A Case for a Working-set-based Memory Hierarchy

Steve Carr
Department of Computer Science
Michigan Technological University
Houghton MI 49931–1295, USA
carr@mtu.edu

Soner Önder
Department of Computer Science
Michigan Technological University
Houghton MI 49931–1295, USA
soner@mtu.edu

ABSTRACT

Modern microprocessor designs continue to obtain impressive performance gains through increasing clock rates and advances in the parallelism obtained via micro-architecture design. Unfortunately, corresponding improvements in memory design technology have not been realized, resulting in latencies of over 100 cycles between processors and main memory. This ever-increasing gap in speed has pushed the current memory-hierarchy approach to its limit.

Traditional approaches to memory-hierarchy management have not yielded satisfactory results. Hardware solutions require more power and energy than desired and do not scale well. Compiler solutions tend to miss too many optimization opportunities because of limited compile-time knowledge of run-time behavior. This paper explores a different approach that combines both approaches by making use of the static knowledge obtained by the compiler in the dynamic decision making of the micro-architecture. We propose a memory-hierarchy design based on working sets that uses compile-time annotations regarding the working set of memory operations to guide cache placement decisions.

Our experiments show that a working-set-based memory hierarchy can significantly reduce the miss rate for memory-intensive tiled kernels by limiting cross interference. The working-set-based memory hierarchy allows the compiler to tile many loops without concern for cross interference in the cache, making tile size choice easier. In addition, the compiler can more easily tailor tile choices to the separate needs of different working sets.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories, interleaved memories*; D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*

General Terms

Algorithms, Languages

Keywords

loop tiling, cache design

CF'05, May 4–6, 2005, Ischia, Italy.

1. INTRODUCTION

Modern microprocessors require efficient operation of the memory hierarchy to achieve high performance. In this respect, the continually increasing gap between the speed of the processors and the speed of the memory pushes the current memory hierarchy approach to its limits. In modern architectures, each cache miss is becoming increasingly difficult to tolerate using traditional means. Hardware-only solutions tend to scale poorly, while compiler-only solutions rely on information that may not be available at compile time. To help alleviate these problems, we propose a combined compiler/hardware solution to deal with cache misses caused by *cache interference*.

Interference in scientific applications comes in two forms: *cross* and *self*. Cross interference occurs when multiple array locations that are needed in the cache simultaneously map to the same cache location. Self interference occurs when multiple locations in the same array map to the same location. Previous work has detailed the detrimental effects of cache interference on loop tiling when using the wrong combination of array size, tile size and cache layout [15]. As an example, consider the graph in Figure 1. This graph shows the miss rate for a tiled two-dimensional loop from 171.swim on an 8K direct-mapped cache as a function of array size. The loop tiling factor is the same for each array size. The figure shows that the miss rate of tiled loops can be highly dependent upon the interaction between the tile size and the array sizes. In this paper, we present a new approach to minimizing cross interference and examine the effectiveness of the approach on tiled loops.

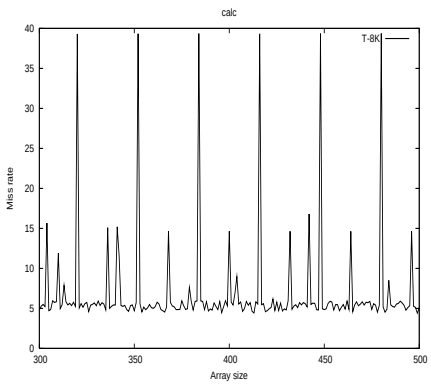


Figure 1: Miss Rates of Tiled Loop

To deal with cache interference, current approaches to loop tiling require knowledge of the size and layout of arrays in memory [8, 11, 18, 23]. Unfortunately, in pre-compiled library code or for dy-

namically allocated arrays this information may not be available. Current architectural approaches to deal with cache interference use skewed associativity [2] or prime-based indexing functions [25, 22, 14]. While these architectural techniques may not be applicable to the L1 cache, they are orthogonal to our approach and can be used to limit self interference. Rather than rely solely on the compiler or the cache design to lessen the effects of interference, our approach utilizes the strengths of the compiler and architecture together. We expose compile-time analysis related to the working sets of a program to the cache controller to enable the controller to make better decisions regarding layout in the cache. We try to answer the question, "Can we provide consistent cache performance across a variety of array sizes without tuning tiled algorithms specifically based on array size?"

In order to make a compiler/micro-architecture approach successful, we have identified and developed key techniques which are necessary to communicate only the relevant information from the compiler to the micro-architecture in a condensed way using *sets*. Instead of communicating completely precise information about a given load or store operation, we construct *working sets* of the load/store instructions at compile time and communicate the *set membership* to the hardware using very small integers on a per memory instruction basis. The hardware can then make quite effective use of this information under the guidance of precise run-time information and enhance it as deemed necessary.

In this paper, we show how to limit cross interference misses by using the working-set information collected by the compiler to help guide placement in the cache. This set information is associated with a particular load/store operation by the compiler and hence can be obtained by the micro-architecture directly from the instruction fields or by using the instruction pointer. We show that utilizing working-set information can effectively lower the miss rate of common computational kernels by limiting the amount of interference in the cache between distinct working sets.

We begin this paper in Section 2 with a review of the related work. Then, in Section 3 we give an overview of how we determine working-set colors in the compiler. In Section 4 we describe a cache organization that utilizes the color information and in Section 5 we describe an experiment that evaluates our techniques on a set of common computational kernels. Finally, in Section 6 we present our conclusions and propose future work.

2. RELATED WORK

Loop transformations have long been proposed and evaluated for improving the cache performance of loops. These techniques include loop permutation [6, 17, 24], fusion [17] and tiling [4, 5, 15, 19, 24]. McKinley and Temam [16] show that conflict misses may represent about half of all misses for typical programs. To address this issue, researchers have looked at choosing tile sizes for loop tiling that minimize both cross and self interference. Lam, *et al.* [15], select the largest tile size that that does not cause self interference. Coleman and McKinley [5] maximize the tile size while eliminating self interference and minimizing cross interference. They report good results for direct-mapped caches.

In addition to loop transformations, compilers can also limit the effects of interference by modifying the storage layout of arrays. Rivera and Tseng [19] extend the Euclidean algorithm for tile-size selection and use array padding to help minimize interference. Ghosh, *et al.* [9], introduce cache-miss equations for precise analysis of perfectly nested loops. They use the cache-miss equations to guide tiling and padding to eliminate conflict misses. Vera, *et al.* [23], use cache-miss equations with a more accurate cache cost model and apply genetic algorithms to find the tile and padding factors. Kandemir, *et al.* [10], use loop transformations and data transformations such as array transpose to improve locality.

Each of the previous approaches uses array bounds information and possibly changes the storage layout of arrays in memory. Unfortunately, array bounds information may not be available until run-time. Additionally, modifying the storage layout in languages with pointer arithmetic (like C) or in languages that allow array reshaping on function calls and equivalences (like Fortran) may not always be legal. Our approach tries to improve locality and limit cross interference without storage modification or knowledge of the array bounds.

Alongside the compiler based techniques outlined above, there are a number of hardware techniques which aim to reduce general interference in the cache. Most hardware techniques center around the idea of using better mapping functions instead of the simple indexing schemes commonly used in contemporary cache designs. Yang, *et al.*[25] propose a prime indexed cache for vector processors. Using special properties of Mersenne prime numbers and taking into account the property of vector accesses, it is possible to compute the cache index with a simple adder as part of the address computation. Kharbutli, *et al.* [14], propose the use of prime numbers for cache indexing. Their techniques mainly target level two caches, as the proposed indexing functions would lengthen the critical path of the processor and would cause fragmentation. Topham, *et al.* [22], use polynomial modulo functions to calculate the index and discuss how to perform the computation using *xor* functions. Their technique is orthogonal to the techniques proposed in this paper and can be used to reduce self-interference.

Similar to the above approaches that use different indexing functions, various approaches employ *skewing* functions to minimize conflict misses for associative caches [2]. Skewed associativity also targets level-2 caches and is orthogonal to our approach.

3. COMPUTING WORKING SETS

In this section, we begin with a review of the background related to loop tiling and cache optimization. Then, we discuss our analysis for tiling and exposing working sets.

3.1 Data-Reuse Analysis

The two sources of data reuse are *temporal* reuse, multiple accesses to the same memory location, and *spatial* reuse, accesses to nearby memory locations that share a cache line or a block of memory at some level of the memory hierarchy. Temporal and spatial reuse may result from *self-reuse* from a single array reference or *group-reuse* from multiple references.[1] In this paper we use the method developed by Wolf and Lam [24] to determine the reuse properties of loads. An overview of their method follows.

A loop nest of depth n corresponds to a finite convex polyhedron Z^n, called an iteration space, bounded by the loop bounds. Each iteration in the loop corresponds to a node in the polyhedron, and is identified by its index vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, where x_i is the loop index of the i^{th} loop in the nest, counting from the outermost to the innermost. The iterations that can exploit reuse are called the localized iteration space, L. The localized iteration space can be characterized as a localized vector space if the loop bounds are abstracted away.

In Wolf and Lam's model, data reuse can only exist between *uniformly generated* references as defined below [6].

DEFINITION 1. *Let n be the depth of a loop nest, and d be the dimensions of an array* A. *Two references* $\mathsf{A}(f(\vec{x}))$ *and* $\mathsf{A}(g(\vec{x}))$,

[1]Without loss of generality, we assume Fortran's column-major storage.

where f and g are indexing functions $Z^n \rightarrow Z^d$, are uniformly generated *if*

$$f(\vec{x}) = H\vec{x} + \vec{c}_f \text{ and } g(\vec{x}) = H\vec{x} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

References in a loop nest are partitioned into different sets, each of which operates on the same array and has the same H. These sets are called *uniformly generated sets* (UGSs).

A reference is said to have *self-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{0}$. The solutions to this equation give the self-temporal reuse vector space R_{ST}. A reference has *self-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{0}$, where H_S is H with the first row set to $\vec{0}$. The solutions to this equation give the self-spatial reuse vector space R_{SS}. Two distinct references in a UGS, $\mathtt{A}(H\vec{x} + \vec{c}_1)$ and $\mathtt{A}(H\vec{x} + \vec{c}_2)$ have *group-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{c}_1 - \vec{c}_2$. And finally, two references have *group-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{c}_{1,S} - \vec{c}_{2,S}$.

References can be partitioned into *group-temporal sets* (GTS) and *group-spatial sets* (GSS) based upon solving the above equations. Since group-temporal reuse is a special case of group-spatial reuse, a GSS can contain many GTSs.

3.2 Loop Tiling

When the working-set size of the data accessed within a loop is larger than the size of the cache, compilers have traditionally used loop transformations to reduce the cache capacity requirements of the loop. Transformations such as loop permutation [17, 24], loop fusion [7, 13, 12, 20] and loop tiling [5, 11, 15, 21, 24] can be used to move references to the same memory location closer together to improve cache performance. While all of these transformations are important, loop tiling is especially important since it is used to build working sets small enough to fit in cache. As an example of how a compiler uses loop tiling to reduce the working set, consider the following vector-matrix multiply example.

```
do j = 1, n
  do i = 1, n
    y(i) = y(i) + x(j) * m(i,j)
  enddo
enddo
```

In this loop, the working set in the innermost loop consists of the entire array `y`, a single element of the array `x` and an entire column of the array `m`. If this working set is too large for the cache to handle, the loop can be tiled as follows.

```
do j = 1, n, is
  do i = 1, n ,js
    do jj = i, min(i+is-1,n)
      do ii = j, min(j+js-1,n)
        y(ii) = y(ii) + x(jj) * m(ii,jj)
      enddo
    enddo
  enddo
enddo
```

Here `is` and `js` are chosen so that the entire data set accessed in the inner two loops is kept in the cache and so that conflicts are minimized [5, 11, 21, 24]. The shaded regions in Figure 2 graphically depict the regions of `y`, `x` and `m` accessed in the inner two loops.

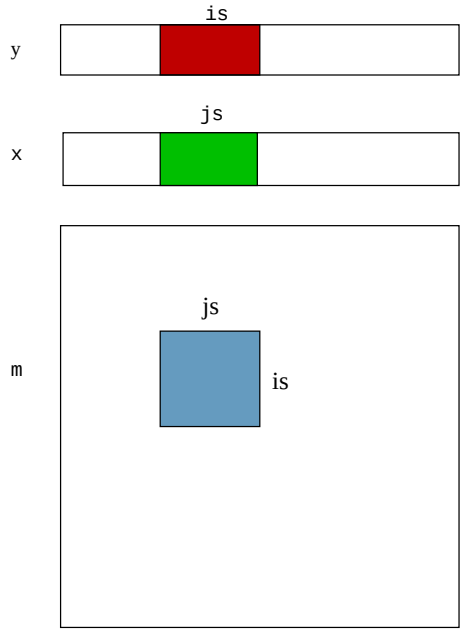


Figure 2: Tiled Arrays

Compilers typically choose tiling factors based upon the size of the array bounds so that interference in the cache does not occur [15, 5, 23]. Unfortunately, array sizes may not be known until run-time, or in the case of library code, may not be available and change depending upon the context used. To deal with interference between array references, array padding is often used [18, 23]. Padding modifies the storage layout of arrays by adding dummy locations to the rows or columns of an array or by adding dummy locations between different arrays. Padding is effective, but may not be legal in cases where pointer arithmetic or array reshaping and equivalences are used.

3.3 Working-set Analysis

To address the problem with dependence on array bounds and storage modification for loop tiling, we utilize a compiler/hardware approach that identifies the working sets in a loop in between which cross interference may occur. The compiler identifies the working sets in a loop and passes this information on to the cache controller via a working-set color. The micro-architecture then utilizes this information to limit cross interference. In this section, we discuss how the compiler identifies the working sets, chooses the tile sizes for a loop and generates the final code exposing the working-set-color information to the cache controller.

The first step of our algorithm determines which loops can be tiled. We set the localized iteration space to be those loops that can be legally tiled, and then compute the group-spatial sets within the localized iteration space per the algorithm of Wolf and Lam. After computing the group-spatial sets, we choose to tile a loop if

1. Some other loop in the localized iteration space carries self-temporal reuse, or
2. An outer loop carries group-spatial reuse.

The second step of our algorithm computes the working sets of the loop and colors each working set with a unique color. Within each group-spatial set, we compute the working sets as follows:

Two references are in the same working set if their address computations differ by a simple constant offset and can be allocated the same address register. For Fortran (C), this encompasses those references that access the same column (row) of an array. Each working set is assigned a unique color, although working sets that have no self-temporal or group-spatial reuse may be assigned the same color since tiling will not capture reuse in the cache. As an example of our technique, consider the following loop.

```
do j = 1, n-1
  do i = 1, n
    b(i,j) = a(i,j) + a(i,j+1)
  enddo
enddo
```

Our analysis finds three working sets: `b(i,j)` receiving color 1, `a(i,j)` receiving color 2, and `a(i,j+1)` receiving color 3.

Once the working sets have been computed, we compute the tile sizes for those loops chosen for tiling based upon the cache footprints of the working sets. We use the algorithm in Figure 3 to compute the tile sizes. This algorithm allows tiling of up to two dimensions for each group-spatial set as tiling more than two dimensions is not likely beneficial. Tile choices are computed on a per group-spatial set basis since every working set within a group-spatial set has the same cache footprint size.

```
for each tiled dimension i
    tileFactor[i] = MAXINT
P_B = cache size / number of working sets // bytes per partition
P_L = P_B / cache line size // lines per partition
for each group-spatial set g {
    P_E = P_L * elements per line for g // Array elements per partition
    if g has self-temporal or group-spatial reuse {
        if g has self-temporal reuse and ∃ 2 dimensions
                without self-temporal reuse {
            T =a square tile factor for g based on P_B
            for each dimension i | g is not self-temporal wrt i
                tileFactor[i] = min(tileFactor[i],T)
        }
        else {
            for each tiled dimension i
                if g does not have self-temporal reuse wrt i
                    if g has self-spatial reuse wrt i
                        tileFactor[i] = min(tileFactor[i],P_E)
                    else
                        tileFactor[i] = min(tileFactor[i],P_L)
        }
    }
}
```

Figure 3: Tile-size Selection

For each working set, we allocate a portion of the cache based upon the number of distinct working-set colors in the loop. This is denoted as P_B in the algorithm in Figure 3. Note that as discussed in Section 4 if the cache is physically partitioned for colors, P_B could be set based upon the physical partition size. For those working-sets not having self-temporal or group-spatial reuse (within or across working sets), we assign them to the same partition. Currently, we give each working-set color an equal portion of the cache. Future work will address prioritizing working sets and partitioning the cache space in a heterogeneous fashion.

For each group-spatial set *g*, we set the tile size for each tiled loop based upon the number of elements of the array that will fit in a partition. If the array has a one-dimensional tile, we set that limit based upon the number of cache lines in the partition. If *g* has self-spatial reuse with respect to the tiled loop, then the tile size accounts for the ability to use the entire cache line. If *g* does not have self-spatial reuse, then we set the tile size to the number of lines in a partition. For two-dimensional tiles, we compute the largest square tile size that will fit in the partition.

In the example loop, tiling alone does not capture the reuse within a group-spatial set when a set contains multiple working sets (*e.g.*, between `a(i,j)` and `a(i,j+1)`). When a single group-spatial set contains multiple working sets, we apply loop unrolling to the loop across which the group-spatial reuse occurs, if that loop is the column index (row index for C) for the array, and rotate the working-set colors in a modulo fashion to capture the spatial reuse [3]. This technique is similar to tiling for vector-register allocation [1].

Assuming that the `j`-loop has an even number of iterations in the previous example, we would generate the following code after unrolling the `j`-loop by a factor of 2:

```
do ii = 1, n, is
  do j = 1, n-1, 2
    do i = ii, min(ii+is-1,n)
      b(i,j) = a(i,j) + a(i,j+1)
    enddo
    do i = ii, min(ii+is-1,n)
      b(i,j+1) = a(i,j+1) + a(i,j+2)
    enddo
  enddo
enddo
```

The load of `a(i,j+1)` in the second loop receives working-set color 3 and the load of `a(i,j+2)` receives working-set color 2. This additional color assignment captures the spatial reuse between the two references to `a(i,j+1)` and the spatial reuse between the reference to `a(i,j+2)` on iteration `j` and the reference to `a(i,j)` on iteration `j+2`.

The unroll factor of a loop is chosen by computing the distance in the column dimension (row for C) between the first and last element of a group-spatial set containing more than one working set. The unroll factor is the maximum such distance over all group-spatial sets plus one. In the example, the maximum distance is one, giving an unroll factor of two.

One key difference in our approach to tiling and coping with interference as compared to previous approaches is that we choose to ignore the array bounds when choosing a tile size. Since the array size is not always available until run-time, our approach deals with interference by designing a cache that uses the working-set color of a reference to limit the effects of cross interference. By dealing with interference at run-time, we utilize a more general solution that applies whether or not the compiler has complete information.

4. A COLOR CACHE

Key issues in the design of a working-set based memory hierarchy include the means of communication of set information from the compiler to the micro-architecture and how the working-set information is utilized by the micro-architecture to achieve the desired goals. As previously stated, we choose to communicate the working-set information by encoding it as a simple integer as part of the load or store operation accessing the memory hierarchy (Figure 4). The working-set number is used to locate a data item in the cache together with the memory address. Similar to the tag array

in a conventional cache, the accessed entry must be compared with the working-set color of the accessing memory operation. Therefore each cache row is extended to include the working-set color of the cache row.

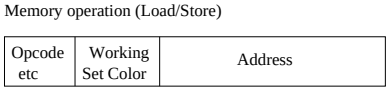


Figure 4: Colored Load / Store Instruction.

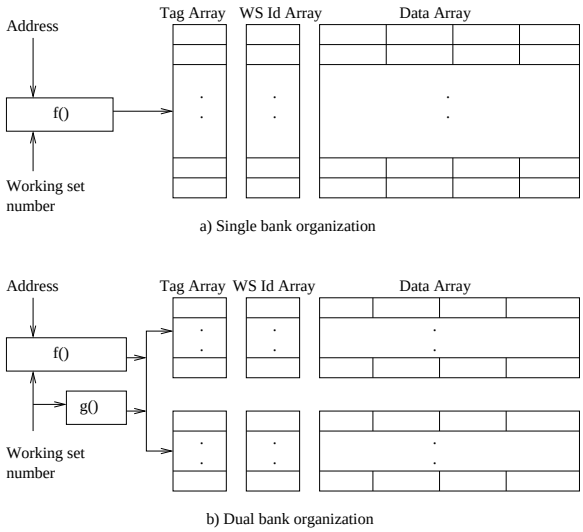


Figure 5: Cache organizations

4.1 Mapping Working Sets and Parallelism

There are many ways that the working-set color can be useful to the micro-architecture for optimizing the behavior of the memory hierarchy. For a given amount of chip real-estate, the architecture may choose to implement a single cache bank for reduced miss rates, or in order to achieve a higher clock rate and increased amount of memory parallelism, it may choose to use multiple independently operating cache banks (Figure 5 (a) and (b)). Despite these variations, the key problem from the micro-architecture perspective is how to map a given memory address and a working-set color to an independent bank (i.e., g() in Figure 5) and to a cache row within that bank (i.e., f() in Figure 5). In this paper, we choose to use the simple function *working-set color* mod *number of banks* to choose the bank. This function places the working sets into different banks in a round-robin fashion. For cache row selection, we simply follow the traditional indexing function to compute the row index and add the working-set color to the computed index. Use of *xor* in place of addition yields similar results.

Our aim in choosing the above mapping functions is to use simple functions that will not hamper the access time of the cache, but will minimize the cross-interference among sets. Although we could allocate each working set to a distinct partition within a bank by concatenating the working-set color to the cache row index, our experiments indicate that it is better to have the freedom to balance the amount of cross-interference to that of self-interference. As we have shown in the previous sections, by allocating each working set to different colors the compiler removes most cross-interference from the program. On the other hand, by carefully selecting our mapping function, the hardware can balance the amount of cross interference among sets to that of self interference by allowing each working set to use a larger portion of the cache bank.

4.2 Issues of Aliasing and Coherence

Since the working-set color is part of the addressing scheme, the correct working-set color must be presented together with each memory access. Although the compiler may allocate working-set colors naturally for the part of the code that can be analyzed fully, it is possible that the same data element may be accessed with two different working-set colors. As a result, the data in the cache would not be coherent. Our solution to this problem is to make sure that the data stored in the cache is always stored under the same working-set color. In order to guarantee this property, we probe the cache using the same address and all the available working sets upon a cache miss. If the data is found to be under another working-set color it is moved to the new working-set color under which it was sought in the first place. In other words, to handle such a rare case of aliasing correctly, cache misses are handled by concurrently starting a probe of other cache locations as well as issuing a request to the lower level. If the data is found under another working-set color, the request issued to the lower level is canceled. Obviously, with such a scheme, during a cache miss further accesses to the cache may not be allowed. In order to alleviate the effects of probing, an aggressive superscalar processor may choose to replicate the tag arrays.

Assuming that we would like to finish probing the cache for other possible locations before the data is returned from the lower level, the largest number of sets that the compiler may employ at any given time is given by the equation $m \times t$, where m is the miss penalty and t is the number of tag arrays. For example, an architecture that has two banks and a 12-cycle miss penalty can accommodate 24 colors.

It should be clear that in a design that seeks to improve the memory parallelism, multiple memory operations may be steered into multiple load/store queues each being served by one cache bank. Since the working-set color is available as part of the load/store instruction, this information becomes available very early in the pipeline. As a result, we can steer the load/store instruction into a different queue, and hence to a different bank. As long as there are no cache misses, each cache bank operates independently. In contrast, a set-associative cache cannot provide the desired parallelism and incorporates a large multiplexer to select the proper bank after the bank accesses are complete.

5. EXPERIMENT

In this section, we first present our experimental methodology. Then, we discuss the performance of our working-set color cache approach on a set of computational kernels. Finally, we summarize the results of the experiment and relate it to previous work.

5.1 Methodology

We have implemented the method described in Section 3 in a source-to-source Fortran compiler and tested its effectiveness on a set of computational kernels that exhibit different characteristics. The source-to-source transformer annotates the Fortran source with working-set color information. We then run the annotated source through another compiler to generate assembly code with the appropriate color information.

Table 1 gives a short description of each kernel used in our experiment and the number of working-set colors required by the tiled version of the kernel. The kernels afold and fold exhibit large amounts of self-temporal and self-spatial reuse and have triangu-

Kernel	Description	Colors
afold	adjoint convolution of two time series	3
calc	kernel form 171.swim	8
dmxpy	vector-matrix product	3
dy3d6p	compute y derivatives	6
fold	convolution of two time series	3
jacobi	jacobi iteration	4
mm	matrix-matrix multiplication	3
sor	successive over-relaxation	3

Table 1: Benchmark Kernels

lar iteration spaces. Jacobi and sor have outer-loop group-spatial reuse and have multiple columns active at once. Jacobi uses two arrays while sor uses one. Calc requires the most working-set colors and has outer-loop group-spatial reuse. Dy3d6p contains both self-temporal reuse and outer-loop group-spatial reuse. Dy3d6p also uses three-dimensional arrays. Dmxpy and mm exhibit self-temporal reuse and mm uses only two-dimensional arrays. Together, these kernels cover a spectrum of reuse and test the method's ability to eliminate interference in the presence of higher dimensional arrays.

We ran the kernels in Table 1 over a range of 200 different arrays sizes on a micro-architecture simulator using a color cache as described in Section 4 and using a standard direct-mapped cache. We recorded the miss rates for each array size for each kernel on each cache organization. We compared the tiled version of the code without working-set colors with the tiled version with working-set colors. For the non-color version, we tiled based upon the footprint size of the arrays in the loop, irrespective of the array bounds. For the tiled code with working-set colors, we use the algorithm in Figure 3 to select the tile sizes. Figures 6, 7, and 8 detail the miss rates for both versions. Since the capacity misses are minimal in these tiled loops, the changes in miss rates represent the change in conflict misses. We used a standard 8K direct-mapped cache to test the non-color version. For the working-set color version we ran the tests on an 8K direct-mapped single module cache (1×8K Color) using the mapping function described in Section 4 and also an 8K cache split into two 4K modules (2×4K Color).

5.2 Performance of the Color Cache

Figures 6(a)–(c) show the miss rates for afold on the standard 8K, 1×8K color and 2×4K color caches, respectively. If we examine the figures we see that the 2×4K color cache gives the lowest miss rate and the standard 8K cache gives the highest miss rate. Afold uses three one-dimensional arrays that all have a self-temporal and self-spatial reuse. The 2×4K cache effectively eliminates the cross interference between the arrays and exhibits little self interference. The 1×8K color cache limits some of the cross interference, but not nearly as much as the 2×4K cache.

Figures 6(d)–(f) show that the color cache effectively limits the amount of cross interference for calc. Calc accesses seven different columns in the innermost loop. Using working-set colors, we effectively eliminate cross interference between the columns and smooth the performance considerably over the range of array sizes.

In Figures 6(g)–(i), the working-set color cache smooths and lowers the miss rate for dmxpy. The 1×8K color cache performs slightly better than the 2×4K color cache due to the existence of self interference. When self interference is present, the single 8K cache module gives a larger area for the working set to be held, reducing the amount of self collisions.

In dy3d6p, self interference causes most of the cache misses. As can be seen in Figures 7(a)–(c), the color cache reduces the misses and smooths the curve slightly, but a significant number of self-interference misses still occur. The 1×8K color cache smooths the curve slightly more than the 2×4K color cache because the single-module cache is less susceptible to self interference.

The graphs in Figure 7(d)–(i) show that fold and jacobi have similar results to afold and calc, respectively. Afold and fold are quite similar, but the iteration space shapes are different and different access sequences result. Jacobi uses fewer arrays than calc, but more columns within a single array are used, resulting in 4 working sets being discovered by our analysis. Since both of these kernels exhibit mostly cross interference and little self interference, the color cache limits misses effectively.

Finally, the graphs in Figure 8 show that current color cache approach is not completely effective. Both mm and sor mostly exhibit self interference. The 2×4K cache lowers the amplitude of some of the spikes in Figure 8(a), but because each module in the cache is only 4K, self interference is frequently a problem that remains untouched (or sometimes exacerbated). For sor, little or no cross interference exists, the 2×4K color cache introduces a little more self interference and the 1×8K cache performs as well as the 8K traditional cache.

5.3 Discussion

The results presented in Section 5.2 show that our technique performs well when cross interference is the dominant cause of cache misses in a loop nest. When self interference is not dominant, the 2×4K color cache removes the most misses. However, we do not yet adequately handle self interference. Our results do not demonstrate that our approach is superior to previous compiler techniques that use array bounds information in computing the tile size when the array bounds information is available. Those techniques are orthogonal to ours and may additionally be applied when bounds information is available and padding is legal. However, we have shown a new approach that shows promise when all information regarding arrays is not available.

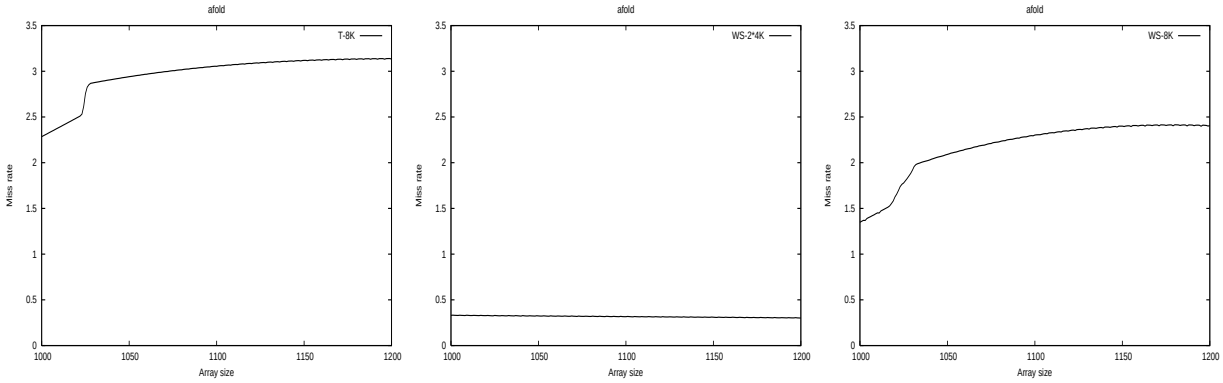
Our work in exploring the potential of the color cache is not complete. We have yet to consider the role of compile-time dead working-set identification in cache replacement decisions. In addition, we have not explored the use of prefetching for working-sets or alternate cache organizations that use compile-time working-set analysis to limit self interference.

6. CONCLUSIONS

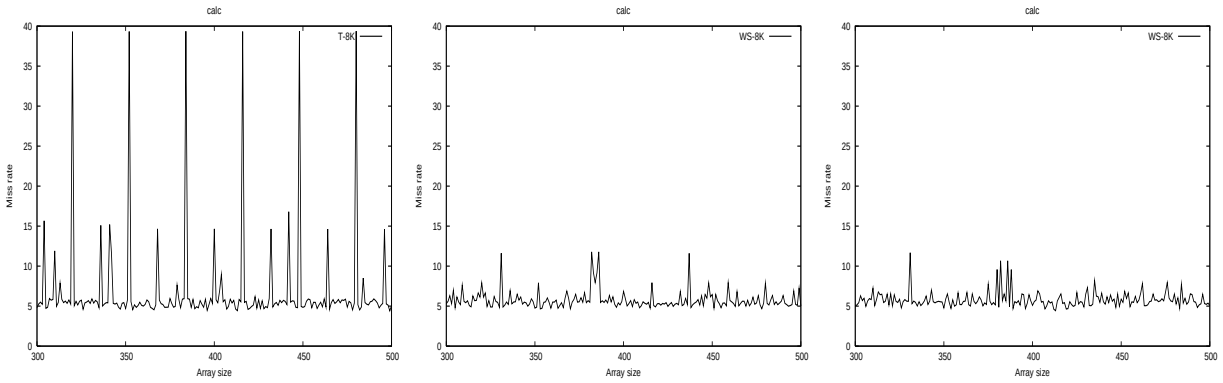
We have described a combined compiler/architecture approach based upon working sets to limit the effects of cross interference in the cache. We have shown that our technique effectively eliminates cross interference on a set of kernels with different reuse patterns. In many cases, our technique smooths the wide variability in cache performance found when array sizes that cause significant amounts of cross interference are used.

The main advantage of our technique over previous work is that it does not rely on knowledge of the bounds of arrays. This becomes especially important in library code and for dynamically allocated arrays since in these instances array bounds cannot be known at compile time. In addition, the increase in cache complexity to manage working sets is manageable.

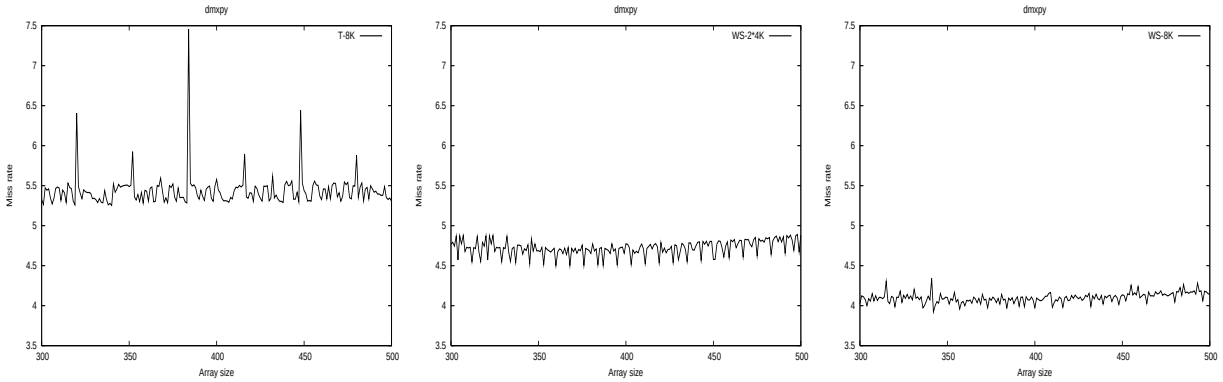
The next step of our work involves effectively dealing with self interference within the context of working sets. When self interference is the dominant form of interference, our techniques alone will not limit the interference as effectively. We continue to look at mapping techniques utilizing compile-time knowledge to limit the effects of self interference.



(a) afold 8K Traditional (b) afold 2×4K Color (c) afold 1×8K Color

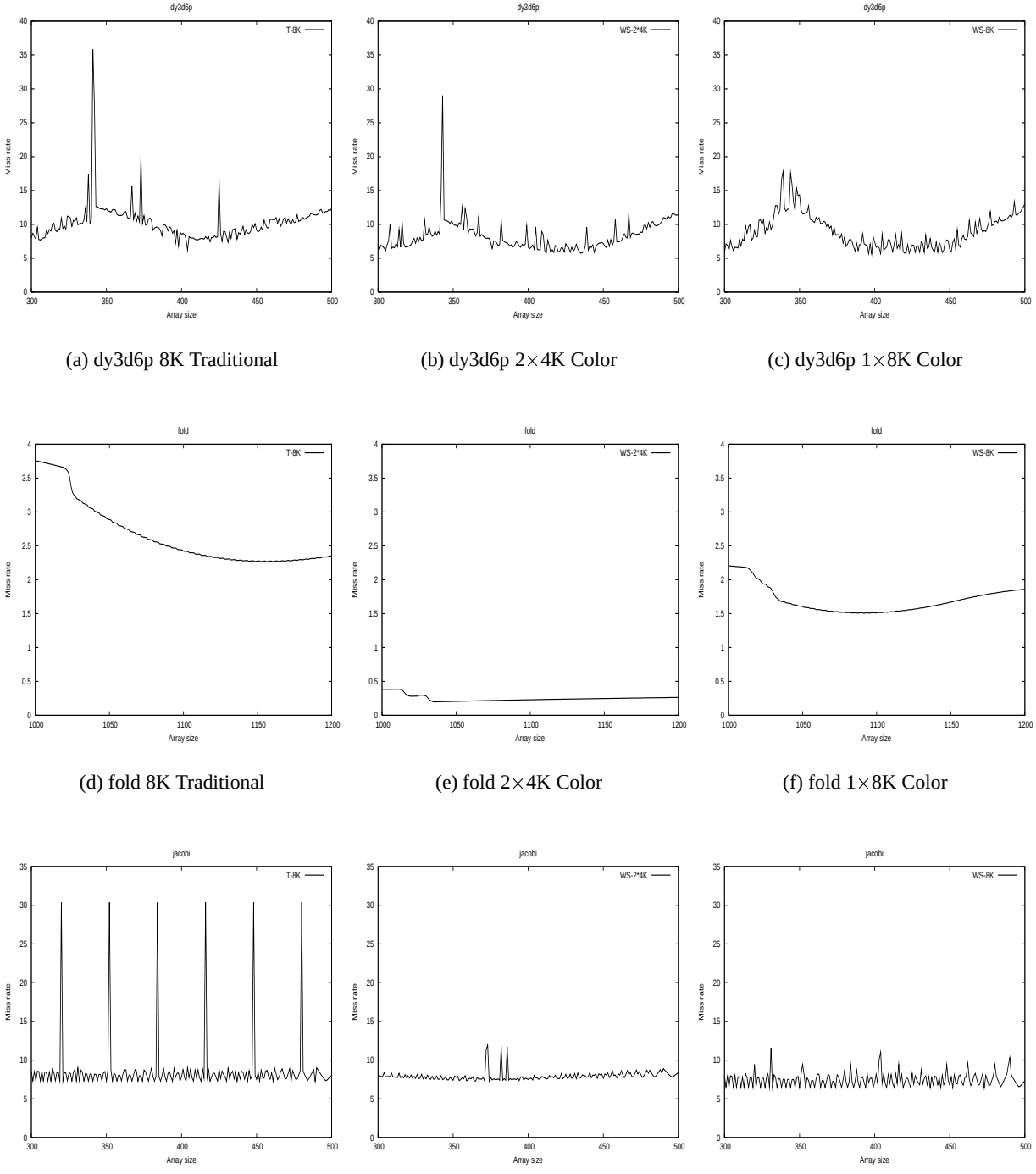


(d) calc 8K Traditional (e) calc 2×4K Color (f) calc 1×8K Color



(g) dmxpy 8K Traditional (h) dmxpy 2×4K Color (i) dmxpy 1×8K Color

Figure 6: Miss Rates for afold**,** calc **and** dmxpy

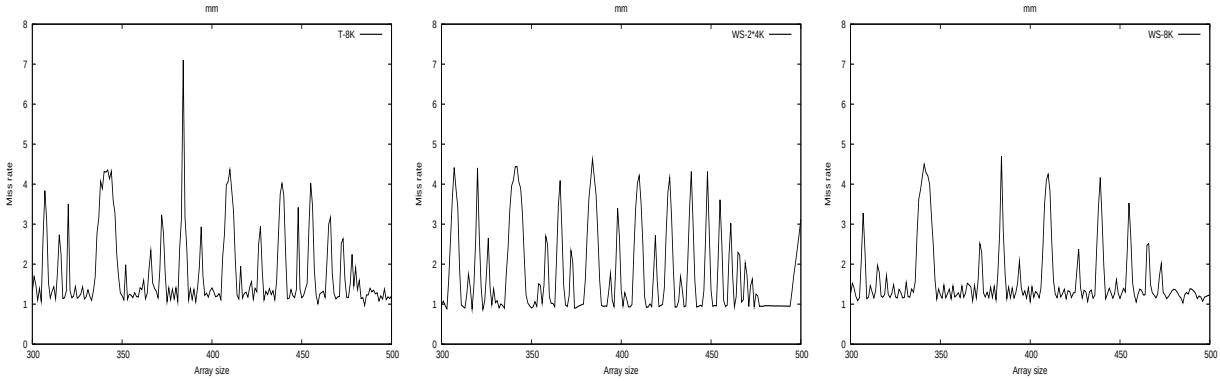


(a) dy3d6p 8K Traditional (b) dy3d6p 2×4K Color (c) dy3d6p 1×8K Color

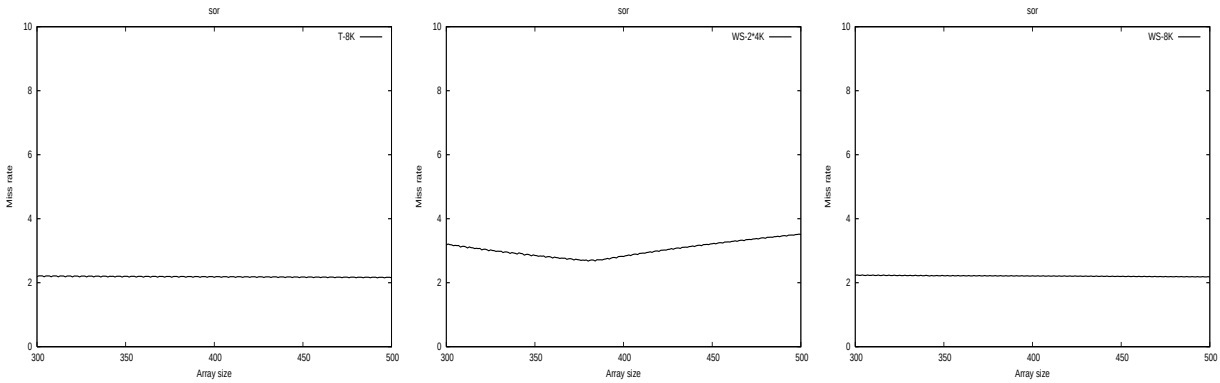
(d) fold 8K Traditional (e) fold 2×4K Color (f) fold 1×8K Color

(g) jacobi 8K Traditional (h) jacobi 2×4K Color (i) jacobi 1×8K Color

Figure 7: Miss Rates for dy3d6p**,** fold **and** jacobi



(a) mm 8K Traditional (b) mm 2×4K Color (c) mm 1×8K Color



(d) sor 8K Traditional (e) sor 2×4K Color (f) sor 1×8K Color

Figure 8: Miss Rates for mm **and** sor

In order for future memory systems to perform well, they must deal with cache interference. We believe that our approach is a positive first step in a comprehensive set of compile-time/run-time techniques to manage cache interference effectively.

Acknowledgments

This work was partially supported by U.S. National Science Foundation grant CCR-0312892.

7. REFERENCES

[1] J. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, Oct. 1992.

[2] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 265–274. ACM Press, 1995.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, June 1990.

[4] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software*, 23(3):336–361, Sept. 1997.

[5] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 279–280, La Jolla, CA, June 1995.

[6] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, 1987.

[7] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT USA, Aug. 1992.

[8] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.

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

[10] M. Kandemir, A. Choudary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 285–296, Dallas, TX, Dec. 1998.

[11] M. T. Kandemir, J. Ramanujam, and A. Choudary. A compiler algorithm for optimizing locality in loop nests. In *International Conference on Supercomputing*, pages 269–276, May 1997.

[12] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, May 2000.

[13] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compiler for Parallel Computing*, pages 301–321, Portland, OR USA, Aug. 1993.

[14] M. Kharbutli, K. Irwin, Y. Solohin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Tenth International Symposium on High-Performance Computer Architecture*, pages 288–299. IEEE Computer Society, 2004.

[15] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, 1991.

[16] K. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, MA, Oct. 1996.

[17] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

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

[19] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction*, Amsterdam, The Netherlands, Mar. 1999.

[20] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 194–205, June 1991.

[21] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, GA USA, May 1999.

[22] N. Topham, A. González, and J. González. The design and performance of a conflict-avoiding cache. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 71–80. IEEE Computer Society, 1997.

[23] X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through cmes and gas. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 68–78, New Orleans, LA, September 2003.

[24] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, June 1991.

[25] Q. Yang and L. W. Yang. A novel cache design for vector processing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 362–371. ACM Press, 1992.