# Defensive Loop Tiling for Shared Cache

Bin Bao *
Adobe Systems Incorporated
bbao@adobe.com

Chen Ding
Department of Computer Science,
University of Rochester,
Rochester, NY, USA
cding@cs.rochester.edu

## Abstract

Loop tiling is a compiler transformation that tailors an application's working set to fit in a cache hierarchy. On today's multicore processors, part of the hierarchy especially the last level cache (LLC) is shared. The available cache space in shared cache changes depending on co-run applications. Furthermore on machines with an inclusive cache hierarchy, the interference in the shared cache can cause evictions in the private cache, a problem known as the inclusion victims.

This paper presents defensive tiling, a set of compiler techniques to estimate the effect of cache sharing and then choose the tile sizes that can provide robust performance in co-run environments. The goal of the transformation is to optimize the use of the cache while at the same time guarding against interference. It is entirely a static technique and does not require program profiling. The paper shows how it can be integrated into a production-quality compiler and evalutes its effect on a set of tililing benchmarks for both program co-run and solo-run performance, using both simulation and testing on real systems.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Compilers

***General Terms*** Performance

***Keywords*** Loop tiling, Multicore, Cache sharing

## 1. Introduction

Loop tiling is a compiler optimization that reorganizes a loop nest so it computes on data tiles whose size can be adjusted to fit in one or more levels of cache. A basic problem in loop tiling is the selection of the best tile shape and size. The best strategy in the past utilizes the most space that does not cause data conflicts due to limited associativity (for examples in [5, 10, 18, 25, 32, 38, 39]). However, these methods do not consider the effect of cache sharing.

Contemporary chip multiprocessor (CMP) has greatly improved system throughput and power efficiency. An important characteristic in CMP is cache sharing. In a typical multi-core processor today, each core may have multiple levels of caches used exclusively by the core, but a group of cores would share the last level cache (LLC). Cache sharing allows a program to use the full LLC space when no one else is active. However, when running with other programs, sharing leads to interference. The portion of LLC occupied by a program's data can be hard to ascertain. If a program uses more than its share of cache, its performance can drop. The problem is especially serious for a tiled program since it depends on its data tiles residing in cache.

In this paper, we present defensive tiling to safeguard performance in the presence of interference. As a starting point, we may tile for private cache only. However, the solution is problematic on machines with an inclusive cache hierarchy, which includes most processor families from Intel. On these systems, the action in the shared LLC by one program can cause eviction in the private cache of another program.

Consider a 2-core toy machine with private L1 and shared L2. The cache sizes are 2 and 8 blocks respectively. Suppose we have two programs: one uses just one block $a$ and the other iterates over 8 other blocks. As block $a$ stays in the private cache, its copy in the shared cache is not accessed. As a result of good private-cache locality, $a$ becomes "stale" in the shared cache. In the meanwhile, program 2 constantly loads "fresh" data into the shared cache. After every 8 accesses, program 2 erases the old content of the shared cache and evicts $a$. To maintain inclusion, $a$ has to be purged from program 1's private cache as it leaves the shared cache. The next access to $a$ then incurs a cache miss. Following Jaleel et al. [20], we call such a cache miss an *inclusion victim*.

The following example shows the co-run traces and the private-cache misses incurred by program 1. The first miss

* The work was done when Bin Bao was a graduate student at the University of Rochester.



*CGO'13* 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

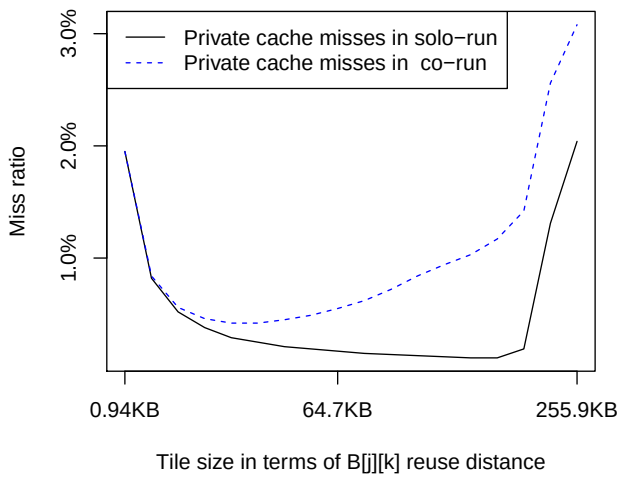



Figure 1: The effect of cache interference on tiled matrix multiply. For the tile size ranging from 0 to the size of L1 (256KB), the figure shows the private cache miss ratios of solo execution and of co-run with another program. The best tile size for the solo run incurs 10 times as many misses in the co-run execution, showing the dramatic impact of inclusion victim misses.

is compulsary (c), while the other two are inclusion victims (v).

| misses: | c | | | | | | | v | | | | | | | | v | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| prog. 1: | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | ... |
| prog. 2: | p | q | u | v | w | x | y | z | p | q | u | v | w | x | y | z | p ... |

Inclusion victim has been identified as a major factor that limits the performance of inclusive cache, particularly in the case of cache sharing [20]. As an optimization that improves data locality, loop tiling magnifies the problem. As a demonstration, we measure the number of inclusion victim misses in a simulator when running tiled matrix multiply with the STREAM benchmark [28] and compare it with the solo-run of the same matrix multiplication.

Figure 1 shows the simulated miss ratios for square tile sizes up to the size of the private L1 cache (256KB). To ease the comparison, we link the miss ratios into curves. The best tile size is given by the lowest point of a curve. Comparing the two curves, we see that the best tile size differs by a factor of 10 with and without cache interference. If we use the best tile size from the solo run in the co-run, the program would incur 10 times as many cache misses because of interference, in particular, inclusion victim misses.

The figure shows the importance of considering cache interference in loop tiling. On machines with inclusive cache, it is far from sufficient to tile for the private cache. Defensive tiling must minimize the number of inclusion victim misses, a problem we solve in this paper. We call the program optimization *defensive tiling*.

In the following, Section 2 describes a model of inclusion victim misses, the compiler technique for defensive tiling, its integration in the Open64 compiler, and a compiler analysis of interference. Section 3 evaluates defensive tiling using the PLUTO [3] benchmarks. The remaining sections discuss related work and conclude.

## 2. Defensive Tiling

In this section we describe loop tiling, examine the new problem of inclusion victim misses and present the techniques that enable defensive tiling.

### 2.1 Loop Tiling

Figure 2a shows the kernel code of matrix multiplication and Figure 2b the tiling transformation that we will use as an example throughout this section, where $N = 2048$.

$$
\begin{array}{l}
for(i = 0; i < N; i = i + 1) \\
\quad for(j = 0; j < N; j = j + 1) \\
\quad\quad for(k = 0; k < N; k = k + 1) \\
\quad\quad\quad C[i][j] = beta * C[i][j] + alpha * A[i][k] * B[k][j];
\end{array}
$$

(a) Original code

$$
\begin{array}{l}
for(jj = 0; jj < N; jj = jj + B_j) \\
\quad for(kk = 0; kk < N; kk = kk + B_k) \\
\quad\quad for(i = 0; i < N; i = i + 1) \\
\quad\quad\quad for(j = jj; j < min(jj + B_j, N); j = j + 1) \\
\quad\quad\quad\quad for(k = kk; k < min(kk + B_k, N); k = k + 1) \\
\quad\quad\quad\quad\quad C[i][j] = beta * C[i][j] + alpha * A[i][k] * B[k][j];
\end{array}
$$

(b) Tiled code

Figure 2: Loop tiling in matrix multiplication

The results presented in Figure 1 show the relation between loop tiling and cache performance. For ease of presentation, it shows the results for only square tiles ($B_j = B_k = B$) for $B$ from 10 to 180, with an increment of 10. The x-axis shows the 18 data points with the tile size measured by the reuse distance of $B[k][j]$ in i-loop from about 1KB to 256KB. In this experiment, the machine has private 8-way 256KB L1 and shared 16-way 2MB L2.

When the matrix multiplication is running alone, the L1 miss count goes down as the tile size increases, due to the greater utilization of the L1 cache. However, the benefit diminishes when $B$ grows larger than 160, and the reuse distance of $B[k][j]$ larger than 200KB. This is due to partly capacity, not enough L1 space is left for other data, and partly conflict, interference among data tiles. Such phenomenon is well known.

When the same matrix multiply is co-running with a stream benchmark, larger private-cache usage decreases the number of misses only until $B = 50$ or reuse distance of 24KB, after which the miss count increases rapidly. The best tile size occupies less than 10% of the private cache in this co-run case, in sharp contrast to near 80% in the solo-run

case. The reason is the interference from the peer program in the shared cache, which we model in the next section.

### 2.2 Inclusion Interference Modeling

The execution of a tiled program has a regular set of data being reused: the data tiles. The program computes on them for a duration before moving to the next set of tiles. In general, it may access other data blocks that are not reused. We introduce two metrics to represent this type of cache usage:

- *Reused data*, which is the volume of data being reused;
- *Active period*, which is the duration of the time the same collection of data is reused.

While a tiled program runs, peer programs bring their data into the shared cache and evict the data tiles. We model the interference using the following metric:

- *Survival window*, which is the time taken for co-run programs to access the amount of data equal to the size of the shared cache.

Consider a data tile that fits inside the private cache. Its copy in shared cache would be evicted by the end of each survival window and has to be reloaded, incurring inclusion victim misses in the size of the data tile. The following example demonstrates the three metrics:

```
prog. 1:  a a b a a c a a d a a e ...
prog. 2:  p q u v w x y z p q u v ...
```

In the first program, the reused datum is $a$, which we assume resides in private cache. Their active period is the full execution. Suppose the shared cache is of size 4. In every three accesses by each program, the two programs together access four blocks in the shared cache. Hence, the survival window is 3-access long. Our model predicts that program 1 incurs a inclusion victim miss after every 3 accesses.

In general, the following equation shows how to use the three metrics, the active period $ap(p_1)$, the reused data $reuse(p_1)$ of program 1, and the survival window from both programs $sw(p_1 + p_2)$, to compute the number of inclusion victim misses in program 1 $iv(p_1)$:

$$iv(p_1) = \frac{ap(p_1)}{sw(p_1 + p_2)} * reuse(p_1) \qquad (1)$$

The model shows that the inclusion victim misses may happen even without a peer program. In the last example, program 1 would still miss for $a$ in the private cache (after every 12 instead of 3 accesses). This suggests that defensive tiling may improve even the solo-run performance.

The model uses the logical rather than the physical time, which can introduce imprecsion when computing the joint survival window. Part of the problem is the mutual interaction between the co-run programs, as studied by Xiang et al. using a recursive equation and a fixed-point solution [41]. In this paper, we treat the interaction only qualitatively, in particular, its important effect in determining the "friendliness" of defensive tiling as discussed at the end of Section 2.4.

The remaining problem is when we do not know the co-run peers. In the following solution, we will add a compiler flag for a user to specify the level of defensivenss in anticipation of the amount of interference in shared cache. This can permit an adaptive defense strategy based on online analysis, but this is beyond the scope of the present paper.

Next we show how to estimate the above three factors inside a compiler, without profiling the execution of a target program.

### 2.3 Compiler Enhancement of Defensiveness

The compiler design for loop tiling can be divided into two parts. The first is the loop analysis and transformation. The second is the cost function to select the tiling parameters. In this work, we choose the Open64 compiler as a basis for defensive tiling, partly because its optimizer performance is among the best in production-quality compilers, and its loop tiling design is highly modular. In Open64, loop tiling is a component of the loop nest optimization (LNO), which includes a host of other transformations including loop fusion, distribution, interchanging and unroll and jam. The strategy for combining them is described in Wolf et al.[39] Part of the description (Section 3.2) deals with the problem of tile-size selection. The description is more intuitive than it is precise. Our design is based on the current Open64 design, which we review in detail before describing our extension.

The core of the parameter selection is a cost model which estimates the execution speed of a loop. The overall cost model is composed of two parts: a processor model that estimates the effect of instruction scheduling and register pressure, and a cache model that estimates the miss counts.

The cache model is parameterized by the tile size. For example for the *ijk* loop nest of matrix multiplication in Figure 2b, the compiler assumes that the inner *j,k* loops are tiled with tile sizes $B_j$ and $B_k$. The cache model is a cost function that computes the miss count based on $B_j$ and $B_k$. The best tile sizes are found through a (binary) search to be the ones that minimize the cost function. In this way, for an *n*-level loop that can be tiled, the compiler finds the tile sizes with a minimum cost. If some of the loop levels are permutable, the search will also include loop interchange.

We next describe the formulas used in the cost function and then add the inclusion-victim miss model. We will show how to estimate the length of the active period, the amount of reused data, and the relation with the survival window. Once we extend the cost function to include cache interference, defensive tiling is accomplished by the same search procedure to find the best tile sizes.

The first step of constructing the cost function is computing the footprint of each loop, that is, the volume of data touched by one execution of the loop. For matrix multiplication (Figure 2b), the *footprint* of i, j, and k loop are de-

noted as $F_i$, $F_j$ and $F_k$ respectively. The formulas are: $F_i = 8*(N*B_k+B_j*B_k+N*B_j)$, $F_j = 8*(B_k+B_j*B_k+B_j)$, $F_k = 8*(B_k+B_k+1)$.

Next the compiler computes the cache requirement of each loop. The requirement is actually computed for each iteration. It is the total amount of data used by each iteration, whether the data is reused between iterations or not. In addition, it must include complete cache blocks, so it must consider spatial reuse. If a cache block contains $d_i$ data elements, the requirement for loop $i$ of matrix multiply is as follows. Since each $i$ iteration is a loop $j$, we denote the per-iteration cache requirement of loop $i$ as $R_j$:

$$R_j = F_j + (d_i - 1) * \frac{F_i - F_j}{N - 1} \quad (2)$$

Part of the required cache space holds the reused data, in the amount

$$reuse_j = F_j - \frac{F_i - F_j}{N} \quad (3)$$

To understand the formula, notice that the footprint numbers of $F_i, F_j$ are not additive. The loop $i$ has $N$ instances of loop $j$, but $F_i$ is smaller than $N * F_j$. The difference is due to the reuse of the data tile, of size $B_j*B_k$ in this case. The formula of $reuse_j$ formalizes this relation.

Based on the requirement and the data reuse, the number of cache misses for each i-loop iteration (loop $j$) is calculated using Equation 4 as follows.

$$CM_j = \frac{F_i}{N} + (\alpha * \frac{R_i}{ecsz} + \beta * \frac{|R_i - ecsz|^+}{ecsz}) * reuse_j \quad (4)$$

where $ecsz$ is the effective cache size. The effective cache size is a fraction of the actual size to take into account of the fact that cache is not fully associative. The discount is based on an empirical formula involving the cache size, cache line size and the cache associativity.

$\alpha$ and $\beta$ are factors used to model program induced cache conflict and capacity misses. The former penalizes if the cache requirement is close to the cache size. The second penalizes if the requirement exceeds the cache size (the term is 0 if $R_i \leq ecsz$). They are computed as a fraction by which the data reuse is not realized as cache reuse.

Now we are ready to introduce the cache interference model. Assuming each iteration of loop $i$ is a loop $j$, the amount of data reused in private cache is $reuse_j$ and the length of the active period is the duration of loop $i$. The eviction frequency is the ratio of the active period to the survival window. Both terms are time and difficult to estimate inside the compiler. Instead, we convert the time ratio into the data ratio. In particular, we assume it is proportional to ratio of the footprint of loop $i$ to a fraction of the *shared* cache size. The formula is given by Equation 5 as follows:

$$IV_j = \frac{F_i}{\frac{scsz}{\gamma}} * reuse_j \quad (5)$$

We should note the distinction between $ecsz$, which is the effective size of the private cache, and $scsz$, which is the size of the shared cache. $\gamma$ is a number greater than 1. It represents the *defensiveness*. The larger is the number, the shorter is the survival window. Ideally, the defensiveness is tuned based on the co-run cache interference. In implementation, we control $\gamma$ using a compiler option *"-LNO:blocking_defensiveness"*. We will experimentally study the effect of $\gamma$ in Section 3.

The revised formula for the miss estimate, $CM'_j$ is then

$$CM'_j = CM_j + IV_j \quad (6)$$

Given this cost function, defensive tiling continues by searching for the tile sizes that minimize $CM'_j$, employing loop interchange and other loop nest transformations as before [39].

In the above description we have used the matrix multiplication as an example. The main assumption is that loop $j$ is one iteration of loop $i$. The $i, j$ distinction helps to clarify when we describe the formulas that use both the metrics for the loop and the metrics for one of its iterations. The formulas for a generic loop $i$ are the same except that we replace the metrics for loop $j$ with those for an iteration of loop $i$.

### 2.4 Compiler Analysis of "Friendliness"

A program's role in cache interference is two sided. On the one side, it is affected by the interference from others. Jiang et al. called it the sensitivity [21]. Defensive tiling is to minimize the sensitivity. The other side is the program's interference to others. It has been called politeness and friendliness [22] and niceness [35]. Various notions of friendliness have been estimated using on-line measurements (e.g. [52]), profiling (e.g. [42, 43] in linear and real time), and simulation and modeling [34, 40].

In this section, we describe as far as we know the first compiler analysis that estimates the friendliness. The analysis is not used for defensive tiling. Still, through the analysis we can see the impact of defensive tiling on others and identify any conflict between optimizing for defensiveness and in the future, optimizing for friendliness.

We start with reuse distance, which has been shown amenable to static analysis through dependence analysis [6, 7] and reuse distance equations [2]. Given a loop nest, the reuse distance histogram shows the distribution of reuse distances. For an execution of matrix multiply, the top graph in Figure 3 shows that one third of references have a reuse distance of 24B, 2.4KB and 178KB respectively.

Compiler analysis produces a histogram parameterized by the loop trip counts. For tiled matrix multiply in Figure 2b, the reuse distances for different references are given in Table 1. The per-loop histogram is the aggregate of the per-reference histograms.

The second step computes the (capacity) miss ratio curve from reuse distance. Assuming that $rd(c)$ percentage refer-

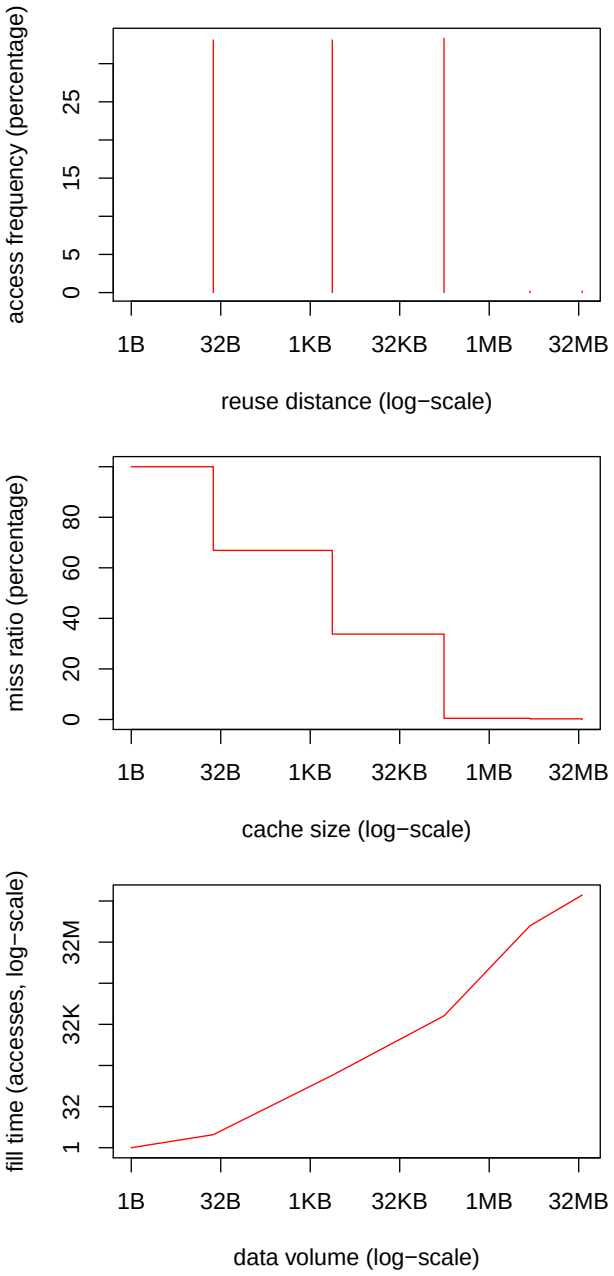



Figure 3: Deriving the fill time (friendliness) using the reuse distance.

ences have a reuse distance $c$, the miss ratio $mr(c)$, including both capacity and compulsary misses, is:

$$mr(c) = \Sigma_{i=c+1}^{\infty} rd(i) \tag{7}$$

The reuse distance is $\infty$ if it is the first access to a data block. As an example, the middle graph in Figure 3 shows the miss rate curve for the reuse distance shown in the upper graph.

The third step computes the average inter-miss time, which is the execution time divided by the miss count. For static analysis, we use the logical time measured by the number of data accesses (between misses).

| Loop | Array | Reuse distance (in bytes) |
|---|---|---|
| k | $C[i][j]$ | $8*3$ |
| j | $A[i][k]$ | $8*1+8*B_k+8*B_k$ |
| i | $B[k][j]$ | $8*B_j+8*B_k+8*B_k*B_j$ |
| kk | $C[i][j]$ | $8*N*B_j+8*N*B_k+8*B_k*B_j$ |
| jj | $A[i][k]$ | $8*N*B_j+8*N*N+8*N*B_j$ |

Table 1: Parameterized reuse distance for tiled matrix multiply in Figure 2b

$$im(c) = \frac{1}{mr(c)} \tag{8}$$

The fourth step computes the fill time $vt(v)$, the time taken for a program to access data in the amount of $v$. The fill time is the residence time of a data block in fully-associative LRU cache of size $v$, if it is not reused after the initial access. It is also the average length of time the program takes to fill up an empty cache.

The fill time and inter-miss time are related in the following way. First we let a program run for $vt(c)$ time and access $c$ data blocks. Then we let the program continue to run until it touches a new data block. This is the time $vt(c+1)$. The interval from time $vt(c)$ to time $vt(c+1)$ is the average time before the next capacity or compulsory miss, that is, the inter-miss time $im(c)$. Therefore, the lifetime $vt(c+1)$ can be computed:

$$vt(c+1) = im(c) + vt(c) \tag{9}$$

Using the same relation to compute $vt(c), vt(c-1), \ldots$ and observing $im(0) = vt(1) - vt(0) = 1$, we have

$$vt(c) = \Sigma_{i=1}^{c-1} im(i) + 1 \tag{10}$$

The friendliness of a program is given by its lifetime function $vt(c)$, which is the average time it fills the cache of size $c$ with newly accessed data and evicts the data from peer programs that have not been used in the last $vt(c)$ time period. In other words, the data access of this program leads to the survival window of $vt(c)$ in size-$c$ cache for the data from the peer programs.

Combining the preceding equations, we have the (static) model of the survival window $sw$ as computed from the reuse distance.

$$sw(c) = lf(c) = \Sigma_{i=0}^{c-1} im(i) = \Sigma_{i=0}^{c-1} \frac{1}{\Sigma_{j=i}^{\infty} rd(j)} \tag{11}$$

Consider the extreme cases as a sanity check. The least friendly program loads a new data block at each access, so $rd(\infty) = 1$. The survival window it gives to others is $sw(c) = c$. The friendliest program uses a single data block,

so $rd(1) \approx 1$. The survival window is $sw(1) = 1$ and $sw(c) = \infty$ for $c > 1$. A real program is somewhere in between these two extremes. Tiled programs have few long distance reuses. As a result, they are among the friendliest programs. Figure 3 shows the lifetime of matrix multiply increases rapidly with the cache size, making it an amicable player in shared cache.

The "friendliness" analysis presented in this section helps to understand the use of defensive tiling. First, different levels of defensiveness are needed depending on how friendly the co-run programs are. We can now quantify this friendliess. In the evaluation, we will test self co-run and co-run with the streaming benchmark and see the effect of friendliness.

Second, the aggregate interference from multiple peer programs can be computed by combining the data growth in their lifetime functions. Given two programs $A, B$, the combined lifetime $lf_{A+B}(c)$ is such that $lf_A(c_A) = lf_B(c_B) = lf_{A+B}(c)$ and $c_A + c_B = c$. The window of survival shortens with every additional program in the mix.

Third, the fill time for tiled programs can be parameterized by the tile sizes just as the reuse distance is. Since tiling reduces the reuse distance, it increases the fill time. As a result, it makes a program friendlier. On the other hand, because defensive tiling uses smaller tile sizes, it does not reduce the reuse distance as aggressively. On the surface, we may see that by being defensive, a program becomes less polite.

The interference is a coupled phenomenon. A normally tiled program actually incurs more misses in a co-run than a defensive program does. Since its inter-miss time is actually shorter, so is its actual fill time. With the model of friendliness presented in this section, we now see the second benefit of defensive tiling: it improves both the defensivenss and the friendliness of the tiled program, if the amount of inclusion victim misses is significant. Like defensive driving, it makes the shared environment safer both for itself and for others.

# 3. Evaluation

***Implementation*** The defensive tiling is implemented in the LNO component in Open64 5.0. The Loop Nest Optimizer in Open64 combines a set of loop transformations such as loop fusion, loop interchange, loop tiling, and unroll and jam [39]. We use -O3 to compile the test programs. We tile loops only for the private cache and do not consider tiling for TLB.

***Test Suite*** We have compiled all 25 benchmarks distributed with the PLUTO compiler [3]. 5 programs are not included because Open64 can not tile the imperfectly nested loops in them, for example, the one in *dsyrk*. 2 more programs cannot be tiled for other reasons. Another 13 are excluded because they do not show a significant problem of inclusion victim misses, that is, their solo-run and co-run miss counts do not differ much. A common reason seems to be that the amount of data reuse is relatively small. For instance, the *mvt* benchmark is a matrix-vector multiplication kernel in which the reuse only happens on the vector data. According to the model in Section 2.3, when the reused data are few, the number of inclusion victim misses is low relative to the number of misses caused by the matrix data. Indeed when measured in simulation, the number of L1 misses increases by less than 0.1% from the solo- to the co-run. After removing these programs, we have 5 remaining programs which have significant data reuse in private cache and for which Open64 can perform the tiling transformation.

Next we evaluate defensive tiling first on a cache simulator to measure the miss ratio and on real hardware to measure the performance.

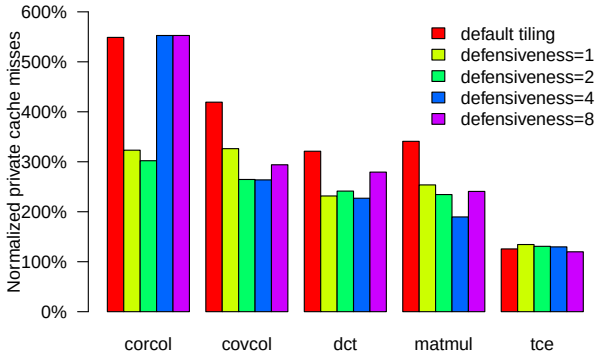
## 3.1 The Effect on Cache

For simulation, we have extended the basic cache simulator in Intel Pin tool [27] to simulate a multi-level CMP cache. The simulator is designed similar to CMP$im[19] (CMP$im is not publicly available). One difference is that our simulator does not include an L1 cache as they did, because the L1 cache does not significantly affect the interference between L2 and L3, which we model using our simulator. L1 has a performance impact, which we will include when testing on a real system. Other than L1, we use the same cache configuration used by Jaleel et al.: 2-core CMP, each has 8-way 256KB unified private cache, and both share inclusive 16-way 2MB unified cache [20].

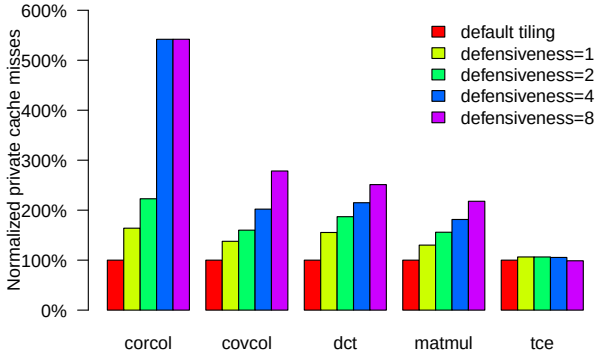
Our CMP simulator is Pin-based and trace-driven. The simulator reads the same binaries as those running on the real machine. Then Pin will instrument binaries and run the cache simulation. The cache sharing is implemented through process shared memory. With the cache simulator, we can measure the total number of misses in the private cache. For this experiment, we set the cache parameters in the Open64 compiler according to the simulated cache configuration: private 256KB cache and shared 2MB cache.

We test the 5 PLUTO benchmarks in solo-run and in co-run with a STREAM benchmark on the neighboring core. We test five versions of each program: original Open64 tiling and defensive tiling with the defensiveness level ($\gamma$) set to 1, 2, 4 and 8. Figures 4a and 4b show the relative number of private cache misses. The former is for the 5 versions when the program co-runs with STREAM, and the latter for the 5 program versions when the program runs alone. In each program, the number of misses is normalized to that of the default Open64 tiling, so the first bar in each group in Figure 4b is always 1.

The default tiling is vulnerable to program co-run. We see in Figure 4a that the number of private cache misses (the first bar in each group) increases by 321% to 449% in the first four programs, *corcol, covcol, dct, matmul*, and 26% in the last program *tce*. Inclusion victim is the culprit as it is




(a) Co-run simulation result




(b) Solo-run simulation result

Figure 4: Private cache miss comparison between the orignal Open64 loop tiling and the defensive tiling for single run and co-run cases.

the only way the STREAM benchmark can affect the private cache of the tiled program.

Defensive tiling reduces the number of misses by as much as 45% for *corcol* and *matmul*, 37% for *covcol*, over 29% for *dct*, and less than 5% for *tce*. For the first four programs, the defensiveness (*"-LNO:blocking_defensiveness"*) of 2 gives consistently good cache performance. The best performance is seen when the defensiveness is either 2 or 4.

The effect of defensive tiling on solo runs is shown in Figure 4b. As tiling become more defensive, there is a steady increase in the number of cache misses. This is expected since being more aggressive means choosing smaller tile sizes.

Being too aggressive may be counter productive. In *corcol*, the defensiveness of 4 and 8 cause so much loss in cache reuse that the inclusion victim has a negligible impact. As a result, defensive tiling (at the highest two defensiveness levels) does not improve the program co-run results. At level 2, however, defensive tiling is not overly conservative and shows over 45% reduction in the co-run miss count.

Defensive tiling does not improve *tce* significantly (5%). Open64 default tiling causes 26% more private cache misses in the co-run, which is far less significant than in the other four benchmarks but it is not negligible. The loops in *tce* all have 5 levels. Further tuning may be needed because of the program complexity and the narrow room for improvement.

### 3.2 The Effect on Performance

We now test defensive tiling on a real machine. Intel Xeon E5520 Nehalem processor has four cores. Each has 32KB L1 and 256KB L2 cache. While the L2 cache is non-inclusive (wrt L1), the four cores share an inclusive 8MB L3 cache. We should note that the the processor model in the Open64 compiler is not tuned for Intel x86. For default tiling, we have specified a different shared cache size (8MB) than in the simulation (2MB), but the compiler still produces the same code as it did for the simulated machine, because the size of private cache in both cases is 256KB.

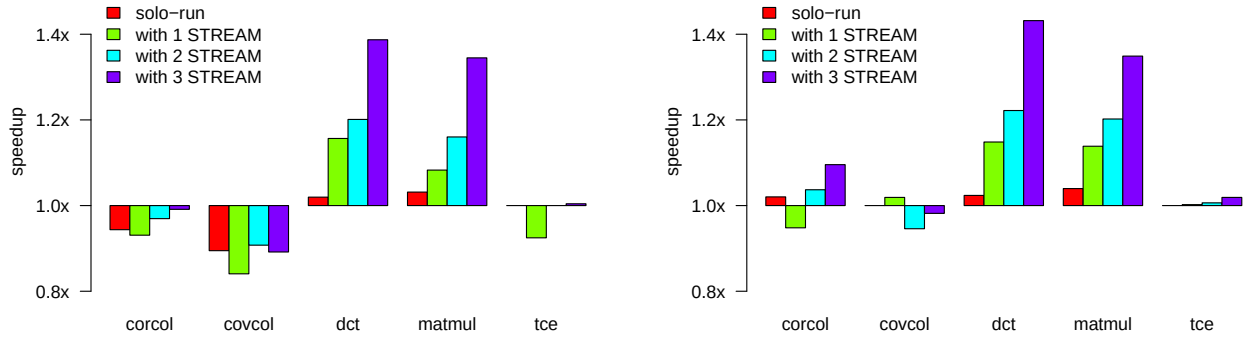
***Co-run with STREAM*** With the four cores in the Nehalem processor, we can test the co-run performance for up to 3 STREAM benchmarks. Figure 5 shows four graphs, each for a level of defensiveness. In each graph, five programs are shown. Each has four bars showing the normalized performance of the solo run and the three co-runs with STREAM.

Overall we see improvements in most cases by defensive tiling. Two programs, *dct,matmul*, show double-digit percentage improvements in almost all co-run cases and all defensiveness levels. For *dct*, the improvement increases from 4% in the solo run to near 40% when co-run with 3 STREAM programs. *matmul* shows similar variation. The improvements are especially high when the pressure in the shared cache is high.

The *corcol* benchmark does not benefit as much from the defensive tiling when the defensiveness level is 1 and 2, which suggests that the active period of the tiled data is rather long. When the defensiveness is set to 4 as in Figure 5c, the improvements become significant, as much as 1.17x.

The *covcol* benchmark shows similar variation when we vary the defensiveness level. At the lowest level, defensive tiling causes similar degradation in all co-run tests. Greater defensiveness helps to restore the performance. When the defensiveness level is set to 4 and 8, the speedup for the solo-run is greater than 1.2x. The co-run improvements, however, are lower. The other unexpected result is the 1.18x speedup for *tce* in Figure 5d. We do not yet have an explanation for these two results. For *tce*, we do not expect to see significant improvements based on our simulation, which is the case except for the 1.18x speedup.

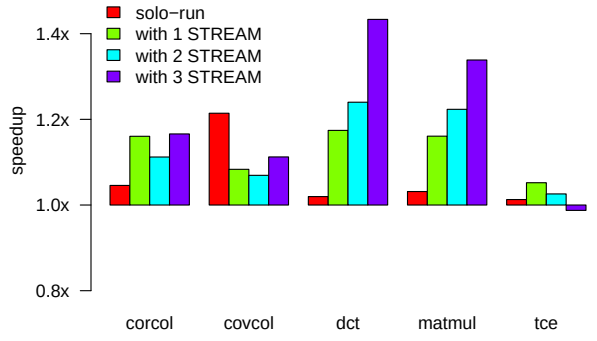
***Symmetric Co-runs*** We have tested each PLUTO benchmark running with one, two, or three of its own replicas. Figure 6 shows the result for the solo- and co-run tests when



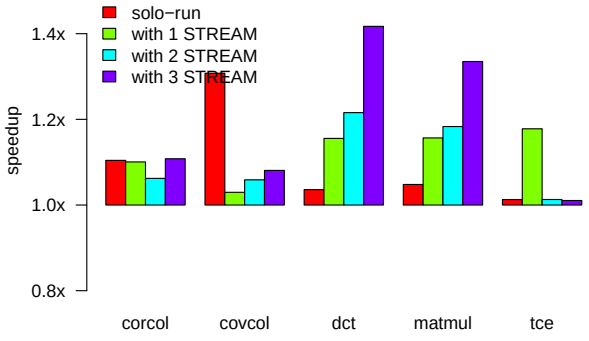

(a) -LNO:blocking_defensiveness = 1




(b) -LNO:blocking_defensiveness = 2




(c) -LNO:blocking_defensiveness = 4




(d) -LNO:blocking_defensiveness = 8

Figure 5: Speedup of defensive tiling over Open64 default tiling as measured on Intel Nehalem. Each benchmark co-runs with 1 to 3 STREAM benchmarks. The four graphs show defensive tiling with $\gamma = 1, 2, 4, 8$.

the defensiveness level is set to 4. The baseline is the default tiling in the solo run and in 2 to 4 symmetric co-runs. The first bar in each group shows the same speedup as those in Figure 5c.

All tests show improvements, although most are small and lower than the solo-run improvement. The results for other defensiveness levels are similar, thus we omit them for brevity.

Defensive tiling seems not effective since in all programs the lead over the default tiling is narrowed, often significantly. For explanation we need to examine the friendliness as defined and discussed in Section 2.4. The tiled programs have excellent locality, so they are among the friendliest peers and do not yield much room for improvement by defensive tiling. More importantly, as discussed at the end of Section 2.4, in these tests the default tiling produces friendlier code than defensive tiling. Hence the default tiling co-runs better and regains some of the losses we see in the solo-run test.

***The Defensiveness of Cache Oblivious Algorithms*** Cache oblivious algorithms recursively divide the computation [12]. For example, a matrix multiplication can be broken into eight sub-matrix multiplications, and the subproblems can be further divided until a threshold size is reached. The recursion in effect tiles the computation for all possible cache levels. Yi et al. [49] developed a compiler transformation to convert loop nests into a recursive form. Our test suite and theirs have one overlap—the matrix multiply.

Figure 7 shows the performance comparison between cache oblivious algorithms and the default Open64 tiling in the solo- and co-run tests with 1 to 3 STREAM benchmarks. The numbers are reported for different threshold sizes from as small as 16 to as large as 256. When the termination size is 16 and 32, the recursive version shows 20% and over 40% improvements in the high cache-contention cases (2 and 3 streaming peers). The largest improvement exceeds that of defensive tiling. Open64 shows better performance in other cases. The results suggest that although the recursive version

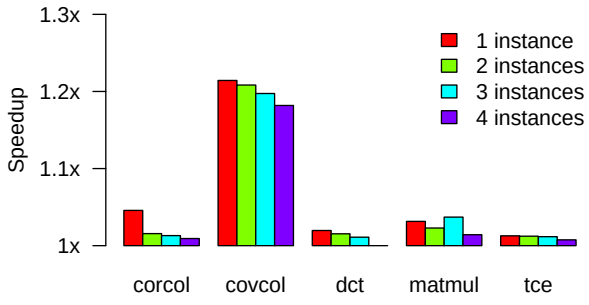



Figure 6: Speedup of the defensive tiling over the default tiling for the solo-run and symmetrical co-runs with 2 to 4 replicas. The defensiveness level is 4.

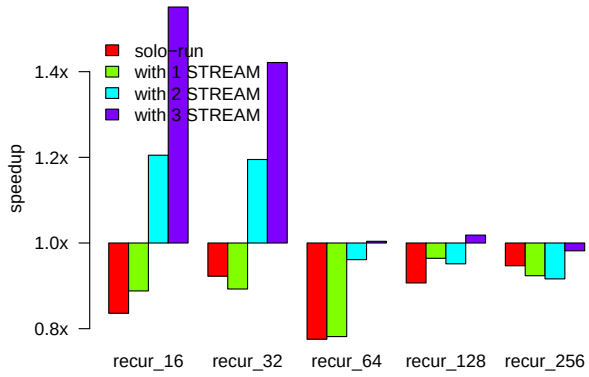



Figure 7: Speedup of cache oblivious matrix multiplication by Yi et al. [49] over default tiling.

has a higher overhead in the solo run, there can be significant benefits gained from being defensive. While a detailed analysis is beyond the scope of the paper, we note that the higher improvements in small threshold sizes are consistent with our model and technique of defensive tiling, which also seek to reduce the size of data reuse.

## 4. Related Work

***Peer-Aware Program Optimization*** QoS-Compile from Tang et al. [35] is the first compiler solution to mitigate memory hierarchy contention for independent co-located programs on multicore processors. The optimization first requires a profiling pass to identify contentious code regions. Once the high-interference regions of a program are found through profiling and modeling, the compiler will pad non-memory instructions and insert intermittent sleep in those regions to throttle back the program's memory request rate. The two transformations are done at the binary level.

Defensive tiling differs in several aspects. First, it improves the performance of the transformed code, instead of slowing it down to make its peer run faster. Second, it is a static technique at the loop level and does not require any profiling information. The static notions of defensiveness and friendliness are new, so is the model of cache inclusion victim misses. However, defensive tiling is limited to certain kinds of applications, while QoS-Compile is generally applicable.

***Single-level Tiling*** For a single-level cache, loop tiling has been used to reduce capacity miss [5, 18, 25, 38]. Temam et al. [11, 37] showed that the number of conflict misses in numerical code can be modeled and data locality optimizations should consider conflict misses. Coleman and McKinley [10] developed a Tile Size Selection algorithm which eliminates self-interference misses and minimizes cross-interference misses. Ghosh et al. [13] proposed Cache Miss Equations which consider both loop structure and data layout, including loop tiling and array padding. Hsu and Kremer [16] gave several algorithms to combine tile selection and array padding. Huang et al. combined loop tiling with data tiling and showed robust performance by a single tile size for different problem sizes [17].

***Multi-level Tiling*** Open64 uses a combined model to unify several loop optimizations including loop tiling [39]. Unified transformations have been studied with unimodular transformation to maximize reuse [46], in data shackling using high-dimensional optimization [24], and for multiple loop nests, through loop fusion enabled by loop tiling and array copying [47]. For a two-level cache, Open64 performs loop tiling for the L1 cache first, and then continues to transform inter-tile loops for the L2 cache if it is profitable. Mitchell et al. [29] proposed global multi-level cost functions to guide the choice of optimal tile size and shape. Rivera and Tseng explored the impact of multi-level caches on data locality transformations [33]. For loop tiling, they pointed out that simply targeting the L1 cache often gives near best performance. Renganarayana and Rajopadhye [31] formalized the tile selection problem as Geometric Programming, and multi-level tiling could be solved recursively.

***Auto-tuning of Loop Tiling*** Rather than static tile selection, auto-tuning searches through different versions of compiled code at the installation time and find the best choice for the host machine. It is known as iterative compilation. Compiler models and heuristics were used to guide auto-tuning. Kisuki et al. [23] implemented an iterative compilation system which finds good tile sizes and unrolling factors in a trimmed optimization space. Chen et al. [8] included more optimizations such as unroll-and-jam [4] and prefetching. They also considered optimizing across multiple cache levels. More recent work includes a general type of parameterization using linear inequalities in PTile [1], dynamic tile-size variation [36], and programmable control in POET to tune not just tiling but others such as unroll-and-

jam and parallelization through scripted transformations and script reuse [48]. Hall et al. called the scripts transformation recipes [14].

***Tiling for Parallel Programs*** Loop tiling can be used to reduce the communication overhead between processors[15, 26, 30, 44]. Xue gave a book-length treatment [45]. On multicore processors, Zhang et al. tiled multi-threaded code for shared cache [50], and Chen et al. [9] applied tiling to pipeline MapReduce applications. Recently, Zhou et al. [51] showed hierarchical overlapped tiling can reduce communication overhead without introducing much redundant computation and can be used to transform OpenCL programs. The nature of parallel-loop tiling is collaborative rather than defensive. The transformation in one task knows the access pattern of other threads. Defensive tiling deals with unknown peers that share cache. The specific problem of inclusion victim has not been considered in prior compiler analysis.

## 5. Summary

This paper has presented compiler analysis of friendliness and defensiveness of loop-based programs. Based on these analyses, it has developed an entirely static framework for peer-aware program optimization. The framework has been implemented in a production-quality compiler, and has been shown to significantly reduce slowdowns caused by inclusion victim misses on today's multicore machines.

***Acknowledgement*** We wish to thank Shin-Ming Liu and Sun Chan for the advice and guidance in the past on the design and the use of the Open64 compiler, which made this research possible. Qing Yi provided the code used in the paper for cache-oblivious tiling. The presentation was improved thanks to the feedback of the anonymous reviewers of CGO 2013 and the insightful comments from Xiaoya Xiang, Jingling Xue, Shin-Ming Liu, and Sandhya Dwarkadas. The funding for the research is provided in part by the National Science Foundation (Contract No. CCF-1116104, CCF-0963759, CNS-0834566), several IBM CAS Faculty Fellowships, and a grant from Huawei.

## References

[1] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of CGO*, pages 200–209, 2010.

[2] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of PLDI*, pages 101–113, 2008.

[4] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. on Prog. Lang. and Sys.*, 16(6):1768–1810, 1994.

[5] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *ASPLOS*, pages 252–262, 1994.

[6] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of ICS*, pages 150–159, 2003.

[7] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *Proceedings of ICS*, pages 295–304, 2010.

[8] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of CGO*, pages 111–122, 2005.

[9] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of PACT*, pages 523–534, 2010.

[10] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of PLDI*, pages 279–290, 1995.

[11] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interferences on blocked loops: A case study with matric-vector multiply. *ACM Trans. on Prog. Lang. and Sys.*, 17(4): 561–575, 1995.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of Symposium on Foundations of Computer Science*, Oct. 1999.

[13] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. on Prog. Lang. and Sys.*, 21(4), 1999.

[14] M. W. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the LCPC Workshop*, pages 50–64, 2009.

[15] K. Högstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):307–321, 2003.

[16] C.-H. Hsu and U. Kremer. A quantitative analysis of tile size selection algorithms. *The Journal of Supercomputing*, 27(3): 279–294, 2004.

[17] Q. Huang, J. Xue, and X. Vera. Code tiling for improving the cache performance of pde solvers. In *Proceedings of ICPP*, 2003.

[18] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of POPL*, pages 319–329, 1988.

[19] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.

[20] A. Jaleel, E. Borch, M. Bhandaru, S. C. S. Jr., and J. S. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of ACM/IEEE MICRO*, pages 151–162, 2010.

[21] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiproces-

sors. In *Proceedings of HiPEAC*, pages 201–215, 2010.

[22] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1192–1205, 2011.

[23] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of PACT*, pages 237–248, 2000.

[24] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *Proceedings of PLDI*, pages 346–357, 1997.

[25] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS*, pages 63–74, 1991.

[26] L. Liu, L. Chen, C. Wu, and X. Feng. Global tiling for communication minimal parallelization on distributed memory systems. In *Proceedings of the Euro-Par Conference*, pages 382–391, 2008.

[27] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[28] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[29] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[30] J. Ramanujam and P. Sadayappan. Tiling multidimensional itertion spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[31] L. Renganarayanan and S. V. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 18, 2004.

[32] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of PLDI*, pages 38–49, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: http://doi.acm.org/10.1145/277650.277661.

[33] G. Rivera and C.-W. Tseng. Locality optimizations for multilevel caches. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 2, 1999.

[34] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.

[35] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for QoS in warehouse scale computers. In *Proceedings of CGO*, pages 1–12, 2012.

[36] S. Tavarageri, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *Proceedings of the International Conference on High Performance Computing*, pages 1–10, 2011.

[37] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of SIGMETRICS*, pages 261–271, 1994.

[38] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of PLDI*, pages 30–44, 1991.

[39] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. *International Journal of Parallel Programming*, 26(4):479–503, 1998.

[40] M.-J. Wu and D. Yeung. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of PACT*, pages 264–275, 2011.

[41] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPOPP*, pages 91–102, 2011.

[42] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of PACT*, pages 350–360, 2011.

[43] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache conscious task regrouping on multicore processors. In *Proceedings of CCGrid*, 2012.

[44] J. Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

[45] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-7933-0.

[46] J. Xue and C.-H. Huang. Reuse-driven tiling for improving data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.

[47] J. Xue, Q. Huang, and M. Guo. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *Proceedings of ICPP*, pages 107–115, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2380-3. doi: 10.1109/ICPP.2005.37. URL `http://dx.doi.org/10.1109/ICPP.2005.37`.

[48] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *Proceedings of CGO*, pages 97–106, 2011.

[49] Q. Yi, V. S. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of PLDI*, pages 169–181, 2000.

[50] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of PPOPP*, pages 203–212, 2010.

[51] X. Zhou, J. P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. A. Padua. Hierarchical overlapped tiling. In *Proceedings of CGO*, pages 207–218, 2012.

[52] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of ASPLOS*, pages 129–142, 2010.