes that are promising choices, and then select the final shape from this candidate set. For simplicity, only square arrays of size $n \times n$ and the set of two-dimensional *rectangle* blocks, $\mathcal{B} = \{h \times w |\ 1 \leq h, w \leq n\}$, are considered. In addition, we assume that an array column can completely fit into the cache.

In the following, we refer to the cache size as C, the cache line size as l, and the column size of an array as n. All values are defined in terms of array elements. The set of all *non-conflicting* blocks for cache line size l is denoted $\mathcal{N}(C, l)$. A block is non-conflicting if it generates no self conflicts [26]. These notations are summarized in Table 1.

Notation	Interpretation
n	size for a two-dimensional array $n \times n$
C	cache capacity
l	cache line size
$\mathcal{B}$	the set of all blocks
$\mathcal{N}(C, l)$	set of all non-conflicting blocks

Table 1: The summary of notations used in defining the candidate set of block shapes. All values are defined in terms of array elements.

Algorithm	**qualified**($\mathcal{B}$)	**cost**($h \times w$)
lrw [19]	$\{h \times w \mid h = w, h \times w \in \mathcal{N}(C, 1)\}$	$1/h + 1/w + (2h + w)/C$
ess [6]	$\{h \times w \mid h = n, h \times w \in \mathcal{N}(C, 1)\}$	$C/(h * w)$
tss [5]	$\{(\lfloor h/l \rfloor l) \times w \mid h * w + h + w \leq C, h \times w \in \mathcal{N}(C, 1)\}$	$(2h + w)/(h * w)$
euc [26]	$\{(h - l + 1) \times w \mid h \times w \in \mathcal{N}(C, 1)\}$	$1/h + 1/w$
moon [23]	$\{h \times w \mid h \times w \in \mathcal{N}(C, l)\}$	$1/h + 1/w + (h + w)/C$
tli [3]	$\{h \times w \mid h \times w \in \mathcal{N}(C, l)\}$	$1/h + 1/w + (h + w)/C + h * w/C^2$
wmc [32]	$\{h \times w \mid h = w, h * w \leq \alpha C, h \times w \in \mathcal{B}\}$	$C/(h * w)$
mhcf [22]	$\{h \times w \mid hw(1/h + 1/l + 1/n) \leq \alpha C, h \times w \in \mathcal{B}\}$	$(1/h + 1/w)(1/n + 1/l) + 2/(h * w)$

Figure 2: Different tile selection algorithms. Underlined algorithms are used for our quantitative comparison.

A general tile selection algorithm chooses from the set of *qualified* candidate blocks the one which minimizes a particular cost model, i.e.,

$$\arg\min\{\mathbf{cost}(h \times w) \mid \mathbf{qualified}(\mathcal{B})\}$$

where **qualified**($\mathcal{B}$) is a subset of $\mathcal{B}$ in which all blocks satisfy certain qualifications such as "fit-in" constraints [22] or shape constraints. Based on the observation that *capacity misses* and *conflict misses* are two major sources of cache misses in a loop nest for low associativity caches [20], most algorithms focus on blocks which generate very few self conflicts, and their cost models try to quantify either capacity misses, cross conflict misses, or both. For those candidate blocks, some algorithms consider only non-conflicting blocks, and some use *effective* cache size [19, 32], αC, to reduce the impact of self conflicts. Figure 2 shows a number of tile selection algorithms for matrix multiply.

Note that algorithm **mhcf** originally uses a multi-level qualifications and cost functions, in particular the combined effect of cache and TLB accesses. For simplicity, we present their simple one-level qualification and cost function here.

As shown in Figure 2, all algorithms differ in the set of qualified blocks they consider, the shape they constrain, or the cost models they associate with. Algorithms **lrw**, **ess**, **tss**, **euc**,

moon, and **tli** all focus on non-conflicting blocks but in different precision with respect to arbitrary l. **moon** and **tli** use the most precise qualified blocks $\mathcal{N}(C, l)$ while all the others "search" an approximation $\mathcal{N}(C, 1)$, i.e., assume that $l = 1$ independent of its actual value on a given target machine. **tss** [1] and **euc** even adjust blocks a little to better approximate the set. Both **lrw** and **wmc** constrain the block shape to be square while all the others allow rectangle blocks for better cache utilization. Algorithms **lrw**, **moon**, and **tli** quantify the total impact of both capacity misses and conflict misses. **tss** and **euc** try to model cross conflict misses, while **mhcf** considers capacity misses. The cache utilization is used as the cost model by **ess** and **wmc**.

lrw is recognized to have good average performance improvement for a range of array sizes, but it selects very small blocks for pathological array sizes. This low cache utilization severely degrades the effectiveness of tiling. For example, when $n = 512$, **lrw** will select a 4×4 block for $C = 2048, l = 1$. Many later algorithms can therefore incorporate modification to avoid the occurrences of pathological array sizes. Algorithms **ess**, **tss** and **euc** avoid the problem by allowing rectangle blocks. For example, under the same configuration, **euc** is able to select a 512×4 block instead. However, simply allowing rectangle blocks will not be enough. For example, for $n = 516$, **euc** will select a 16×127 block. The overly "fat" block introduces TLB misses and degrades the performance improvement in near-pathological array sizes. Algorithm **ess** guarantees not to select overly "fat" blocks, but its preference on overly "skinny" blocks results in low cache utilization. For example, for $n = 516$, **ess** selects a 516×3 block, which has only 76% cache utilization. The mixed nature of shape and cache utilization on the effectiveness of tiling is explicitly discussed in [23].

Other Related Work

Kodukula and Pingali in [17] proposed a new program transformation that aims at the partition and traversal of the data space. Their algorithm determines square block sizes that maximize the cache utilization without causing any capacity misses [2] [18]. PHiPAC [1] used the profile-driven approach to determine the block size. Specifically, the performance of matrix multiply with various block sizes is profiled and the best one is selected. ATLAS [31] has a dispatcher which determines the best block size case by case.

Significant previous work has been done to quantify the footprint of a loop nest [7, 24, 4], which can be used to specify the "fit-in" constraints and estimate the number of capacity misses. Recent work tries to quantify the number of conflict misses as well [29, 12, 13, 11, 8, 9].

3 Quality Metrics

Quality metrics are needed to summarize the overall performance of a tiling algorithm. In this section, we will explore several possible definitions of the quality of a tile selection algorithm based on a set of measured data points that are summarized in a single numerical value. In

[1] In the original paper [5], the adjustment is not to better approximate the result but to align at cache line boundary for better performance. However, it has the side effect of better approximation.

[2] That is, the total footprint of a partition of the data space does not exceed the capacity of the effective L2-cache.

Notation	Interpretation
n	an instance of the samples
N	total number of instances
A	a tile selection algorithm
$t_n(A)$	the execution time of tiled program by algorithm A for array size n
T_n	the execution time of untiled program for array size n
t_n^*	the execution time of the best possible choice for array size n
$[p]$	equal to one if the predicate p is evaluate to be true, and zero otherwise.

Table 2: Notations used in describing tile selection algorithms.

order to do so, some notation has to be introduced. Table 2 lists the used abbreviations. We denote the execution time of the selected tile size for array size n and algorithm A as $t_n(A)$. Each data point n is called an *instance*. For this case study, we use 251 different array sizes, i.e., we looked at 251 different instances. Let t_n^* denote the execution time of the best possible choice in the candidate set of Euclidean and maximal square tile sizes for array size n. The execution time of the untiled version of the code for instance n is denoted by T_n, and N is the total number of different array sizes or instances ($N = 251$). Notation $[p]$, borrowed from [10], is defined to be equal to one if the predicate p evaluates to true, and zero otherwise.

The following are several quality metrics used in this paper. The first two can be considered "standard" in summarizing data. It is important to note that these metrics are only representative of queries that a compiler writer may want to ask about the performance of an optimization algorithm and its underlying performance model. The first metric captures the expected average performance improvement of an optimization. The second metric describes the stability of an algorithm in terms of the performance deviation from its average performance improvement, while the third one only considers the deviation with respect to a performance decrease relative to the average performance improvement. The fourth metric tries to unify the first and second metrics in a single value. The notation of optimality here is relative to a particular algorithm, in our case, **min**, and not relative to the performance of all possible tile sizes. The hypothetical algorithm **min** is the result of taking the minimum execution time of **lrw**, **ess**, and **euc** for each measured array size. It is an open question whether **min** can be constructed or approximated effectively. An algorithm is desirable if it has a stable and good average performance improvement. In addition, it may be undesirable to use an algorithm that does well in most cases, but generates "outfliers" with severe performance degradation.

"Standard" average performance improvement.

$$Q_1(A) = \frac{1}{N} \sum_n \frac{T_n}{t_n(A)} \tag{1}$$

"Standard" variation of performance improvement.

$$Q_2(A) = \sqrt{\frac{1}{N}\sum_n(\frac{T_n}{t_n(A)} - Q_1(A))^2} \qquad (2)$$

Downward variation of performance improvement.

$$Q_3(A) = \sqrt{\frac{1}{N}\sum_n(\frac{T_n}{t_n(A)} - Q_1(A))^2[\frac{T_n}{t_n(A)} < Q_1(A)]} \qquad (3)$$

Average variation of performance loss compared to the "optimal" choice.

$$Q_4(A) = \sqrt{\frac{1}{N}\sum_n(\frac{t_n(A)}{t_n^*})^2} \qquad (4)$$

4 Experiments

In our experiments, we varied the array size n from 100 to 1100 data elements [3], stepped by 8, on the three benchmark kernels. Each array element is of double precision (8 bytes). The subset of tile selection algorithms for this experiment consists of **lrw**, **ess**, and **euc**. **new** is our new algorithm which will be discussed in Section 4.2. In order to allow a comparison, the numbers for **new** are already reported here.

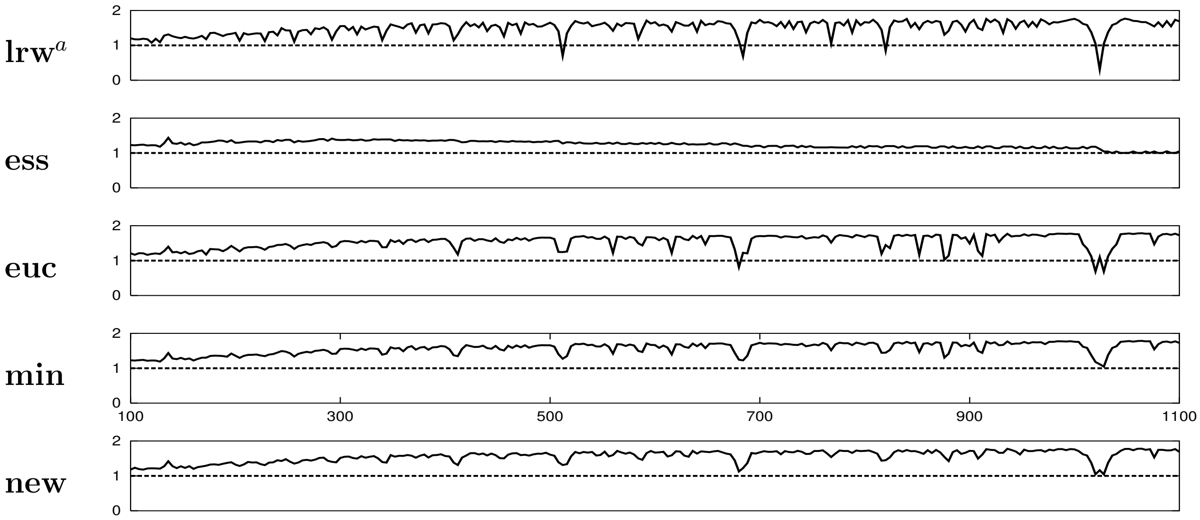
For each tiling algorithm and array size n we determined the selected tile sizes by hand, generated the corresponding Fortran 77 program [4], compiled it using Sun's SC5.0 Fortran 77 compiler with the optimization option switched on (`f77 -O`), and measured the resulting execution times on different workstations. The Fortran programs initialized all arrays before running the kernels. However, the execution time of initialization was excluded in the final measurements. Each program was executed five times in sequence, and the minimum running time was selected to minimize noises and avoid file cache effects. The target architectures used in the experiments are listed below, together with their different cache organizations. All caches are directly mapped. Entries of the form $s/l/X$ give the cache size s in Kbytes and the line size l in bytes. The letters D and U specify the cache as either a data cache or a unified cache, respectively. All experimental results are represented in terms of performance improvement [5] of the tiled version over the untiled version of a benchmark program. The loop orders for the untiled versions were determined by applying loop order optimizations as proposed by McKinley, Carr and Tseng [21].

Machine	Data Cache		TLB
	L1-cache	L2-cache	
UltraSparc-1 (US1)	16KB/32B/D	512KB/64B/U	8KB/64 entries/D
SparcStation-5 (SS5)	8KB/16B/D		4KB/64 entries/U
SparcStation-20 (SS20/HS11)	256KB/64B/U		4KB/64 entries/U

[3] For SS5, since the whole cache can hold up to 1024 elements, we only enumerate array sizes from 100 to 1024 to guarantee that a single array column can completely fit in the cache, as specified in Section 2.

[4] The Fortran program uses column-major array layout.

[5] Also called *speedup* [14].



^{a}$\mathbf{cost}(h \times w) = 1/h + 1/w + (2h + w)/C$

Figure 3: The speedup of tiled matrix multiply over its untiled version for five different tile selection algorithms on US-1. **min** is a hypothetical algorithm which takes the minimum execution time of the three selected algorithms.

4.1 Results

Figures 3 through 11 plot the speedup of our three tile selection algorithms [19, 6, 26] for the three program kernels over their untiled versions for array sizes between 100 and 1100 double precision (8 bytes) elements. The code for the tiled versions of matrix multiply is shown in Figure 1. The code for the tiled versions of LU factorization and SOR can be found in the appendix (Figures 13 and 14, respectively). The experiments were performed on three different target systems. In addition, the values of the four quality metrics are shown for each program kernel and target system.

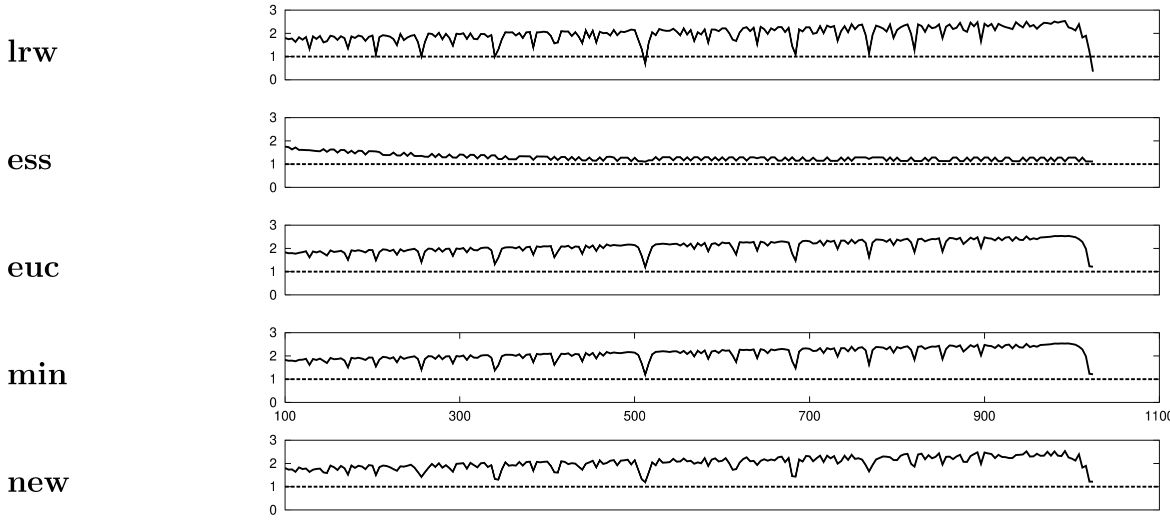


Figure 4: The speedup of tiled matrix multiply over untiled version on SS5.

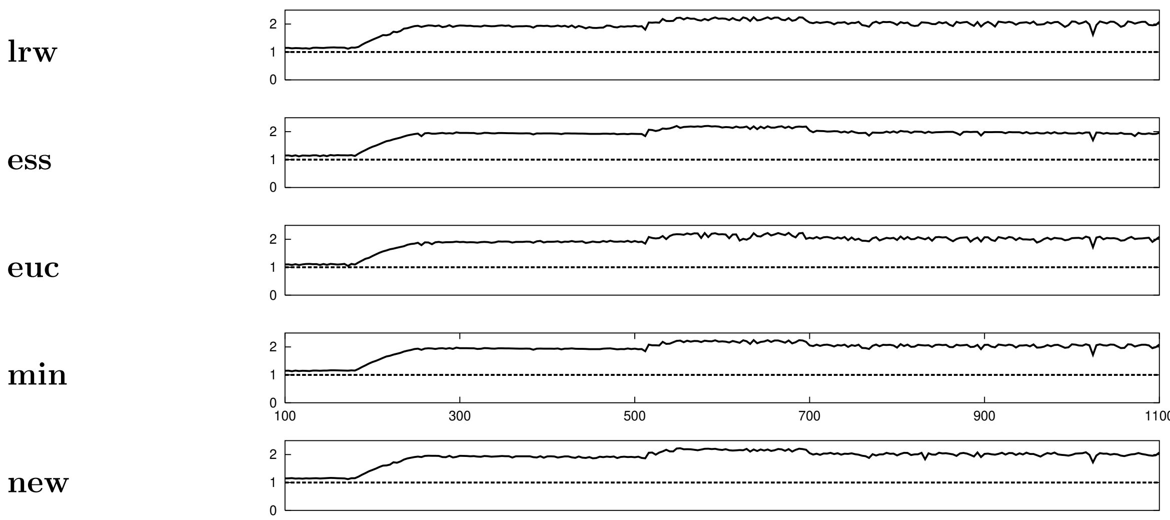
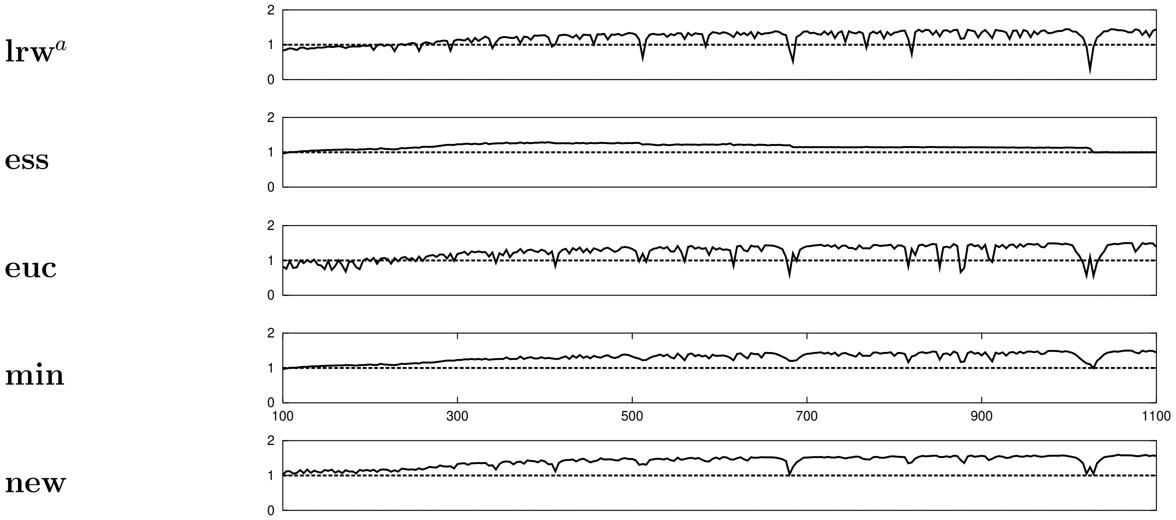


Figure 5: The speedup of tiled matrix multiply over the untiled version on SS20/HS11.



[a] $\mathbf{cost}(h \times w) = 1/h + 1/w + (h + w)/C$

Figure 6: The speedup of tiled LU factorization over untiled version on US-1.

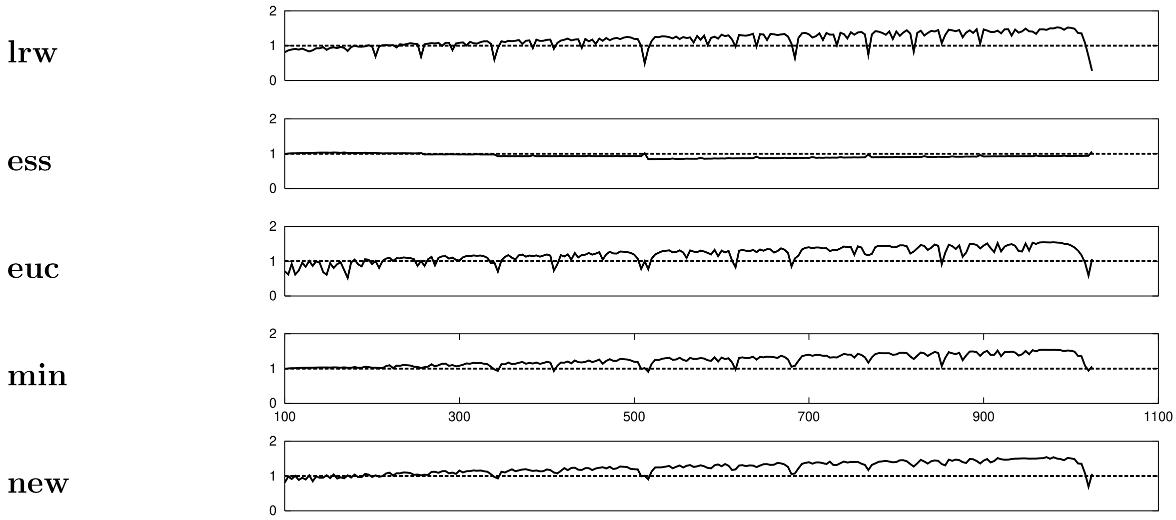


Figure 7: The speedup of tiled LU factorization over untiled version on SS5.

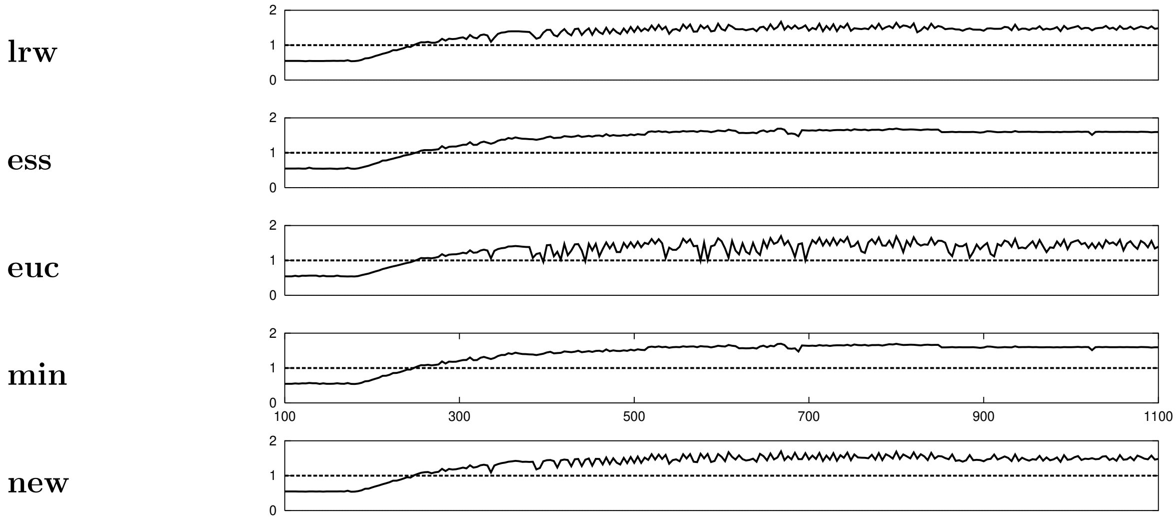
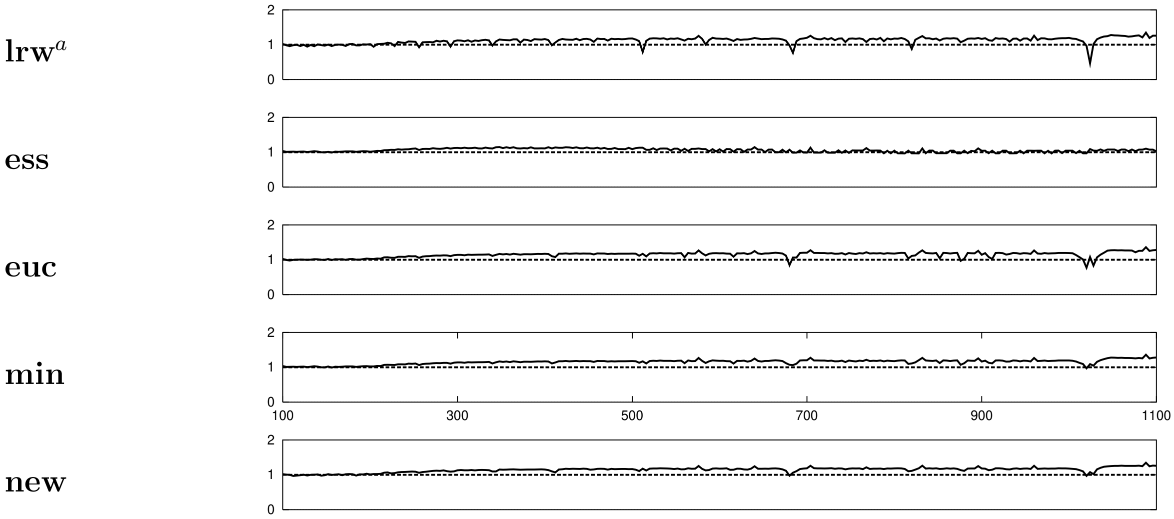


Figure 8: The speedup of tiled LU factorization over untiled version on SS20/HS11.



[a] $\mathbf{cost}(h \times w) = C/(h * w)$

Figure 9: The speedup of tiled SOR over untiled version on US-1.

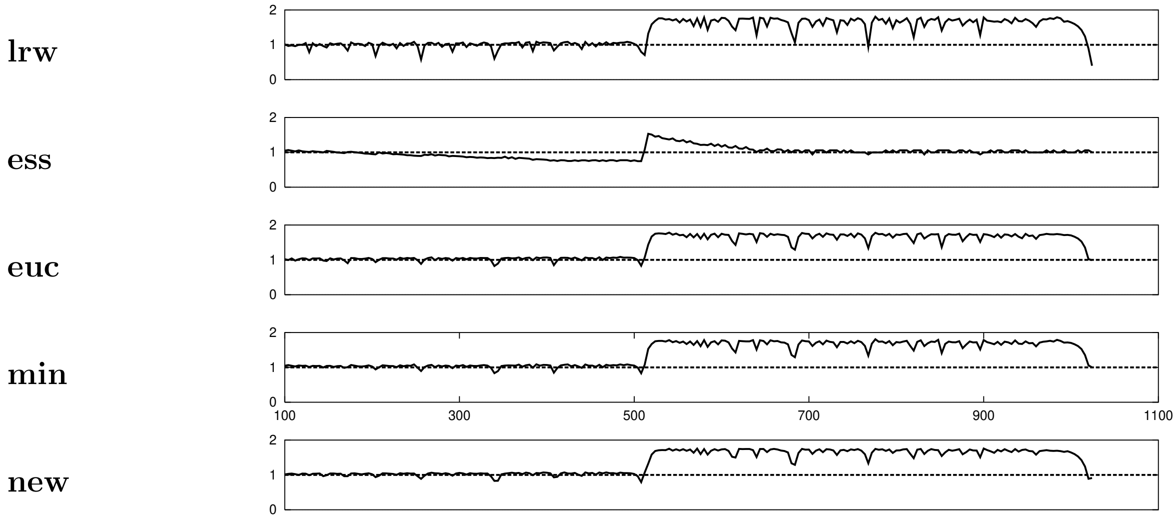


Figure 10: The speedup of tiled SOR over untiled version on SS5.

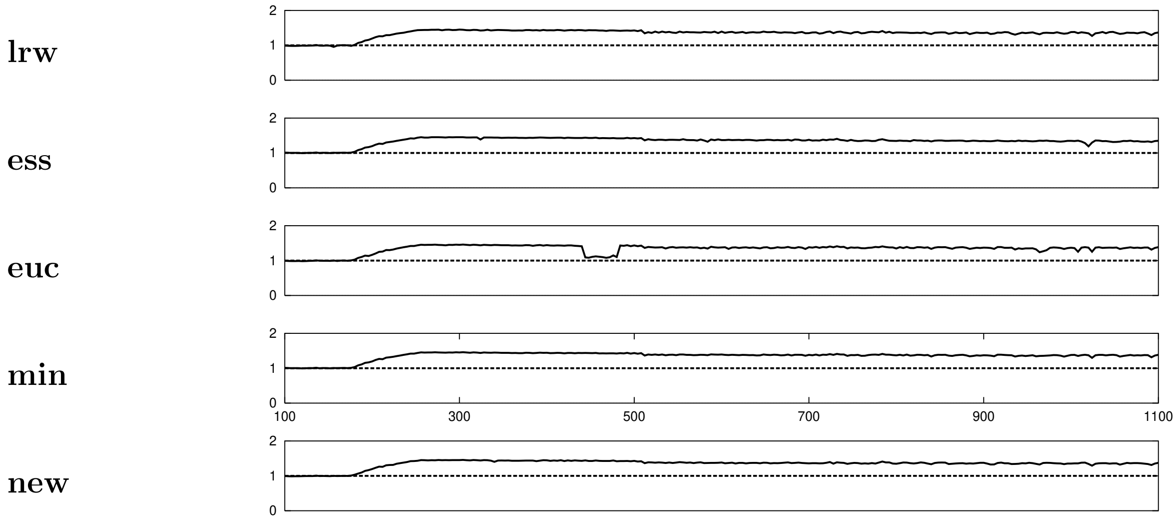


Figure 11: The speedup of tiled SOR over untiled version on SS20/HS11.

Matrix Multiply					
Quality	**lrw**	**ess**	**euc**	**min**	**new**
US-1					
Q_1	1.50	1.24	1.54	1.57	1.56
Q_2	0.21	0.10	0.21	0.16	0.17
Q_3	0.17	0.08	0.17	0.13	0.14
Q_4	1.08	1.29	1.03	1.00	1.01
SS5					
Q_1	1.99	1.29	2.10	2.11	2.03
Q_2	0.33	0.14	0.27	0.27	0.27
Q_3	0.27	0.08	0.21	0.20	0.20
Q_4	1.09	1.69	1.00	1.00	1.04
SS20/HS11					
Q_1	1.92	1.90	1.90	1.94	1.91
Q_2	0.28	0.27	0.29	0.28	0.27
Q_3	0.25	0.24	0.25	0.25	0.24
Q_4	1.01	1.02	1.02	1.00	1.01

LU factorization					
Quality	**lrw**	**ess**	**euc**	**min**	**new**
US1					
Q_1	1.21	1.16	1.24	1.30	1.41
Q_2	0.19	0.08	0.22	0.14	0.16
Q_3	0.16	0.06	0.18	0.10	0.13
Q_4	1.12	1.14	1.08	1.00	0.93
SS5					
Q_1	1.18	0.93	1.19	1.24	1.24
Q_2	0.21	0.05	0.22	0.16	0.18
Q_3	0.16	0.03	0.17	0.12	0.13
Q_4	1.10	1.36	1.07	1.00	1.01
SS20/HS11					
Q_1	1.31	1.40	1.27	1.40	1.33
Q_2	0.31	0.35	0.31	0.35	0.32
Q_3	0.27	0.31	0.26	0.30	0.28
Q_4	1.06	1.00	1.11	1.00	1.05

SOR					
Quality	**lrw**	**ess**	**euc**	**min**	**new**
US1					
Q_1	1.12	1.06	1.15	1.15	1.14
Q_2	0.09	0.05	0.08	0.07	0.07
Q_3	0.08	0.04	0.07	0.05	0.06
Q_4	1.04	1.10	1.01	1.00	1.01
SS5					
Q_1	1.35	1.00	1.38	1.39	1.37
Q_2	0.36	0.15	0.34	0.34	0.33
Q_3	0.26	0.10	0.24	0.24	0.24
Q_4	1.05	1.42	1.01	1.00	1.01
SS20/HS11					
Q_1	1.34	1.34	1.33	1.35	1.35
Q_2	0.12	0.12	0.13	0.12	0.12
Q_3	0.11	0.10	0.11	0.11	0.11
Q_4	1.01	1.01	1.02	1.00	1.01

4.2 New Tiling Algorithm

As mentioned in Section 2, algorithm **lrw** selects square block sizes and has good average performance improvements, but suffers from the pathological array sizes. Algorithms **ess** and **euc** avoid the problem by allowing rectangle block sizes. However, **ess** prefers overly "skinny" blocks and ends up with low cache utilization, while **euc** selects overly "fat" blocks which may introduce severe TLB misses. This observation seems to suggest that blocks of near-square shape and largest cache utilization are more favorable. This proposition has been examined analytically in [23] with respect to their cost model. In this section we will analyze the same proposition based on our experimental data by characterizing the hypothetical algorithm **min** in terms of shape and cache utilization, as shown in Figure 12. A few observations can be made:

- The shape ε is almost always greater than 1. It indicates that overly "fat" blocks are not desirable.
- The size ρ is not always close to 1. It *might* indicate that cache utilization is not the primary performance factor.
- For a number of cases, when ε is close to 1, the cache utilization ρ is pretty low. Again, the squareness of the block seems to have more impact than the cache utilization.

These observations confirm that *moderately tall* blocks are more preferable. It also shows that cache utilization might not be as equally important as the squareness of the blocks. Based on these observations, we propose an "add-on" to the algorithm **euc**: height > width. We term this modified version algorithm **new**. Intuitively, **new** selects among the set of *tall* blocks and thus avoids the overly "fat" problem **euc** suffers. Note that, unlike **euc** which

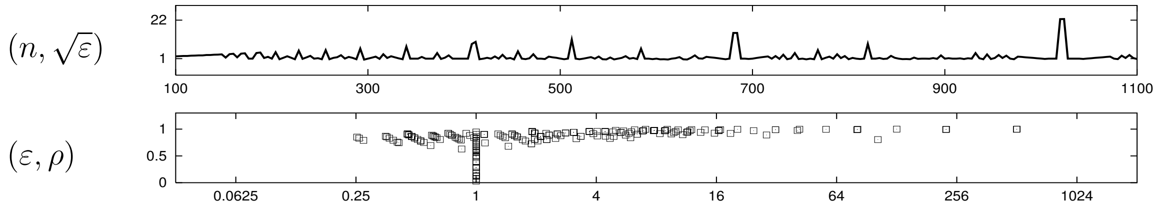


Figure 12: The characterization of **min** for tiled matrix multiply in terms of shape $\varepsilon = h/w$ and cache utilization $\rho = h * w/C$. The upper figure shows $\sqrt{\varepsilon}$ as a function of array size n. The square root has been chosen to make the figure easier to plot. The lower figure shows ρ as a function of ε (logarithmic scale).

adjusts block sizes to take into account arbitrary cache line size ($l \neq 1$) [26], algorithm **new** does not perform any adjustment. As the results show, the under-estimation of conflict misses in **new** is not a significant performance factor.

4.3 Interpretation of Results

The results show that the tile selection algorithms perform differently for different performance metrics. Overall, loop tiling is a successful transformation to speed up the program kernels in our benchmark suite. However, there are cases where loop tiling has no effect, or significantly slows down program execution as compared to the untiled version. Our new algorithm had the best overall performance across all quality metrics. In addition, it performed on average the best for Q_1, the "standard" metric of average performance improvement over the untiled version of a program. Other observations are:

- For Q_1, **new** has the best average performance, followed by **euc**, **lrw**, and **ess**.
- For Q_2, **ess** has the best average performance, followed by **new**, **euc**, and **lrw**. **euc** and **lrw** are very similar in their performance.
- For Q_3, **ess** has the best average performance, followed by **new**, **euc**, and **lrw**.
- For Q_4, **new** has the best average performance, followed by **euc**, **lrw**, and **ess**.
- **min**'s speedup is almost always greater than the speedup of **lrw**, **ess**, and **euc** for matrix multiply and LU-factorization. It indicates that none of the existing three algorithms is a clear winner across the measured array sizes. This observation also suggests that there is still significant room for improvements. Efforts should be made to find an algorithm that approximates or even surpasses **min**'s performance.
- In most cases, **new** shows performance improvements and stability comparable to **min**. In one case, namely, LU factorization on US-1, it even beats **min** for its average performance impact.

- For the target machine with the largest cache size (SS20/HS11), **ess** has comparable performance with the other algorithms. In one case, namely LU factorization, **ess** is actually significantly better than any other algorithm. The reason for this is that the entire data set fits into cache, giving **ess** an advantage due to its low loop overhead.

- Tiling algorithms for LU factorization were harmful for smaller problem sizes in the sense that they increased the execution time relative to the untiled version. New tiling algorithms should include models that can identify the cross-over points. (see Figures 6, 7, and 8). The same holds for SOR and its cross-over points for the different target architectures (see Figures 9, 10, and 11).

5 Conclusions and Future Work

In this paper a framework was presented to qualitatively characterize different tile selection algorithms. The algorithms can be distinguished by their different choices of qualified candidate sets and cost models. To quantify the quality of the tile selection algorithms, four metrics were introduced and applied to the data collected by our experiments with different tiling algorithms. The experiments showed that tiling algorithms can be very similar in performance, or significantly different depending on the choice of metrics. We also identified the possibility of tile shape as the primary performance factor, followed by cache utilization as the secondary factor. A new tile selection algorithm was introduced based on this proposition. The new algorithm was evaluated against three published algorithms. The experiments showed that the new algorithm has a better overall performance and stability across different quality metrics than the three previous tiling algorithms. In fact, the new algorithm can be considered a realization of the "minimal" hypothetical algorithm which is defined by picking the tile shape with the best minimum execution time across all three previous algorithms.

In the future, we are planning to extend our work along several research directions. First, the new algorithm will be evaluated on a wider selection of benchmark programs and underlying architectures, or even new quality metrics, to verify the importance of tile shape. The impact of the TLB and tiling for multiple arrays can also be evaluated using the same methodology. The complexity of cost models and their performance impact is another issue we are planning to study. We believe that more precise and costly performance prediction models may not be justified by the resulting performance improvements since such improvements may also be achieved by much simpler and therefore less expensive models. In addition, we are planning to investigate the impact of other program transformations such as array padding on tiling algorithms and their cost models. Finally, the presented methodology to evaluate existing optimization transformations may be incorporated into a semi-automatic tool, as discussed in [15].

References

[1] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *1997 ACM*

International Conference on Supercomputing, Vienna, Austria, July 1997.

[2] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *22nd International Symposium on Computer Architecture*, pages 265–271, 1995.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, October 1998.

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

[11] J. Harper, D. Kerbyson, and G. Nudd. Efficient analytical modeling of multi-level set-associative caches. In *High Performance and Networking (HPCN) '99*, 1999.

[12] J. S. Harper, D. J. Kerbyson, and G. R. 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.

[13] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical modeling of set-associative cache behaviour. Technical Report CS-RR-349, Department of Computer Science, University of Warwick, Coventry, UK, 1998. To appear in IEEE Transactions on Computers.

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

[15] 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.

[16] N. Jouppi. Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[17] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 346–357, New York, June15–18 1997. ACM Press.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, 17–19 June 1998.

[28] D. Stiliadis and A. Varma. Selective victim caching: A method to improve the performance of direct-mapped caches. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume 1 : Architecture*, pages 412–421, Los Alamitos, CA, USA, January 1994. IEEE Computer Society Press.

[29] 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.

[30] 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 *Supercomputing '93*, pages 410–419. IEEE, November 1993.

[31] R. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS). In *Supercomputing '98*, 1998.

[32] 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.

A Tiled Programs

The figures show the tiled versions of LU factorization and SOR used in the experiments.

```
do jj=1,n,w
   do ii=1,n,h
      do k=1,n
         do j=max(k+1,jj),min(jj+w-1,n)
            do i=max(k+1,ii),min(ii+h-1,n)
               if (jj≤k+1≤jj+w-1 && j=max(k+1,jj)) then
                   a(i,k)=a(i,k)/a(k,k)
               a(i,j)=a(i,j)-a(i,k)*a(k,j)
```

Figure 13: Tiled LU factorization without pivoting

```
do jj=2,n+t-1,w
   do ii=2,n+t-2,h
      do k=max(1,jj-n+2),min(t,jj+w-2)
         do j=max(2,jj-k+1),min(n-1,jj+w-k)
            do i=max(2,ii-k+1),min(n-1,ii+h-k)
               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 14: Tiled SOR