Understanding Unfulfilled Memory Reuse Potential in Scientific Applications

Gabriel Marin and John Mellor-Crummey
{mgabi,johnmc}@cs.rice.edu
Department of Computer Science
6100 Main St., MS 132
Rice University
Houston, TX 77005

Technical Report TR07-6
Department of Computer Science
Rice University

Abstract

The potential for improving the performance of data-intensive scientific programs by enhancing data reuse in cache is substantial because CPUs are significantly faster than memory. Traditional performance tools typically collect or simulate cache miss counts or rates and attribute them at the function level. While such information identifies program scopes that suffer from poor data locality, it is often insufficient to diagnose the causes for poor data locality and to identify what program transformations would improve memory hierarchy utilization. This paper describes a memory reuse distance based approach that identifies an application's most significant memory access patterns causing cache misses and provides insight into ways of improving data reuse. We demonstrate the effectiveness of this analysis for two scientific codes: one for simulating neutron transport and a second for simulating turbulent transport in burning plasmas. Our tools pinpointed opportunities for enhancing data reuse. Using this feedback as a guide, we transformed the codes, reducing their misses at various levels of the memory hierarchy by integer factors and reducing their execution time by as much as 60% and 33%, respectively.

1 Introduction

The potential for improving the performance of data-intensive scientific programs by enhancing data reuse in cache is substantial because CPUs are significantly faster than memory. For data intensive applications, it is widely accepted that memory latency and bandwidth are the factors that most limit node performance on microprocessor-based systems.

Traditional performance tools typically collect or simulate cache miss counts and rates and attribute them at the function level. While such information identifies the program scopes that suffer from poor data locality, it is often insufficient to diagnose the causes for poor data locality and identify what program transformations would improve memory hierarchy utilization.

To understand why a particular loop experiences many cache misses, it helps to think of a non-compulsory cache miss as a reuse of data that has been accessed too far in the past to still be in cache. Memory reuse distance is an architecture independent metric that tells us the number of distinct memory blocks accessed by a program between pairs of accesses to the same block.

Over the years, memory reuse distance has been used by researchers for many purposes. These include investigating memory hierarchy management techniques [5, 14], characterizing data locality in program executions for individual program inputs [6, 9], and using memory reuse distance data from training runs to predict cache miss rate for other program inputs [10, 12, 17]. To understand if a memory access is a hit or miss in a fully-associative cache using LRU replacement, one can simply compare the distance to its previous use with the size of the cache. For set-associative caches, others have shown that a reuse distance based probabilistic model yields accurate predictions [13]. Previous work has focused on only part of the information that one can gather through memory reuse distance analysis. In particular, each data reuse can be thought of as an arc from one access to a block of data to the next access to that block. Associating reuse distance data with only one end of a reuse arc fails to capture the correlation between references that access the same data.

This paper describes techniques for collecting reuse distance information and associating it with individual reuse arcs. This work builds on previous techniques developed by Marin and Mellor-Crummey [13]. Unlike prior methods, this new approach provides insight into the causes for poor data locality. Section 2 presents our strategy for collecting and processing reuse distance information. Section 3 describes a different type of analysis for understanding the fragmentation of data in cache lines. Section 4 describes how to use this information to improve data reuse. Section 5 describes the process of tuning two scientific applications using our approach. Section 6 describes the closest related work. Section 7 presents our conclusions and plans for future work.

2 Reuse-distance extensions

Prior algorithms for computing memory reuse distance [9, 12] store a few bits of information about each memory block accessed. By extending the information for a block to include the identity of the most recent access, we can associate a reuse distance with a (*source*, *destination*) pair of scopes where the two endpoints of the reuse arc reside.

This approach enables us to collect reuse distance histograms separately for each pair of scopes that reuse the same data. Once reuse distance histograms are translated into cache miss predictions for a target cache architecture, this approach enables us to understand not only where we experience a large fraction of cache misses, but also where that data has been previously accessed before it was evicted from cache. If we can transform the program to bring the two accesses closer, for example by fusing their source and destination loops, we may be able to shorten the reuse distance so that the data can be reused before it is evicted from cache.

We found that in many cases a single scope was both the source and the destination of a reuse arc. While this provides the insight that data is accessed repeatedly in the same loop without being touched in a different program scope, we found such information insufficient to understand how to correct the problem. What was missing, was a way to tell which outer loop was carrying the reuse, or in other words, which loop was causing the program to access the same data again on different iterations. If we know which loop is causing the reuse and if the distance of that reuse is too large for our cache size, then it may be possible to shorten the reuse distance by either interchanging the loop carrying the reuse inwards, or by blocking the loop inside it and moving the resulting loop that iterates over blocks, outside the loop carrying the reuse.

Loop interchange and blocking are well studied compiler transformations. A more thorough

```
DO I = 1, N
   DO J = 1, M
      A(I,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```
(a)

```
DO J = 1, M
   DO I = 1, N
      A(I,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```
(b)

Figure 1: (a) Example of data reuse carried by an outer loop; (b) transformed example using loop interchange.

discussion of these transformations can be found in [3]. Figure 1(a) presents a simple loop nest written in Fortran. Although Fortran stores arrays in column major order, the inner loop here iterates over rows. There is no reuse carried by the `J` loop, since each element of a row is in a different cache line. However, for non-unit size cache lines, there is spatial reuse carried by the outer `I` loop. By interchanging the loops as shown in Figure 1(b), we move the loop carrying spatial reuse inwards, which reduces the reuse distance for the accesses.

While it is reasonably easy to understand reuse patterns for simple loops, for complex applications understanding reuse is a daunting task. To capture the carrying scope of a reuse automatically, we extended the reuse distance data collection infrastructure of Marin and Mellor-Crummey presented in [12] in several ways.

- We add instrumentation to monitor entry and exit of routines and loops. For loops, we instrument only loop entry and exit edges in the control flow so that instrumentation code is not executed on every iteration.
- We maintain a logical *access clock* that is incremented at each memory access.
- We maintain a dynamic stack of scopes in the shared library. When a scope is entered, we push a record containing the scope id and the value of the access clock onto the stack. On exit, we pop the entry off the scope stack. The stack stores the active routines and loops, in the order in which they were entered.
- On a memory access, in addition to the steps presented in [12], we traverse the dynamic stack of scopes starting from the top, looking for $\mathcal{S}$—the shallowest entry whose access clock is less than the access clock value associated with the previous access to current memory block. Because the access clock is incremented on each memory access, $\mathcal{S}$ is the most recent active scope that was entered before our previous access to current memory block. $\mathcal{S}$ is the least common ancestor in the dynamic scope stack for both ends of the reuse arc and we say it is the carrying scope of the reuse.
- For a reference, we collect separate histograms of reuse distances for each combination of (*source scope*, *carrying scope*) of the reuse arcs for which the reference is the sink.

Compared to Marin and Mellor-Crummey's approach, our extensions increase the resolution at which memory reuse distance data is collected. For one reference, we store multiple reuse distance histograms, one for each distinct combination of source scope and carrying scope of the reuse arcs. In practice, the additional space needed to maintain this information is reasonable and well worth it for the additional insight it provides. First, during execution applications access data in some well defined patterns. A load or store instruction is associated with a program variable that is accessed in a finite number of scopes that are executed in a pre-determined order. Thus, there is not an explosion in the number of histograms collected for each reference. Second, reuse distances seen by an instruction at run-time vary depending on the source and carrying scopes of the reuse arcs. As a result, our implementation maintains more but smaller histograms.

All reuse distance data we collect can still be modeled using the algorithm presented in [13] to predict the distribution of reuse distances for other program inputs. In addition, since patterns are collected and modeled at a finer resolution, the resulting models are more accurate for regular applications.

Our new data enable us to compute cache miss predictions for an architecture separately for each reuse pattern. Thus, when we investigate performance bottlenecks due to poor data locality, we can highlight the principal reuse patterns that contribute to cache misses and suggest a set of possible transformations that would improve reuse. Not only does this information provide insight into transformations that might improve a particular reuse pattern, but it also can pinpoint cache misses that are caused by reuse patterns intrinsic to an application, such as reuse of data across different time steps of an algorithm or reuse across function calls, which would require global code transformations to improve.

We compute several metrics based on our memory reuse analysis. For each scope, we compute traditional cache miss information; we use this to identify loops responsible for a high fraction of cache misses. However, we also associate cache miss counts with the scope that accessed data last before it was evicted, with the scope that is carrying these long data reuses, and a combination of these two factors. To guide tuning, we also compute the number of cache misses carried by each scope. We break down carried miss counts by the source or/and destination scopes of the reuse. These final metrics pinpoint opportunities for loop fusion and provide insight into reuse patterns that are difficult or impossible to eliminate, such as reuse across time steps or function calls. To focus tuning efforts effectively, it is important to know which cache misses can be potentially eliminated and which cannot; this helps focus tuning on cases that can provide a big payoff relative to the tuning effort. In section 5, we describe how we use this information to guide the tuning of two scientific applications.

3 Understanding fragmentation in cache lines

The previous section described techniques for identifying opportunities to improve memory hierarchy utilization by shortening temporal and spatial distance. This section describes a strategy for diagnosing poor spatial locality caused by data layout. Caches are organized as blocks (lines) that typically contain multiple words. The benefit of using non-unit size cache lines is that when any word of a block is accessed, the whole block is loaded into the cache and further accesses to any word in the block will hit in cache until the block is evicted. Once a block has been fetched into cache, having accesses to other words in the block hit in cache is called spatial reuse. To exploit spatial reuse, we need to pack data that is accessed together into the same block.

We call the fraction of data in a memory block that is not accessed the *fragmentation factor*. We compute fragmentation factors for each reference and each loop nest in the program. To identify where fragmentation occurs, we use static analysis. First, we compute symbolic formulas that describe the memory locations accessed by each reference. We compute a symbolic formula for

```
DO I = 1, N, 4
   DO J = 1, M
      A(I+1,J,K) = A(I,J,K) + B(I,J) - B(I+1,J)
      A(I+3,J,L) = A(I+2,J,L) + B(I+2,J) - B(I+3,J)
   ENDDO
ENDDO
```

Figure 2: Cache line fragmentation example.

the *first location* a reference accesses by tracing back along use-def chains in it's enclosing routine. Tracing starts from the registers used in the reference's address computation. For references inside loops, we also compute symbolic formulas describing how the accessed location changes from one iteration to the next. Stride formulas have two additional flags. The first flag indicates whether a reference's stride is irregular (*i.e.*, the stride changes between iterations). The second flag indicates whether the reference is indirect with respect to that loop (*i.e.*, the location accessed depends on a value loaded by another reference that has a non-zero stride with respect to that loop). Second, we recover the names of data objects accessed by each reference using our symbolic formulas in conjunction with information recorded by the compiler in the executable's symbol table.

We say that references in a loop that access data with the same name and the same symbolic stride are *related references*. To analyze the fragmentation of data in cache lines, we work on groups of related references and use the static symbolic formulas described above to compute *reuse groups* and the *fragmentation factor* of each reuse group. Computing the fragmentation factor for each group of references[1], consists of a three step process.

Step 1. Find the enclosing loop for which this group of references experiences the smallest non-zero constant stride. When the reference group is enclosed in a loop nest, traverse the loops from the inside out, and terminate the search if a loop is encountered for which the references have an irregular stride. If a loop with a constant non-zero stride is not present, then we do not compute any *fragmentation factor* for that group of references. Otherwise, let s be the smallest constant stride that we find and go to **step 2**.

For the Fortran loop shown in Figure 2, the arrays are in column-major order, all four accesses to `A` are part of a single group of related references, and all four accesses to `B` are part of a second group of related references. For both groups, the loop with the smallest non-zero constant stride is the outer loop `I`, and the stride is 32 if we assume that the elements of the two arrays are double precision floating point values.

Step 2. Split a group of related references into reuse groups based on their *first location* symbolic formulas. Let F_1 and F_2 be the formulas describing the *first location* accessed by two references of a group. As computed in **step 1**, their smallest non-zero constant stride is s. If the two *first location* formulas differ only by a constant value, we compute how many iterations it would take for one formula to access a location within s bytes of the *first location* accessed by the other formula. If the necessary number of iterations is less than the average number of iterations executed by that loop (identified using data from the dynamic analysis), then the two references are part of the same *reuse group*. Otherwise, the two references are part of distinct *reuse groups*.

[1]Note that all references in a group have equal strides with respect to all enclosing loops. It suffices to consider the strides of only one reference in the group during analysis.

For our example in Figure 2, the group of references to array `A` is split into two *reuse groups*. One *reuse group* contains references `A(I,J,K)` and `A(I+1,J,K)`, and the second *reuse group* contains references `A(I+2,J,L)` and `A(I+3,J,L)`. The four references have been separated into two *reuse groups* because they access memory locations far apart, due to different indices in their third dimension. In contrast, all four references to array `B` are part of a single *reuse group*.

Step 3. Compute the hot foot-print information for each *reuse group* derived from a group of related references. For this, we use modular arithmetic to map the locations accessed by all references of a *reuse group* to a block of memory of size s, effectively computing the coverage of the block, *i.e.*, the number of distinct bytes accessed in the block. For a group of related references we select the maximum coverage, c, over all its *reuse groups*, and the fragmentation factor is $f = 1 - c/s$.

Returning to our example, both *reuse groups* corresponding to the set of references to array `A` have a coverage of 16 bytes, and thus the fragmentation factor for array `A` is 0.5. The single *reuse group* for the four references to array `B` has coverage 32, and thus a fragmentation factor of 0.

While it is possible to have non-unit stride accesses to arrays of simple data types, the main causes of data fragmentation are arrays of records, where only some record fields are accessed in a particular loop. The problem can be solved by replacing an array of records with a collection of arrays, one array for each individual record field. A loop working with only a few fields of the original record needs to load into cache only the arrays corresponding to those fields. If the original loop was incurring cache misses, this transformation will reduce the number of misses, which will reduce both the data bandwidth and memory delays for the loop. This transformation has the secondary effect of increasing the number of parallel data streams in loops that work with multiple record fields. While additional streams can improve performance by increasing memory parallelism [15], they can hurt performance on architectures with small TLBs and architectures that use hardware prefetching but can handle only a limited number of data streams.

For our example in Figure 2, and based only on the code in that loop, array `A` is better written as two separate arrays, each containing every other group of two elements of its inner dimension.

Using the fragmentation factors derived for each group of related references, we compute metrics which specify how many cache misses at each memory level are due to fragmentation effects. The number of cache misses due to cache line fragmentation is computed separately for each memory reuse pattern; we report this information at the level of individual loops and data arrays. Similarly, we compute the number of cache misses due to irregular reuse patterns. A reuse pattern is considered irregular if its carrying scope produces an irregular or indirect symbolic stride, as explained above, for the references at its destination end.

4 Interpreting the performance data

To identify performance problems and opportunities for tuning, we output all metrics described in the previous sections in XML format that we browse with a custom Java top-down viewer which enables us to sort the data by any metric and to associate metrics with the program source code. We can visualize both the exclusive values of the metrics and the inclusive values aggregated at each level of the program scope tree. We can browse the data in a top-down fashion to find regions of code that account for a significant fraction of a given performance metric (*e.g.,* misses, fragmentation, etc.), or we can compare the exclusive values across all scopes of the program.

Scenario	Transformations & comments
large number of fragmentation misses due to one array	data transformation: split the original array into multiple arrays
large number of irregular misses and $S \equiv D$	apply data or computation reordering
large number of misses and $S \equiv D$, C is an outer loop of same loop nest	carrying scope iterates over the array's inner dimension; apply loop interchange or dimension interchange on the affected array; if multiple arrays with different dimension orderings, loop blocking may work best
$S \not\equiv D$, C is inside same routine as S and D	fuse S and D
as the previous case, but S or D are in a different routine invoked from C	strip-mine S and D with the same stripe and promote the loops over stripes outside of C, fusing them in the process
C is a time step loop or a main loop of the program	apply time skewing if possible; alternatively, do not focus on these hard or impossible to remove misses

Table 1: Recommended transformations for improving memory reuse.

Not all metrics can be sensibly aggregated based on the static program scope tree. For example, aggregating the number of misses carried by scopes based on their static program hierarchy is meaningless. The carried number of misses is rather a measure representative of the dynamic tree of scopes observed at run-time. This information could be presented hierarchically along the edges of a calling context tree [4] that includes also loop scopes. A reuse pattern already specifies the source, the destination and the carrying scopes of a reuse arc; aggregating the number of misses carried by scopes does not seem to provide any additional insight into reuse patterns. While for some applications the distribution of reuse distances corresponding to a reuse pattern may be different depending on the calling context, for most scientific programs separating the data based on the calling context may dilute the significance of some important reuse patterns. At this point we do not collect data about the memory reuse patterns separately for each context tree node to avoid the additional complexity and run-time overhead. If needed, the data collection infrastructure can be extended to include calling context as well.

Since we collect information about the reuse patterns in an application, we generate also a database in which we can compare reuse patterns directly. This is a flat database in which the entries correspond not to individual program scopes, but to pairs of scopes that represent the source and destination scopes of reuse patterns. Its purpose is to quickly identify the reuse patterns contributing the greatest number of cache misses at each memory level.

Identifying reuse patterns with poor data locality is only part of the work, albeit a very important part. We need to understand what code transformations work best in each situation. Table 1 summarizes recommended transformations for improving memory reuse, based on the type of reuse pattern that is producing cache misses. We use S, D and C to denote the source, the destination and the carrying scopes of a reuse pattern. These recommendations are just that, general guidelines to use in each situation. Determining whether a transformation is legal is left for the application developer. In some instances, enabling transformations such as loop skewing or loop alignment may be necessary before we can apply the transformations listed in table 1.

5 Case Studies

In this section, we briefly illustrate how to analyze and tune an application using these new performance metrics. We describe the tuning of two scientific applications. Sweep3D [2] is a 3D Cartesian geometry neutron transport code benchmark from the DOE's Accelerated Strategic Computing Initiative. As a procurement benchmark, this code has been carefully tuned already. The Gyrokinetic Toroidal Code (GTC) [11] is a particle-in-cell code that simulates turbulent transport of particles and energy. We compiled the codes on a Sun UltraSPARC-II system using the Sun WorkShop 6 update 2 FORTRAN 77 5.3 compiler, using the flags *-xarch=v8plus -xO4 -depend -dalign -xtypemap=real:64*. We collected extended reuse distance information for each application.

5.1 Analysis and tuning of Sweep3D

Sweep3D performs a series of diagonal sweeps across a 3D Cartesian mesh, which is distributed across the processors of a parallel job. Figure 3(a) presents a schematic diagram of the computational kernel of Sweep3D. The `idiag` loop is the main computational loop on each node. It performs a sweep from one corner of the local mesh to the opposing corner. In each iteration of the `idiag` loop, one diagonal plane of cells is processed by the `jkm` loop. Before and after the `idiag` loop there is MPI communication to exchange data with the neighboring processors. Finally, the outer `iq` loop iterates over all octants, starting a sweep from each corner of the global mesh.

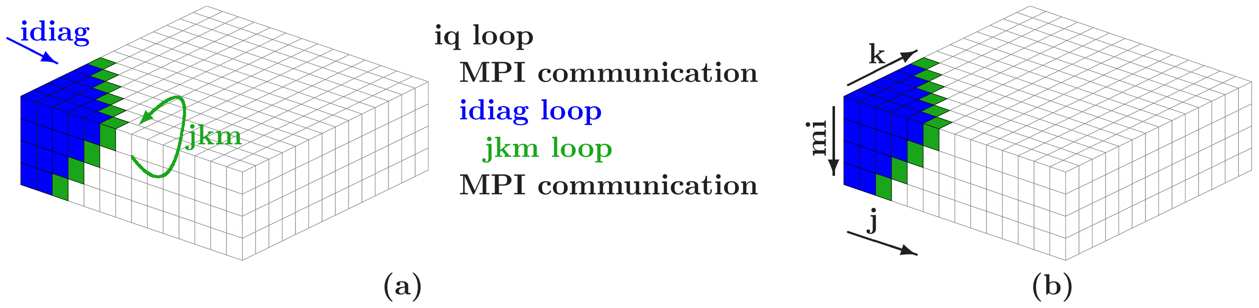
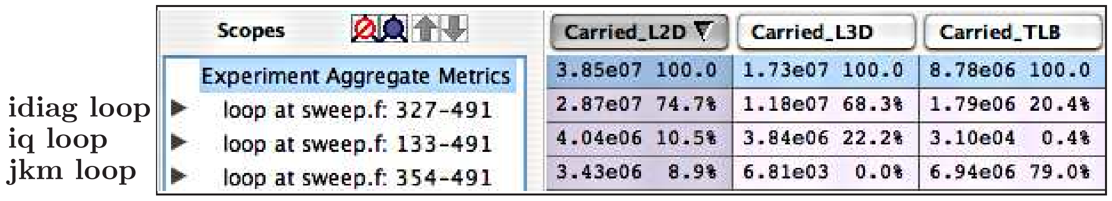


Figure 3: Diagram of Sweep3D: (a) main computation loops; (b) `jkm` iteration space

For Sweep3D, we collected memory reuse distance for a single node run using a cubic mesh size of 50x50x50 and 6 time steps without flux fix-ups. We used the reuse distance data to compute the number of L2, L3, and TLB misses for an Itanium2 processor with a 256KB 8-way set-associative L2 cache, 1.5MB 6-way set-associative L3 cache, and a 128-entry fully-associative TLB.

Figure 4 shows a snapshot from our user interface of the predicted number of carried misses for the L2 and L3 caches and for the TLB. We notice that approximately 75% of all L2 cache misses and about 68% of all L3 cache misses are carried by the `idiag` loop, while the `iq` loop carries 10.5% and 22% of the L2 and L3 cache misses respectively. The situation is different with the TLB misses. The `jkm` loop carries 79% and the `idiag` loop carries 20% of all the TLB misses.



	Scopes	Carried_L2D	Carried_L3D	Carried_TLB
	Experiment Aggregate Metrics	3.85e07 100.0	1.73e07 100.0	8.78e06 100.0
idiag loop	▶ loop at sweep.f: 327-491	2.87e07 74.7%	1.18e07 68.3%	1.79e06 20.4%
iq loop	▶ loop at sweep.f: 133-491	4.04e06 10.5%	3.84e06 22.2%	3.10e04 0.4%
jkm loop	▶ loop at sweep.f: 354-491	3.43e06 8.9%	6.81e03 0.0%	6.94e06 79.0%

Figure 4: Number of carried misses in Sweep3D

We focus on the L2 and L3 cache misses. The fact that such a high fraction of all cache misses are carried by the `idiag` loop is a good thing from a tuning point of view, because we can focus our attention on this loop. While the `iq` loop carries the second most significant number of misses, it contains also calls to communication functions. Thus, it may require more complex transformations to improve, in case it is possible at all. Table 2 summarizes the main reuse patterns contributing the highest number of L2 cache misses in Sweep3D. We notice that four loop nests inside the `jkm` loop account for the majority of the L2 cache misses. For three of these loop nests, only accesses to one data array in each of them result in cache misses. Since the `idiag` loop carries the majority of these cache misses, we can focus our attention on understanding how the array indices are computed with respect to this loop.

Array name	In scope	Reuse source	Carrying scope	% misses
src	loop 384–391	self	ALL	26.7
			idiag	20.4
			iq	3.3
			jkm	2.9
flux	loop 474–482	self	ALL	26.9
			idiag	20.4
			iq	3.4
			jkm	3.0
face	loop 486–493	self	ALL	19.7
			idiag	15.5
			iq	2.4
			jkm	1.9
sigt phikb phijb	loop 397–410	self + others	ALL	18.4

Table 2: Breakdown of L2 misses in Sweep3D.

```
384 do i = 1, it
385    phi(i) = src(i,j,k,1)
386 end do
387 do n = 2, nm
388    do i = 1, it
389       phi(i) = phi(i) +
    &         pn(m,n,iq)*src(i,j,k,n)
390    end do
391 end do
...
474 do i = 1, it
475    flux(i,j,k,1) = flux(i,j,k,1) +
    &         w(m)*phi(i)
476 end do
477 do n = 2, nm
478    do i = 1, it
479       flux(i,j,k,n) = flux(i,j,k,n)
480 &         + pn(m,n,iq)*w(m)*phi(i)
481    end do
482 end do
```

Figure 5: Accesses to `src` and `flux`.

Figure 5 shows the Fortran source code for the first two loop nests that access arrays `src` and `flux` respectively. We notice that both the `src` and the `flux` arrays are four dimensional arrays and that both of them are accessed in a similar fashion. In Fortran, arrays are stored in column-major order. Thus, the first index represents the innermost dimension and the last index is the outer most one. We notice that for both `src` and `flux`, the innermost loop matches the innermost dimension. However, the next outer loop, `n`, accesses the arrays on their outermost dimension. We return to this observation later. For now, we want to understand how the `j` and `k` indices are computed.

We mentioned that in each iteration of the `idiag` loop, the `jkm` loop traverses one diagonal plane of cells as seen in Figure 3. Each cell of the 3D mesh is defined by unique coordinates `j`, `k` and `mi`, as seen in Figure 3(b). Notice that all cells of a 3D diagonal plane have different `j` and `k` coordinates. Thus, there is no temporal reuse of `src` and `flux` carried by the `jkm` loop. The small amount of reuse observed in Table 2 is spatial reuse due to the sharing of some cache lines between neighboring cells. However, even this reuse is long enough that it results in cache misses, because the cells in a plane are not necessarily accessed in the order in which they are stored.

Consecutive `idiag` iterations access adjacent diagonal planes of cells. When we project these 3D diagonal planes onto the (`j`,`k`) plane, we notice there is a great deal of overlap between two consecutive iterations of the `idiag` loop. This explains the observed reuse carried by the `idiag` loop. However, the reuse distance is too large for the data to be still in cache on the next iteration of the `idiag` loop. Finally, the reuse carried by the `iq` loop is explained by the fact that we traverse again all cells of the mesh on a new sweep that starts from a different corner.

Notice that arrays `src` and `flux` (and `face` as well) are not indexed by the `mi` coordinate of a cell. Thus, references to the three arrays corresponding to cells on different diagonal planes that differ only in the `mi` coordinate, but with equal `j` and `k` coordinates access identical memory locations. To improve data reuse for these arrays, we need to process closer together mesh cells that differ only in the `mi` coordinate.

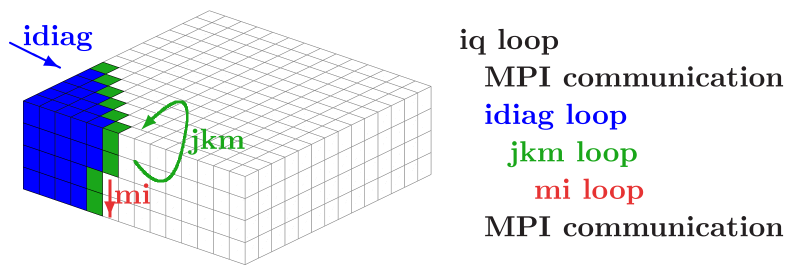
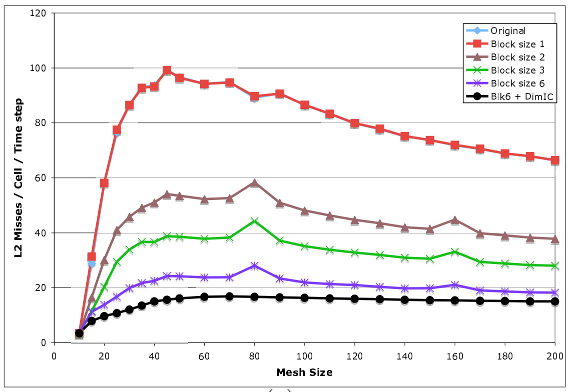


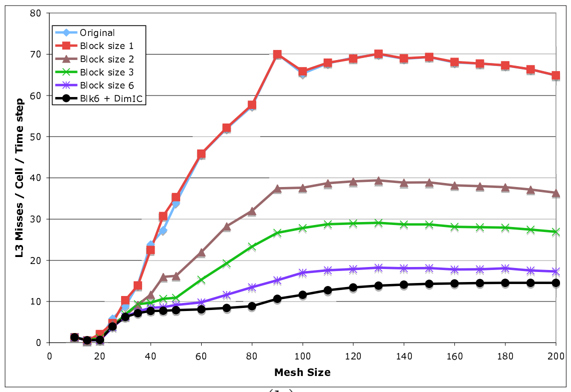
Figure 6: Diagram of Sweep3D after blocking on `mi`.

For this, we apply tiling to the `jkm` loop on the `mi` coordinate.The transformed sweep iteration space is represented graphically in Figure 6, for a blocking factor of two. Note that `mi` is not a physical dimension of the 3D mesh; rather, it represents different angles at which the neutron movements are simulated. The third physical coordinate is `i` which is contained within each cell. Thus, by simulating multiple angles at once, we achieve better data reuse. The number of angles specified in our input file was six. Therefore, we measured the performance of the transformed code on an Itanium2 machine using blocking factors of one, two, three and six.

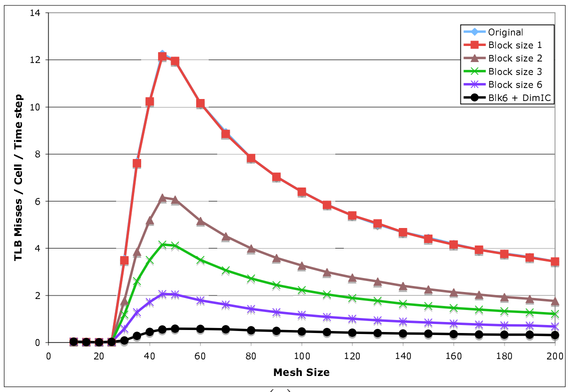
Figures 10(a),(b) and (c) present the number of L2, L3 and TLB misses for the original code and for the transformed code with the four different blocking factors. All figures present the performance metrics normalized to the number of cells and time steps so that the results for different problem sizes can be easily shown on a single graph. The figures show that the original code and the code with a blocking factor of one have identical memory behavior. As the blocking factor increases, less and less accesses miss in the cache. The last curve in each figure represents the performance of the transformed code with a blocking factor of six plus a dimensional interchange for several arrays to better reflect the way in which they are traversed. For the `src` and `flux` arrays we moved



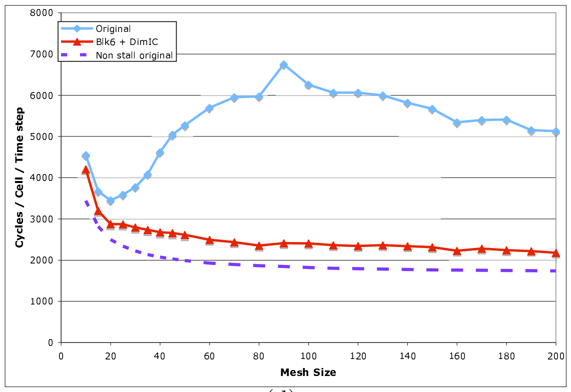
(a)



(b)



(c)



(d)

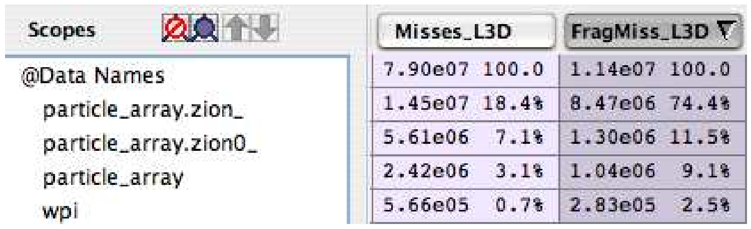
Figure 7: Performance of the original and improved Sweep3D codes on an Itanium2 machine.

the `n` dimension into the second position. These transformations reduce cache and TLB misses by integer factors. Figure 10(d) presents the normalized execution time of the original and transformed codes. The improved code has a speedup of 2.5x and we achieve ideal scaling of the execution time between mesh sizes 20 and 200 which represents a thousand-fold increase of the working set size. The dashed line in Figure 10(d) represents the non-stall execution time as measured with hardware performance counters. Notice that we eliminated a large fraction of the observed stall time with our transformations. Note that the non-stall time depicted in the figure is not the absolute minimum time that can be achieved on the Itanium. It is just the minimum time that can be achieved with the instruction schedule generated by the Intel compiler. We reduced Sweep3D's execution time further, as well as its non-stall time, by improving the compactness of the instruction schedule [1].

5.2 Analysis and tuning of GTC

The Gyrokinetic Toroidal Code is a 3D particle-in-cell (PIC) code used for studying the impact of fine-scale plasma turbulence on energy and particle confinement in the core of tokamak fusion reactors [16]. The PIC algorithm consists of three main sub-steps: 1) deposit the charge from particles onto the grid (routine `chargei`), 2) compute and smooth the potential field (routines `poisson` and `smooth`), and 3) compute the electric field and push particles using Newton's laws of physics (routines `pushi` and `gcmotion`). Compared to the Sweep3D benchmark, the GTC code is significantly more complex with the computation kernel spread over several files and routines.

For GTC, we collected reuse distance data for a problem size consisting of a single poloidal plane with 64 radial grid points and 15 particles per cell. From the reuse distance histograms collected for each reuse pattern, we computed the number of cache misses, the number of misses due to fragmentation in cache lines, the number of irregular misses, and the number of carried misses as explained in sections 2 and 3, for an Itanium2 cache architecture. All metrics are computed at loop level as well as for individual data arrays.

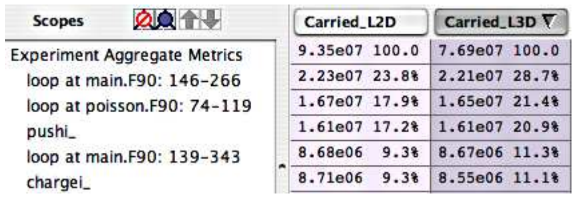


Scopes	Misses_L3D	FragMiss_L3D
@Data Names	7.90e07 100.0	1.14e07 100.0
particle_array.zion_	1.45e07 18.4%	8.47e06 74.4%
particle_array.zion0_	5.61e06 7.1%	1.30e06 11.5%
particle_array	2.42e06 3.1%	1.04e06 9.1%
wpi	5.66e05 0.7%	2.83e05 2.5%

Figure 8: Data arrays contributing the largest number of fragmentation L3 cache misses.

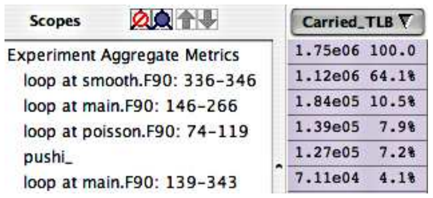
Figure 8 presents a snapshot of our viewer showing the data arrays that account for the highest number of L3 cache misses due to fragmentation of data in cache lines. The first metric in the figure represents the total number of L3 cache misses incurred by all accesses to these arrays in the entire program. Data arrays `zion` and its shadow `zion0` are global arrays storing information about each particle in the local tokamak domain. They are defined as 2D Fortran arrays organized as arrays of records with seven data fields for each particle. Array `particle_array` is an alias for the `zion` array, used inside a "`C`" routine `gcmotion`.

Notice that accesses to the two `zion` arrays, including the alias `particle_array`, account for 95% of all fragmentation misses to the L3 cache. This amounts to about 48% of all L3 cache misses incurred on the `zion` arrays, and about 13.7% of all L3 cache misses in the program. Most loops that work with the `zion` arrays reference only some of the seven fields associated with each particle. Using our viewer, we identified the loops with the highest contribution to the miss and fragmentation metrics. We noticed two loops where only one out of the seven fields of the `zion` array was referenced for each particle. To eliminate unnecessary cache misses due to fragmentation, we transposed the two `zion` arrays, so that each of the seven fields is stored separately in its own vector. This amounts to transforming the array of structures into a structure of arrays.



Scopes	Carried_L2D	Carried_L3D
Experiment Aggregate Metrics	9.35e07 100.0	7.69e07 100.0
loop at main.F90: 146-266	2.23e07 23.8%	2.21e07 28.7%
loop at poisson.F90: 74-119	1.67e07 17.9%	1.65e07 21.4%
pushi_	1.61e07 17.2%	1.61e07 20.9%
loop at main.F90: 139-343	8.68e06 9.3%	8.67e06 11.3%
chargei_	8.71e06 9.3%	8.55e06 11.1%

(a)



Scopes	Carried_TLB
Experiment Aggregate Metrics	1.75e06 100.0
loop at smooth.F90: 336-346	1.12e06 64.1%
loop at main.F90: 146-266	1.84e05 10.5%
loop at poisson.F90: 74-119	1.39e05 7.9%
pushi_	1.27e05 7.2%
loop at main.F90: 139-343	7.11e04 4.1%

(b)

Figure 9: Program scopes carrying the most (a) L3 cache misses, and (b) TLB misses.

Figure 9(a) presents the program scopes that carry more than 2% of all L3 cache misses. The loop at `main.F90:139-343` is the main loop of the algorithm iterating over time steps and it carries about 11% of all L3 cache misses. Each time step of the PIC algorithm executes a 2nd order

Runge-Kutta predictor-corrector method, represented by the second loop of the main routine, at lines `146-266`. The two main loops carry together about 40% of all L3 cache misses. These are cache misses due to data reuse between the three sub-steps of the PIC algorithm, and across consecutive time steps or the two phases of the predictor-corrector method in each time step. Because each of the three sub-steps of the PIC algorithm requires the previous step to be completed before it can start executing, these cache misses cannot be eliminated by time skewing or pipelining of the three sub-steps. Thus, we focus our attention on the other opportunities for improvement.

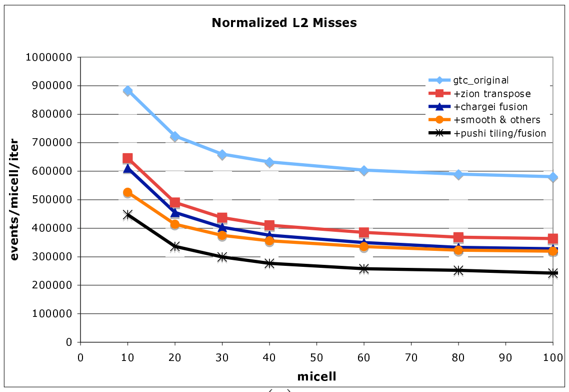
The `poisson` routine computes the potential field on each poloidal plane using an iterative Poisson solver. Cache misses are carried by the iterative loop of the Poisson solver (at lines `74-119`), and unfortunately cannot be eliminated by loop interchange or loop tiling because of a true recurrence in the solver. However, the amount of work in the Poisson solver is proportional to the number of cells in the poloidal plane. As we increase the number of particles that are simulated, the cost of the charge deposition and particle pushing steps increases, while the cost of the Poisson solver stays constant. Thus, the execution cost of the `poisson` routine becomes relatively small in comparison to the cost of the entire algorithm as the number of particles per cell increases.

We focus now on the `chargei` and the `pushi` routines. Our analysis identified that about 11% of all L3 cache misses are due to reuse of data in two loops of the `chargei` routine that iterate over all particles. The first loop was computing and storing a series of intermediate values for each particle; the second loop was using those values to compute the charge deposition onto the grid. However, by the time the second loop accessed the values computed in the first loop, they had been evicted from cache. By fusing the two loops, we were able to improve data reuse in `chargei`, and to eliminate these cache misses.

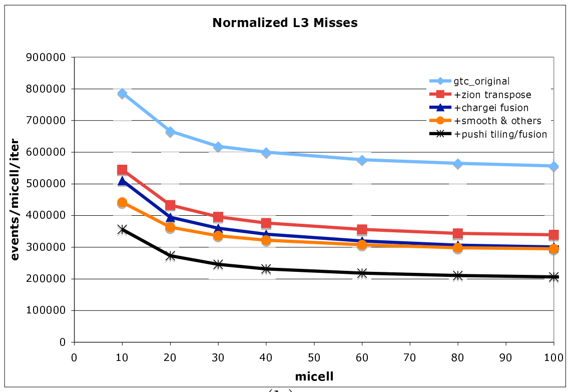
The `pushi` routine calculates the electrical field and updates the velocities of the ion particles. It contains several loop nests that iterate over all the particles, and a function call to a "`C`" routine, `gcmotion`. The `gcmotion` routine consists of a single large loop that iterates over all the particles as well. Our analysis identified that for the problem size that we used, `pushi` carries about 20% of all L3 cache misses between the different loop nests and the `gcmotion` routine. This reuse pattern corresponds to the fifth entry in Table 1, because the `gcmotion` routine is both a source and a destination scope for some of the reuse arcs carried by `pushi`. While `gcmotion` consists of just one large loop, we cannot inline it in `pushi` because these two routines are written in different programing languages. Instead, we identified a set of loops that we could fuse, strip-mined all of them, including the loop in `gcmotion`, with the same stripe s, and promoted the loops over stripes in the `pushi` routine, fusing them. The result is a large loop over stripes, inside of which are the original loop nests and the function call to `gcmotion`. These transformed loop nests work over a single stripe of particles, which is short enough to ensure that the data is reused in cache.

We also identified a loop nest in routine `smooth` that was contributing about 64% of all TLB misses for the problem size that we used. The outer loop of the loop nest, which was carrying all these TLB misses (see Figure 9(b)), was iterating over the inner dimension of a three dimensional array. We were able to apply loop interchange and promote this loop in the innermost position, thus eliminating all these TLB misses.

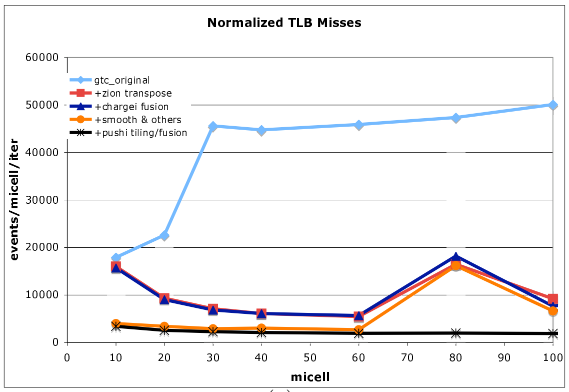
Figure 10 presents the single node performance of GTC on a 900MHz Itanium2. The four graphs compare the number of L2, L3 and TLB misses, and the execution time respectively, of the original and the improved GTC codes, as we vary the number of particles per cell on the `x` axis. Notice how the code performance improved after each transformation. The large reduction in cache and TLB misses observed after the transposition of the `zion` arrays is due in part to a reduction



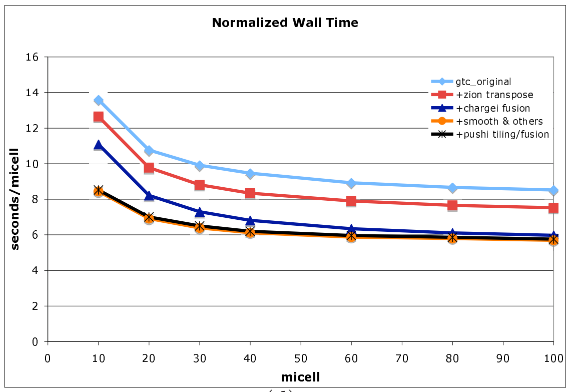
(a)



(b)



(c)



(d)

Figure 10: GTC performance after each code transformation on an Itanium2 machine.

in the number of unnecessary prefetches inserted by the Intel compiler, which was an unexpected side-effect, as well as because of an increase in data locality for other arrays after the loops working on the `zion` array had to stream through much less data because of the reduced fragmentation. The curve labeled as "*smooth & others*" represents not only the loop interchange in the `smooth` routine, but also two other small transformations that we did not explain here because they use analysis techniques outside the scope of this paper [1]. However, the performance improvement due to these transformations is significant only when the number of particles is relatively small. Notice also how the tiling/fusion in the `pushi` routine significantly reduced the number of L2 and L3 cache misses, but these improvements did not translate into a smaller execution time. When we tiled & fused the loop nests in `pushi`, we created a large loop over stripes that overflowed the small 16KB dedicated instruction cache on Itanium. Thus, the improvement in data locality was mitigated by an increase in the number of instruction cache misses. We expect this transformation to have a bigger impact on other architectures that have a larger instruction cache, including Montecito, the new member of the Itanium family of processors.

Overall, we reduced the number of cache misses by at least a factor of two, the number of TLB misses was reduced by a huge margin, and we observed a 33% reduction of the execution time, which amounts to a 1.5x speedup.

6 Related work

Beyls and D'Hollander [7] describe RDVIS, a tool for visualizing reuse distance information clustered based on the intermediary executed code (IEC) between two accesses to the same data, and SLO, a tool that suggests locality optimizations based on the analysis of the IEC. The capabilities of their tools are similar to those we describe in this paper. At the same time, our implementations differ significantly in the ways we collect, analyze and visualize the data. In addition to the histograms of reuse distances, Beyls and D'Hollander collect the sets of basic blocks executed between each pair of accesses to the same data. Afterwards, an offline tool clusters the different reuse patterns based on the similarity of the IEC. A second tool analyzes the IEC to determine the carrying scope of each reuse. In contrast, we directly determine the scopes where the two ends of a reuse arc reside, as well as the carrying scope based on a dynamic stack of program scopes. We cluster the reuse patterns based on their source, destination and carrying scope directly at run-time, which reduces the amount of collected data. Moreover, this approach enables us to leverage the modeling work described in [13] to predict the scaling of reuse patterns for larger program inputs. Finally, our implementations differ also in the way the data is visualized. RDVIS displays the significant reuse patterns as arrows drawn over the intermediary executed code between data reuses. In contrast, we focused on computing metrics that enable us to find the significant reuse patterns using a top-down analysis of the code, which we think it is more scalable to analyzing large codes where reuse patterns span multiple files. In addition, we identify reuse patterns due to indirect or irregular memory accesses, and inefficiencies due to fragmentation of data in cache lines, which enables us to pinpoint additional opportunities for improvement.

Chilimbi et al. [8] profile applications to monitor access frequency to structure fields. They classify fields as hot and cold based on their access frequencies. Small structures are split into hot and cold portions. For large structures they apply field reordering such that fields with temporal affinity are located in same cache block. Zhong et al. [18] describe k-distance analysis to understand data affinity and identify opportunities for array grouping and structure splitting. We use static analysis to understand fragmentation of data in cache lines, and find opportunities for structure or array splitting.

7 Conclusions

This paper describes a data locality analysis technique based on collecting memory reuse distance separately for each reuse pattern of a reference. This approach uncovers the most significant reuse patterns contributing to an application's cache miss counts and identifies program transformations that have the potential to improve memory hierarchy utilization. We describe also a static analysis algorithm that identifies opportunities for improving spatial locality in loop nests that traverse arrays using a non-unit stride. We used this approach to analyze and tune two scientific applications. For Sweep3D, we identified a loop that carried 75% of all L2 cache misses in the program. The insight gained from understanding the most significant reuse patterns in the program enabled us to transform the code to increase data locality. The transformed code incurs less than 25% of the cache misses observed with the original code, and the overall execution is 2.5x faster. For GTC, our analysis identified two arrays of structures that were being accessed with a non-unit stride, which almost doubled number of cache misses to these arrays above ideal. We also identified the main loops carrying cache and TLB misses. Reorganizing the arrays of structures into structures of arrays, and transforming the code to shorten the reuse distance of some of the reuse patterns, reduced cache misses by a factor of two and execution time by 33%.

References

[1] Omitted for blind review.

[2] The ASCI Sweep3D Benchmark Code. DOE Accelerated Strategic Computing Initiative. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`.

[3] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[5] B. Bennett and V. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[6] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *IASTED conference on Parallel and Distributed Computing and Systems 2001 (PDCS01)*, pages 617–662, 2001.

[7] K. Beyls and E. H. D'Hollander. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 373–382, New York, NY, USA, 2006. ACM Press.

[8] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[9] C. Ding and Y. Zhong. Reuse distance analysis. Technical Report TR741, Dept. of Computer Science, University of Rochester, 2001.

[10] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis. In *The Second ACM SIGPLAN Workshop on Memory System Performance*, Washington, DC, USA, June 2004.

[11] W. W. Lee. Gyrokinetic approach in particle simulation. *Physics of Fluids*, 26:556–562, Feb. 1983.

[12] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and Modeling of Computer Systems*, pages 2–13. ACM Press, 2004.

[13] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*, 2005.

[14] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[15] V. S. Pai and S. V. Adve. Code transformations to improve memory parallelism. In *International Symposium on Microarchitecture MICRO-32*, pages 147–155, Nov 1999.

[16] N. Wichmann, M. Adams, and S. Ethier. New Advances in the Gyrokinetic Toroidal Code and Their Impact on Performance on the Cray XT Series, May 2007. The Cray User Group, CUG 2007.

[17] Y. Zhong, S. G. Dropsho, and C. Ding. Miss Rate Prediction across All Program Inputs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, Sept. 2003.

[18] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 255–266, New York, NY, USA, 2004. ACM Press.