

Pinpointing and Exploiting Opportunities for Enhancing Data Reuse

Gabriel Marin and John Mellor-Crummey

Department of Computer Science

Rice University

6100 Main St., MS 132

Houston, TX 77005

{mgabi,johnmc}@cs.rice.edu

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 exhibit a large cache miss rate, 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 an approach that uses memory reuse distance to identify an application’s most significant memory access patterns causing cache misses and provide insight into ways of improving data reuse. Unlike previous approaches, our tool combines (1) analysis and instrumentation of fully optimized binaries, (2) online analysis of reuse patterns, (3) fine-grain attribution of measurements and models to statements, loops and variables, and (4) static analysis of access patterns to quantify spatial reuse. We demonstrate the effectiveness of our approach for understanding reuse patterns in 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.

I. INTRODUCTION

To improve the performance of an application we need to understand not only *where* a code executes inefficiently, but more importantly to understand *why*, *i.e.* we must identify the factors that limit performance at each point in a program. In prior work [15] we describe techniques for understanding performance bottlenecks due to insufficient instruction level parallelism or due to a mismatch between an application’s instruction mix and the type of execution units available on a target architecture. In this paper we present techniques for identifying performance bottlenecks due to poor data locality and we compute metrics that provide insight into ways of improving data reuse.

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.

Typically, performance tools collect or simulate cache miss counts and rates and attribute them at the function level. While such information identifies the functions that suffer from poor data locality, in our experience, this is often insufficient to diagnose the causes for poor locality and to identify what code 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 measures 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 [3], [16], characterizing data locality in program executions for individual program inputs [4], [7], and using memory reuse distance data from training runs to predict cache miss rate for other program inputs [9], [13], [23].

To understand how memory reuse distance data translates into cache miss predictions, it is best to consider a short example. Suppose a memory block B is reused after other n distinct memory blocks have been accessed since B was last accessed. For a fully-associative cache, at the time of the reuse, block B would still be in cache if n is smaller than the cache size; if n is greater than or equal to the cache size, block B would have been evicted from the cache by one of intervening accesses. Thus, to understand if a memory access is a hit or miss in a fully-associative cache using LRU replacement policy, one can simply compare the distance of the reuse with the size of the cache. We have shown elsewhere [14] that a reuse distance based probabilistic model yields accurate predictions for set-associative caches as well.

Both temporal and spatial reuse determine how effectively an application exploits the cache, and thus both are important for an application’s performance. The next two sections describe techniques for highlighting the main data reuse patterns in an application, which provide insight into ways of improving temporal locality, and a static analysis approach that identifies opportunities for improving spatial locality through data layout transformations.

The rest of the paper is organized as follows. Section II presents our strategy for collecting and processing reuse distance information at the level of reuse patterns. Section III describes techniques for pinpointing and understanding opportunities for improving spatial locality through data layout transformations. Section IV describes how to interpret the information computed in Sections III and IV to improve data reuse in applications. Section V describes the process of tuning two scientific applications using our techniques. Section VI describes the closest related work. Section VII presents our conclusions and plans for future work.

II. UNDERSTANDING DATA REUSE PATTERNS

Knowing which loops of an application experience a large number of cache misses, usually does not provide sufficient insight by itself for understanding how to improve data locality. The reason for this is that data reuse, the main determinant of cache performance, is not a local phenomenon. The same data may be accessed in multiple loops located in different routines. Moreover, to understand why an application accesses the same data repeatedly we need to identify not only the places where data is referenced, but also the algorithmic loops that are driving this reuse. In some cases, the loop driving the reuse can be found locally where the data is accessed, *e.g.* when a loop iterates over the inner dimension of an array. Other times, some outer loop, which might be in a different routine, causes the application to access the same data repeatedly, *e.g.*, in consecutive time steps. To understand how to improve data reuse, we must understand both where same data is accessed and what loops are driving the reuse.

Previous work on computing cache miss predictions from memory reuse distance information has explored approaches that associate reuse distance data with either individual references [9], [13], groups of related references from the same loop [14], or an entire application [23]. Associating reuse distance data with a section of code, be it a reference, a loop or an entire application, is sufficient for computing the number of cache misses incurred by that piece of code. However, these approaches use 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. Collecting reuse distance separately for each reuse arc of a memory reference not only provides insight into where cache misses occur, but also captures the correlation between references that access the same data, which provides insight into the application’s data access patterns. As we describe later, knowing the program scope that carries this reuse is important for understanding how to improve reuse.

We use instrumentation of application binaries to measure reuse distance at run-time. Because we analyze and instrument object code, our tools are language independent and naturally handle applications with modules written in different languages. Before each memory reference in the program we invoke an event handler routine that updates a histogram of reuse distance values for the reference. The event handler

routine increments a logical clock by one each time a memory instruction is executed. A three level hierarchical block table is used to associate the logical time of last access with every memory block referenced by the program. The time-stamp enables us to determine how many memory operations are executed between a pair of accesses to the same datum. To understand the number of distinct memory locations accessed between consecutive accesses to a particular datum, we use a balanced binary tree with a node for each memory block referenced by the program. The sorting key for each node in the tree is the logical time of the last access to the memory location represented by the node. On each memory access we can compute how many distinct memory blocks have an access time greater than the time-stamp of current block in $\log(M)$ time, where M is the size of the tree and represents the number of distinct memory blocks touched by the application. A full description of the algorithm is presented elsewhere [13].

We extended the data collection infrastructure presented in [13] to record information about the identity of the most recent access to a memory block. This approach enables us to associate a reuse distance with a (*source, destination*) pair of scopes where the two endpoints of the reuse arc reside. This additional information enables us to report to the user 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.

To determine the program scope that is driving a reuse, we add instrumentation to monitor entry and exit of routines and loops. At run-time, we maintain a dynamic stack of scopes based on the entry and exit events that are executed. 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 program scope that is driving a reuse is the innermost dynamic scope in which the data is reused. Therefore, on a memory access 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 driving scope, which we also call 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.

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

```

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

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

```

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

interchange and blocking are well studied compiler transformations. A more thorough discussion of these transformations can be found in [1]. 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.

Compared to our previous work [13], the more refined approach that we describe in this paper increases 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. For this reason, our previous implementation maintained fewer histograms; however, they had a large number of bins to capture the different distance values encountered. In contrast, our new approach maintains more but smaller histograms.

All reuse distance data we collect can still be modeled using the algorithm presented in [14] to predict the distribution of reuse distances for other program inputs. Essentially, we model the distribution and scaling of reuse distance histograms as a function of problem size by computing an appropriate partitioning of reuse distance histograms into bins of accesses that have similar scaling of their measured reuse distance across several problem sizes. We model the execution frequency and reuse distance scaling of each bin as a linear combination of a set of basis functions. In addition, since with our new approach reuse distance data is collected and modeled at a finer granularity, 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. In general, the further removed the carrying scope of a reuse pattern is from the scopes where the data is accessed, the more difficult it is to improve it.

We compute several metrics based on our memory reuse analysis. For each scope, we compute traditional cache miss counts; we use this data to identify loops responsible for a high fraction of cache misses. In addition, we break down cache miss counts by the scope that accessed data last before it was evicted, the scope that is carrying these long data reuses, or a combination of these two factors. To guide tuning, we also compute the number of cache misses carried by each scope. A scope \mathcal{S} is carrying those cache misses produced by reuse patterns for which \mathcal{S} is the carrying scope. We break down carried miss counts by the source or/and destination scopes of the reuse. These 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 V, we describe how we use this information to guide the tuning of two scientific applications.

III. FRAGMENTATION IN CACHE LINES

Both temporal and spatial reuse determine how efficiently an application exercises the cache. Temporal reuse can be understood from reuse distance measurements. Spatial reuse, however, depends also on the layout of data in memory. Caches are organized in 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 array reference and each loop nest in the program. To identify where fragmentation occurs, we use static analysis.

For each loop nest, we identify references that access the same data arrays with the same stride. We say such references are *related*. Understanding which references are *related* from

```

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

```

Fig. 2. Cache line fragmentation example.

looking at machine code requires detailed binary analysis. First, we compute symbolic formulas that describe the memory locations accessed by each reference. We compute a symbolic formula for the *first location* accessed by a reference by tracing back along use-def chains in its enclosing routine. Tracing starts from the registers used in the reference’s address computation. For references inside loops, we also compute symbolic *stride* formulas, which describe how the accessed location changes from one iteration to the next. Stride formulas have two additional flags. One 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). A more detailed description of how we compute symbolic formulas is presented in [12], [14].

We use the computed symbolic formulas to understand which references access memory with the same stride. 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 [12]. 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 we use the following three step algorithm to compute the *fragmentation factor* of each group¹. At a high level, the algorithm attempts to find the loop level that iterates over the inner-most dimension of an array and then it determines if the combined footprint of all references that are part of a group overlaps the array’s footprint without gaps.

Step 1. Find the enclosing loop, L , for which this group of references experiences the smallest non-zero constant stride. When a reference group is enclosed in a loop nest, we traverse the loops from the inside out. Inner loops are executed much more frequently than outer loops, and we want to find the innermost loop that iterates over an array’s inner dimension. We terminate the search if a loop is encountered for which references have an irregular stride. The reason for this is that we cannot determine through static analysis the locations accessed by irregular access patterns or their contribution to spatial reuse. Moreover, we report separately cache misses produced by irregular memory access patterns, together with

¹Note 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.

information about the scopes where the data is accessed and the loop that is driving the irregular data reuse. If we cannot find a loop with a constant non-zero stride, we do not compute any *fragmentation factor* for that group of references because we do not have enough information to compute the combined footprint of the 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 inner loop I , and the stride is 32 bytes 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 of loop L 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*.

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

Step 3. Compute the hot footprint information for each *reuse group* derived from a group of related references. The hot footprint is defined as the union of the locations into a memory block of size s , accessed by all references of a *reuse group*. We use modular arithmetic to map the locations accessed by different references into the same block of size s , possibly on different iterations of loop L . This is equivalent to 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, as seen with our example in Figure 2, 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 [19], 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 [12, pages 189–196].

For our example in Figure 2, 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 cache miss counts due to fragmentation effects at each memory level. 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 formula for the references at its destination end.

IV. INTERPRETING THE PERFORMANCE DATA

To identify performance problems and opportunities for tuning, we output all metrics described in the previous sections in XML format, and we use the `hpcviewer` user interface [17] that is part of HPCToolkit [18] to explore the data. The viewer enables us to explore the data in a top-down fashion, to sort the data by any metric and to associate metrics with the program source code and with data array names.

For all metrics we compute aggregated values at each level of the program scope tree. The root node of the program scope tree contains the values of the metrics aggregated at the entire program level. On the second level of the tree we have source code files. On the third level we have the routines located within each file. Under each routine we can have zero, one or more levels of loops, corresponding to the source code loop nesting structure in that routine.

We can visualize both the exclusive and the inclusive² values of the metrics at each level of a 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 performance metric (e.g., misses, fragmentation), or we can compare the exclusive values across all scopes of a program.

Not all metrics can be sensibly aggregated based on the static program scope tree structure. 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 [2] 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 entries represent not individual program scopes, but pairs of scopes that correspond to 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 I 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*, *destination*, and *carrying* scopes of a reuse pattern, respectively. 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 I.

V. 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 [8] 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 extensively studied and tuned already [7], [10], [18], [20], [22]. 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.

²An inclusive value quantifies the contribution of a scope and all its children scopes to a given metric. An exclusive value quantifies the contribution of a scope without its children.

Scenario	Transformations & comments
large fragmentation miss count 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
$S \neq D$, 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 I
RECOMMENDED TRANSFORMATIONS FOR IMPROVING MEMORY REUSE.

```

131 DO iq=1,8      ! octants
168 DO mo=1,mmo  ! angle pipelining loop
217 DO kk=1,kb   ! k-plane pipelining loop
237 RECV E/W     ! recv block I-inflows
280 RECV N/S     ! recv block J-inflows
326 DO idiag=1,jt+nk-1+mmi-1
353   DO jkm=1,ndiag
502   ENDDO      ! jkm
504   ENDDO      ! idiag
513   SEND E/W   ! send block I-outflows
550   SEND N/S   ! send block J-outflows
586   ENDDO      ! kk
619   ENDDO      ! mo
623   ENDDO      ! iq

```

Fig. 3. Loop structure of Sweep3D's computational kernel.

A. Analysis and tuning of Sweep3D

Sweep3D performs a series of diagonal sweeps over a 3D Cartesian mesh, which is distributed across the processors of a parallel job. Figure 3 shows the structure of Sweep3D's computational kernel. There are five levels of principal loops, some of them irregular. Furthermore, many of the principal loops contain additional two or three level loop nests that are omitted in Figure 3 for brevity. Figure 4(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 neighboring processors. Finally, the outer `iq` loop iterates over all octants, starting a sweep from each corner of the global mesh.

For Sweep3D, we collected memory reuse distance for a single node run using a cubic mesh size of $50 \times 50 \times 50$ 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 5 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

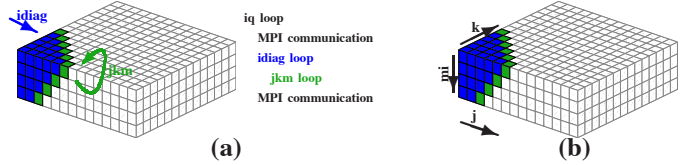


Fig. 4. Diagram of Sweep3D: (a) computation loops; (b) `jkm` iteration space

Scopes	Carried_L2D	Carried_L3D	Carried_TLB
Experiment Aggregate Metrics	3.85e07 100.0	1.73e07 100.0	8.78e06 100.0
idiag loop	2.87e07 74.7%	1.18e07 68.3%	1.79e06 20.4%
iq loop	4.04e06 10.5%	3.84e06 22.2%	3.10e04 0.4%
jkm loop	3.43e06 8.9%	6.81e03 0.0%	6.94e06 79.0%

Fig. 5. Number of carried misses in Sweep3D

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.

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 II summarizes data collected by our tool that was obtained from our user interface. It includes 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.

Figure 6 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 array dimension and the

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
sig phikb phijb	loop 397–410	self + others	ALL	18.4

TABLE II
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

```

Fig. 6. Accesses to arrays `src` and `flux`.

last index is the outermost one. In these code fragments, we notice that the innermost loops accessing the `src` and `flux` arrays respectively match the innermost dimension of these arrays. However, the next outer loop, `n`, accesses these arrays on their outermost dimension. We return to this observation later. First, 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 4. Each cell of the 3D mesh is defined by unique coordinates `j`, `k` and `mi`, as seen in Figure 4(b). References to `src` and `flux` are not indexed by the `mi` coordinate. However, all cells of a 3D diagonal plane have unique coordinates even in the two dimensional (`j,k`) address space. Thus, there is no temporal reuse of `src` and `flux` carried by the `jkm` loop. The small amount of reuse observed in Table II 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 data to be reused from cache on the next iteration of the `idiag` loop. Finally, the reuse carried by the `iq` loop is explained by the fact that we again traverse all cells of the mesh during a new sweep that starts from a different corner.

We made the observation 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.

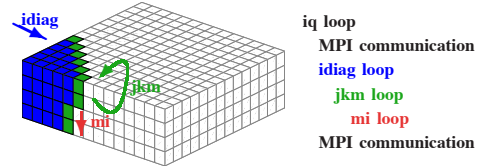


Fig. 7. Diagram of Sweep3D after blocking on `mi`.

For this, we manually tiled the `jkm` loop on the `mi` coordinate.³ The transformed sweep iteration space is represented graphically in Figure 7, 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. Our transformation groups the processing of different angles closer together, achieving better data reuse at the expense of slightly lower fine-grain parallelism. The number of angles specified in our input file was six. Therefore, we measured the performance of the transformed code on an Itanium2-based system using blocking factors of one, two, three and six.

Figures 8(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 displayed 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, fewer 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

³Since the `jkm` loop represents a multi-dimensional diagonal wavefront, tiling this loop is difficult to automate.

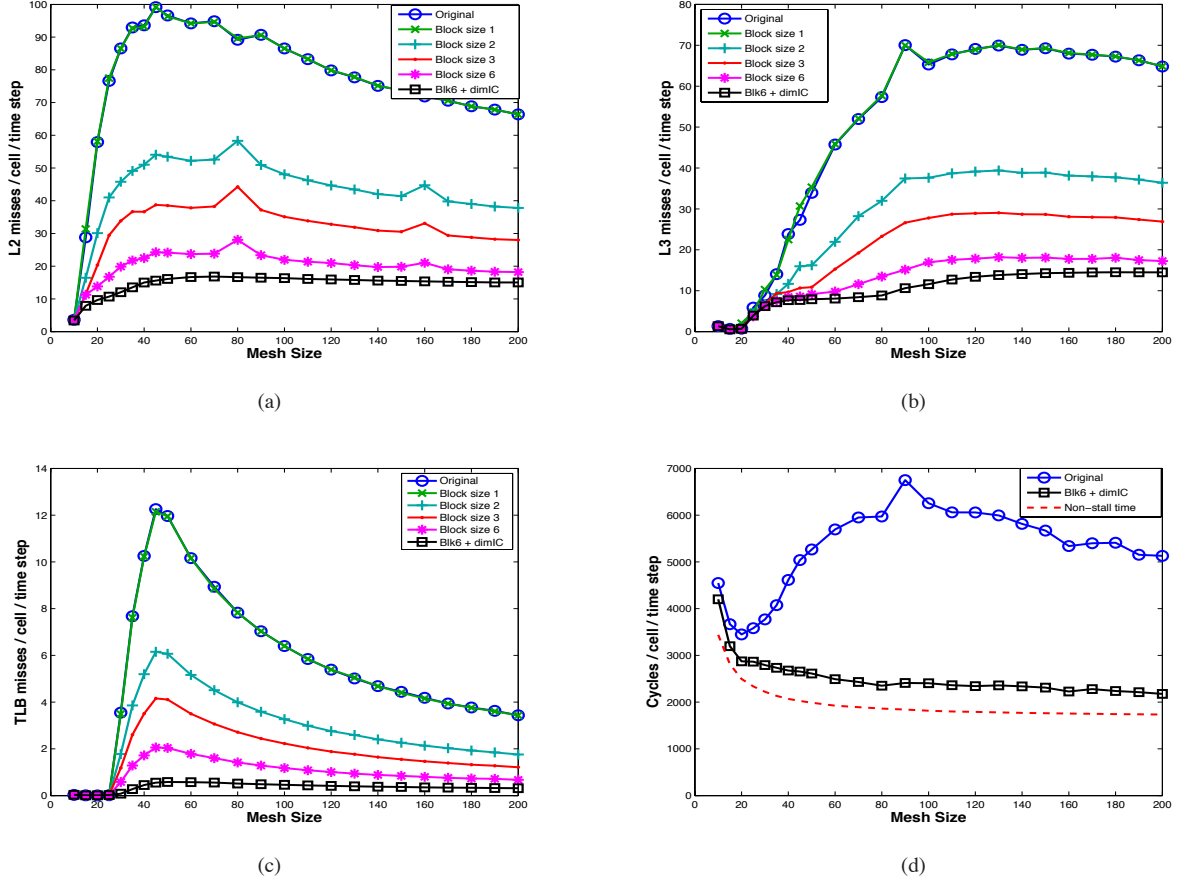


Fig. 8. Performance of the original and improved Sweep3D codes on an Itanium2 machine: (a) L2 misses; (b) L3 misses; (c) TLB misses; (d) running time.

are traversed. For the `src` and `flux` arrays we moved the `n` dimension into the second position. These transformations reduce cache and TLB misses by integer factors. Figure 8(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 8(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 non-stall time and execution time further by improving the compactness of the instruction schedule for an overall speed-up factor of 3x.⁴

⁴In previous work, we described how our tool employs static analysis of instruction schedules and how it uncovered an opportunity for enhancing instruction-level parallelism in Sweep3D [15].

B. 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 [21]. 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 `gmotion`). 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 II and III, for an Itanium2 cache architecture. All metrics are computed at loop level as well as for individual data arrays.

Figure 9 presents a snapshot of our viewer showing the data

	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%

Fig. 9. Data arrays contributing the largest number of fragmentation L3 cache misses.

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 a few 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.

Figure 10(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 across 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 due to a recurrence in the solver. We did however notice that the highest number of cache misses in the `poisson` routine was incurred by accesses to two three-dimensional arrays, `ring` and `indexp`, even though they were accessed with unit stride. With closer inspection, we found that the upper bound of the innermost loop used to iterate over the inner dimension of these two arrays was not constant. Thus, only some of the elements on a column were being accessed, resulting in partially utilized cache blocks at the end of each column. Our static analysis for cache fragmentation cannot detect such cases at this time because the elements are accessed with stride one, and the elements that are not accessed are contiguous at the end of each column. We reorganized these arrays into contiguous linear arrays which improves spatial locality. This transformation removes only a small fraction of the total number of cache misses incurred on these arrays. There is unfulfilled temporal reuse carried by the iterative loop of the Poisson solver, which cannot be improved.

However, the amount of work in the Poisson solver is proportional to the number of cells in the poloidal plane, but independent of the number of particles in each cell. As we increase the number of particles that are simulated, the costs of the charge deposition and particle pushing steps increase, 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 tool identified that about 11% of all L3 cache misses are due to reuse of data between 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 I, 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 programming 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 into the `pushi` routine, fusing them. The result is a large loop over stripes, inside of which are the original loop

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%
chargeL_	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)

Fig. 10. Program scopes carrying the most (a) L3 cache misses, and (b) TLB misses.

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.

For the problem size that we studied, our tool reported that about 64% of all TLB misses were due to a loop nest in routine `smooth`. The outer loop of the loop nest, which was carrying these TLB misses (see Figure 10(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 and eliminate all of these TLB misses.

In prior work we described techniques for understanding performance bottlenecks due to insufficient instruction-level parallelism [15]. When applying these techniques to GTC we identified a recurrence in a prime factor transform routine `spcpft`. We increased the amount of instruction-level parallelism by applying `unroll & jam`. We also identified a similar short recurrence in one loop nest of the Poisson solver where we applied `unroll & jam` to increase fine-grain parallelism.

Figure 11 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 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` arrays had to stream through much less data because of the reduced fragmentation.

Performance improvements due to code transformations in `smooth`, `spcpft` and `poisson` are significant only when the number of particles is relatively small, since the amount of work in these routines is proportional to the number of cells in a poloidal plane and independent of the number of particles.

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, our tool pinpointed significant opportunities for tuning. We were able to capitalize on these opportunities and reduce the number of cache misses by at a factor of two or more (we studied executions for several different problem specifications), 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.

VI. RELATED WORK

Beys and D’Hollander [5] 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. However, our implementations differ significantly in the ways we collect, analyze and visualize the data. In addition to the histograms of reuse distances, Beys 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 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 [14] 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 reuse patterns as arrows drawn over the IEC 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. [6] 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

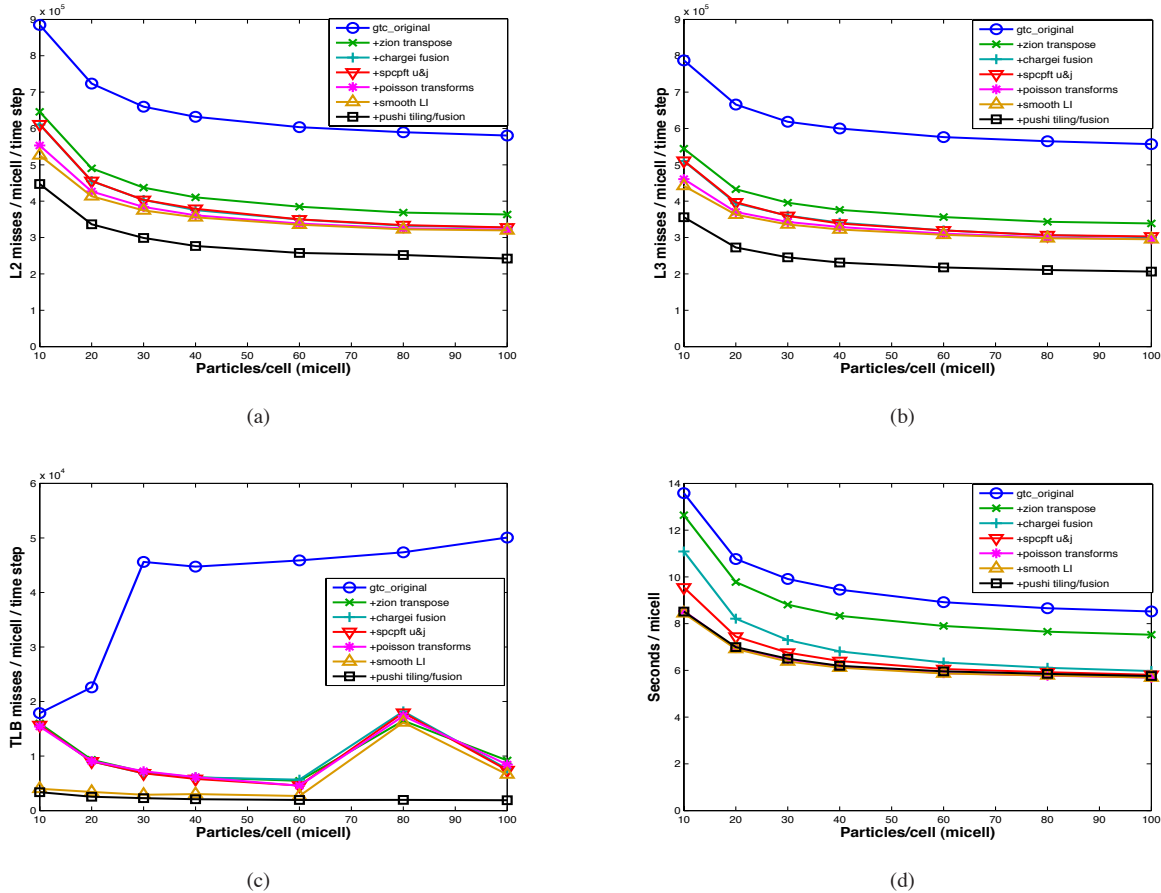


Fig. 11. GTC performance after each code transformation on an Itanium2 machine: (a) L2 misses; (b) L3 misses; (c) TLB misses; (d) execution time.

apply field reordering such that fields with temporal affinity are located in same cache block. Zhong et al. [24] 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.

Ding and Zhong [7] attribute memory reuse distance information at data-structure level to understand data locality problems in Sweep3D. They transformed Sweep3D based on this data and report a speed-up factor of 1.9 on an Origin 2000 system. We compiled and ran their improved code onto the same Itanium2 system that we used for our measurements. We observed a peak speed-up factor of 2.36 at mesh size 70, with the speed-up tailing-off towards a factor of 1.45 for larger problem sizes. The authors obtain a high speed-up for small problem sizes by transforming the code to reduce the reuse distances that we determined to be carried by the `iq` loop. However, this achievement comes at the expense of breaking the wavefront characteristics of the code, resulting in lower coarse-grain and fine-grain parallelism. By understanding the significant reuse patterns in Sweep3D, we focused on improving the reuse carried by the `idiag` loop which results in a consistently high speed-up across all mesh sizes without disrupting the application’s parallelism.

VII. 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%.

ACKNOWLEDGMENTS

This work was supported in part by the Department of Energy's Office of Science (Cooperative Agreement Nos. DE-FC02-06ER25762, DE-FC02-07ER25800), the National Science Foundation (Grant No. ACI 0103759), and Los Alamos National Laboratory (Contract Nos. 03891-001-99-4G, 74837-001-03-49, and 86192-001-04-49).

REFERENCES

- [1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [2] 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.
- [3] B. Bennett and V. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] C. Ding and Y. Zhong. Reuse distance analysis. Technical Report TR741, Dept. of Computer Science, University of Rochester, 2001.
- [8] DOE Accelerated Strategic Computing Initiative. The ASCI Sweep3D Benchmark Code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.
- [9] 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, June 2004.
- [10] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing*, 2000.
- [11] W. W. Lee. Gyrokinetic approach in particle simulation. *Physics of Fluids*, 26:556–562, Feb. 1983.
- [12] G. Marin. *Application Insight Through Performance Modeling*. PhD thesis, Dept. of Computer Science, Rice University, Dec. 2007.
- [13] 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.
- [14] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*, 2005. <http://www.cs.rice.edu/~mgabi/papers/MM-lacsi05.pdf>.
- [15] G. Marin and J. Mellor-Crummey. Application insight through performance modeling. In *Proceedings of the Performance, Computing, and Communications Conference (IPCCC'07)*, pages 65–74, Apr 2007.
- [16] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [17] J. Mellor-Crummey. Using hpcviewer to browse performance databases, Feb. 2004. <http://www.hipersoft.rice.edu/hpctoolkit/tutorials/Using-hpcviewer.pdf>.
- [18] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. HPCVIEW: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.
- [19] 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.
- [20] D. Sundaram-Stukel and M. K. Vernon. Predictive analysis of a wavefront application using LogGP. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 141–150, New York, NY, USA, 1999. ACM.
- [21] 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.
- [22] Y. Yoon, J. C. Browne, M. Crocker, S. Jain, and N. Mahmood. Productivity and performance through components: the ASCI Sweep3D application: Research Articles. *Concurr. Comput. : Pract. Exper.*, 19(5):721–742, 2007.
- [23] 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, LA, Sept. 2003.
- [24] 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.