Eurographics/ IEEE-VGTC Symposium on Visualization 2008
A. Vilanova, A. Telea, G. Scheuermann, and T. Möller
(Guest Editors)

Volume 27 (2008), Number 3

Interactive Visualization for Memory Reference Traces

A.N.M. Imroz Choudhury, Kristin C. Potter and Steven G. Parker

Scientific Computing and Imaging Institute, University of Utah, USA

Abstract

We present the Memory Trace Visualizer (MTV), a tool that provides interactive visualization and analysis of the sequence of memory operations performed by a program as it runs. As improvements in processor performance continue to outpace improvements in memory performance, tools to understand memory access patterns are increasingly important for optimizing data intensive programs such as those found in scientific computing. Using visual representations of abstract data structures, a simulated cache, and animating memory operations, MTV can expose memory performance bottlenecks and guide programmers toward memory system optimization opportunities. Visualization of detailed memory operations provides a powerful and intuitive way to expose patterns and discover bottlenecks, and is an important addition to existing statistical performance measurements.

Categories and Subject Descriptors (according to ACM CCS): I.6.9 [Simulation and Modeling]: Program Visualization

1. Introduction and Background

Processor performance is improving at a rate in which memory performance cannot keep up. As such, careful management of a program's memory usage is becoming more important in fields such as high-performance scientific computing. Memory optimizations are commonplace, however, the most efficient use of memory is not always obvious. Optimizing a program's memory performance requires integrating knowledge about algorithms, data structures, and CPU features. A deeper understanding of the program is required, beyond what simple inspection of source code, a debugger, or existing performance tools can provide. Attaining good performance requires the programmer to have a clear understanding of a program's memory transactions, and a way to analyze and understand them.

The deficit between processor and memory speeds has been increasing at an exponential rate, due to a differing rate of improvement in their respective technologies. The speed gap represents a "memory wall" [WM95] computer development will hit when the speed of computing becomes wholly determined by the speed of the memory subsystem. The primary mechanism for mitigating this diverging speed problem is the careful and efficient use of the cache, which works as fast temporary storage between main memory and the CPU. Current software practices, however, stress the value of abstraction; programmers should write correct code to accomplish their goals, and let the compiler and hardware handle performance issues. Unfortunately, not all optimizations can be accomplished solely by the compiler or hardware, and often the best optimizations, such as source code reorganization, can only be completed by the programmer [BDY02]. Therefore, optimizing high-performance software must involve the programmer, which in turn requires the programmer to have information about a program's interaction with memory and the cache.

This paper presents the Memory Trace Visualizer (MTV), a tool that enables interactive exploration of memory operations by visually presenting access patterns, source code tracking, cache internals, and global cache statistics. A screenshot of MTV can be seen in Figure 1. A *memory reference trace* is created by combining a trace of the memory operations and the program executable. The user then filters the trace by declaring memory regions of interest, typically main data structures of a program. This data is then used as input to the visualization tool, which runs a cache simulation, animates the memory accesses on the interesting regions, displays the effect on the whole address space, and provides user exploration through global and local navigation tools in time, space, and source code. By exploring code with MTV, programmers can better understand the memory performance of their programs, and discover new opportunities for performance optimization.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

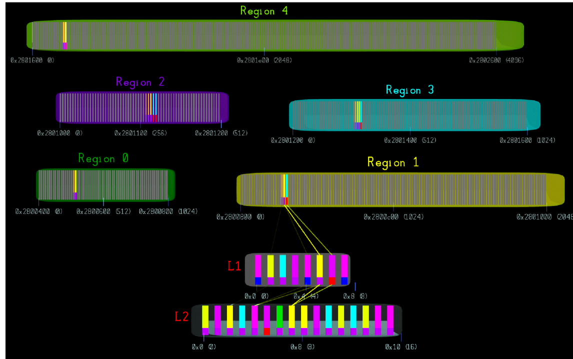

Figure 1: *Screenshot of the Memory Trace Visualizer.*

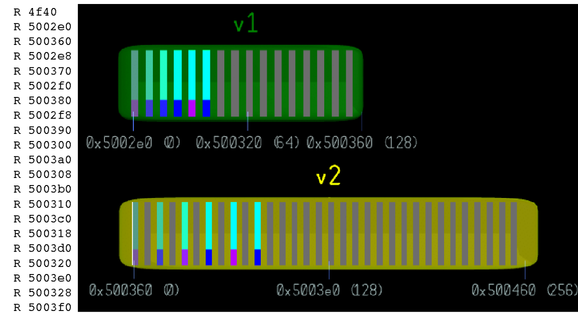

Figure 2: *A portion of a reference trace (left) and its visualization. The access pattern is stride-1 in region v1 (top), and stride-2 in region v2 (bottom).*

1.1. Cache Basics

A cache is fast, temporary storage, composed of several *cache levels*, each of which is generally larger, but slower than the last. In turn, each cache level is organized into *cache blocks* or *lines* which hold some specific, fixed number of bytes. A given memory reference *hits* if it is found in any level of the cache. More specifically, the reference *hits to Ln* when the appropriate block was found in the *n*th cache level. A reference *misses* if it is not found in a cache level, and must therefore retrieve data from main memory. When many references are hitting in a given level of the cache, that level is *warm*; conversely, if references miss often (or if the cache has not yet been used and thus contains no data from the process), it is *cold*. Collectively this quality of of a cache is called its *temperature*.

1.2. Memory Reference Traces

A *memory reference trace* is a sequence of records representing all memory references generated during a program's run. Each record comprises a code denoting the type of access ("R" for a read and "W" for a write) and the address at which the reference occurred. A reference trace carries all information about the program's interaction with memory and therefore lends itself to memory performance analysis. Figure 2, left, shows a small portion of an example trace file, which demonstrates that such a dataset is nearly impossible to inspect directly.

Collecting a reference trace for a program requires running the program, intercepting the load and store instructions, decoding them, and storing an appropriate reference record in a file. Several tools exist for this task. Pin [LCM*05] runs arbitrary binary instrumentation programs, including ones that can intercept load and store instructions, along with the target address. Apple provides the Computer Hardware Understanding Development (CHUD) Tools [App], which can generate instruction traces, and from them, reference traces. Our software examines both an instruction trace (as generated by CHUD) and the program executable, and produces a reference trace that includes source code line number information.

2. Related Work

To better understand performance, researchers have developed tools to provide analysis of overall program performance and the effects of program execution on caches, while cache and execution trace visualization methods provide insight into specific program behaviors.

2.1. Performance Analysis Tools

Shark [App] is Apple's runtime code profiler, from its CHUD suite, which collects information from hardware performance counters as a program runs. It also measures the amount of time spent by the program in each function and on each line of source code. All of this information is displayed textually, allowing a user to search for performance bottlenecks. While Shark does allow the user to count cache misses, it does not focus on the memory subsystem enough to enable a much deeper investigation into memory behavior.

VAMPIR [NAW*96] and TAU [SM06] are examples of systems that display the events encoded in an execution trace from a parallel program, while also computing and displaying associated performance measurements. Such tools operate at a high level, identifying bottlenecks in the communication patterns between computing nodes, for example. They do not observe low-level behavior of programs occurring at memory and thus occupy a role different from that of MTV.

2.2. Cache Simulation

The fundamental way to process a reference trace is to use it as input to a cache simulator [UM97], yielding miss rates for each cache level. Simulation gives a good first approximation to performance, but it *summarizes* the data in a trace rather than exposing concrete reasons for poor performance. Such a summary illustrates a reference trace's global behavior, but in order to understand a program's performance characteristics, programmers require more fine-grained detail, such as the actual *access patterns* encoded in the trace, as well as the specific, step-by-step effects these patterns cause in a simulated cache.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

Valgrind [NS07] is a framework for investigating runtime memory behavior, including a tool called Cachegrind, a cache simulator and profiler. Cachegrind runs a program and simulates its cache behavior, outputting the number of cache misses incurred by each line of program source code. While it provides useful information about local performance characteristics in a program, it does not generate a record of cache events that can be used to construct a visualization. To this end, we have written a special purpose cache simulator that describes the cache events occurring in each step of the simulation. With this information, we can perform step-by-step visualization of cache internals.

2.3. Cache Visualization

The Cache Visualization Tool [vdDKTG97] visualizes *cache block residency*, allowing the viewer to understand, for instance, competition amongst various data structures for occupancy in a specific level of the cache. KCacheGrind [WKT04] is a visual front end for Cachegrind, including visualizations of the call graph of a selected function, a tree-map relating nested calls, and details of costs associated with source lines and assembler instructions.

Cache simulation can be used to produce a static image representing cache events [YBH01]. For each time step, a pixel in the image is colored according to the status of the cache (blue for a hit, red for a miss, etc.). The resulting image shows a time profile for all the cache events in the simulation. This method visualizes the entire sequence of events occurring within the cache, which can lead to identification of performance bottlenecks.

YACO [QTK05] is a cache optimization tool that focuses on the statistical behavior of a reference trace. It counts cache misses and plots them in various ways, including time profiles. The goal is to direct the user to a portion of the trace causing a heavy miss rate. YACO also plots miss rate information with respect to individual data structures, demonstrating which areas in memory incur poor performance.

2.4. Execution Trace Visualization

Execution traces are related to reference traces but include more general information about a program's interaction with functional units of the host computer. JIVE [Rei03] and JOVE [RR05] are systems that visualize Java programs as they run, displaying usage of classes and packages, as well as thread states, and how much time is spent within each thread. These systems generate trace data directly from the running program and process it on the fly, in such a way as to minimize runtime overhead. Stolte et al. [SBHR99] demonstrate a system that visualizes important processor internals, such as pipeline stalls, instruction dependencies, and the contents of the reorder buffer and functional units.

3. Memory Reference Trace Visualization

The novelty of the memory reference trace visualization presented in this work lies in the display of *access patterns* as they occur in user-selected regions of memory. Much of the previous work focuses on cache behavior and performance, and while this information is incorporated as much as possible, the main focus is to provide an understanding of specific memory regions. To this end, MTV provides the user with an animated visualization of region and cache behavior, global views in both space and time, and multiple methods of navigating the large dataspace.

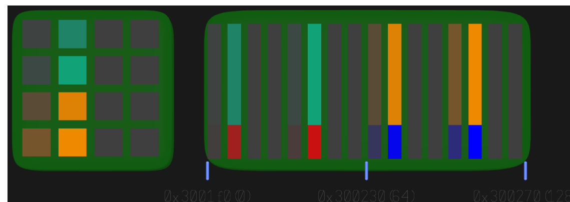

Figure 3: *A single memory region visualized as a linear sequence in memory (right) and as a 2D matrix (left). The read and write accesses are indicated by coloring the region line cyan or orange. Corresponding cache hits and misses are displayed in blue and red. Fading access lines indicate the passage of time.*

3.1. System Overview

MTV's fundamental goal is to intelligibly display the contents of a reference trace. To this end, MTV creates on-screen maps of interesting regions of memory, reads the trace file, and posts the read/write events to the maps as appropriate. In addition, MTV provides multiple methods of orientation and navigation, allowing the user to quickly identify and thoroughly investigate interesting memory behaviors.

The input to MTV is a reference trace, a *registration file*, and cache parameters. A registration file is a list of the regions in memory a user wishes to focus on and is produced when the reference trace is collected, by instrumenting the program to record the address ranges of interesting memory regions. A program can register a region of memory by specifying its base address, size, and the size of the datatype occupying the region. The registration serves to filter the large amount of data present in a reference trace by framing it in terms of the user-specified regions. For the cache simulation, the user supplies the appropriate parameters: the cache block size in bytes, a write miss policy (i.e., write allocate or write no-allocate), a page replacement policy (least recently used, FIFO, etc.), and for each cache level, its size in bytes, its set associativity, and its write policy (write through or write back) [HP03].

3.2. Visual Elements

MTV's varied visual elements work together to visualize a reference trace. Some of these elements directly express data coming from the trace, while others provide context for the user.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

3.2.1. Data Structures

MTV displays a specified region as a linear sequence of data items, surrounded by a background shell with a unique, representative color (Figure 3, right). Read and write operations highlight the corresponding memory item in the region using cyan and orange, colors chosen for their distinguishability. A sense of the passage of time arises from gradually fading the colors of recently accessed elements, resulting in "trails" that indicate the direction in which accesses within a region are moving. Additionally, the result of the cache simulation for each operation is shown in the lower half of the glyph, using a red to blue colormap (see Section 3.2.3).

To further aid in the understanding of the program, the region can be displayed in a 2D configuration, representing structures such as C-style 2D arrays, mathematical matrices, or a simulation of processes occurring over a physical area (Figure 3, left). The matrix modality can also be used to display an array of C-style structs in a column, the data elements of each struct spanning a row. This configuration echoes the display methods of the linear region, with read and write operations highlighting memory accesses. The matrix glyph's shell has the same color as its associated linear display glyph, signifying that the two displays are redundant views of the same data.

3.2.2. Address Space

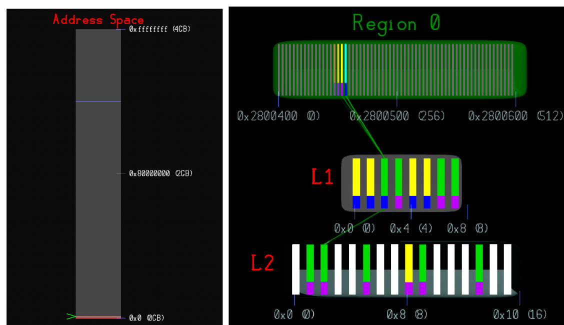

Figure 4: *Left: A visualization of the entire process address space. Right: A single memory region and the cache (labeled L1 and L2).*

By also visualizing accesses within a process address space, MTV offers a global analog to the region views (Figure 4, left). As accesses light up data elements in the individual regions in which they occur, they also appear in the much larger address space that houses the entire process. In so doing, the user can gain an understanding of more global access patterns, such as stack growth due to a deep call stack, or runtime allocation and initialization of memory on the heap. On a 32 bit machine, the virtual address space occupies 4GB of memory; instead of rendering each byte of this range as the local region views would do, the address space view approximates the position of accesses within a linear glyph representing the full address space.

3.2.3. Cache View

In addition to displaying a trace's access patterns, MTV also performs cache simulation with each reference record and displays the results in a schematic view of the cache. As the varying cache miss rate serves as an indicator of memory performance, the cache view serves to connect memory access patterns to a general measure of performance. By showing how the cache is affected by a piece of code, MTV allows the user to understand what might be causing problematic performance.

The visualization of the cache is similar to that of linear regions with cache blocks shown in a linear sequence surrounded by a colored shell (Figure 4). The cache is composed of multiple levels, labeled L1 (the smallest, fastest level) through L*n* (the largest, slowest level). The color of the upper portion of the cache blocks in each level corresponds to the identifying color of the region which last accessed that block, or a neutral color if the address does not belong to any of the user-declared regions. The cache hit/miss status is indicated in the bottom portion of the memory blocks by a color ranging from blue to red—blue for a hit to L1, red for a cache miss to main memory, and a blend between blue and red for hits to levels slower than L1. To emphasize the presence of data from a particular region in the cache, lines are also drawn between the address in the region view and the affected blocks in the cache. Finally, the shells of each cache level reflect the cache temperature: the warmer the temperature, the brighter the shell color.

3.3. Orientation and Navigation

Combining a cache simulation with the tracking of memory access patterns creates a large, possibly overwhelming amount of data. Reducing the visualized data to only important features, providing useful navigation techniques, as well as relating events in the trace to source code is very important to having a useful tool.

3.3.1. Memory System Orientation

The first step in managing the large dataset is to let the user filter the data by registering specific memory regions (for example, program data structures) to visualize. During instrumentation, there is no limit on the number of memory regions that can be specified, although in visualization, the screen space taken by each region becomes a limitation. To ease this problem, the user is given the freedom to move the regions anywhere on the screen during visualization. Clicking on a individual region makes that region *active*, which brings that region to the forefront, and lines that relate the memory locations of that region to locations in the cache are drawn (Figure 4, right).

3.3.2. Source Code Orientation

MTV also highlights the line of source code that corresponds to the currently displayed reference record (Figure 5), of-

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

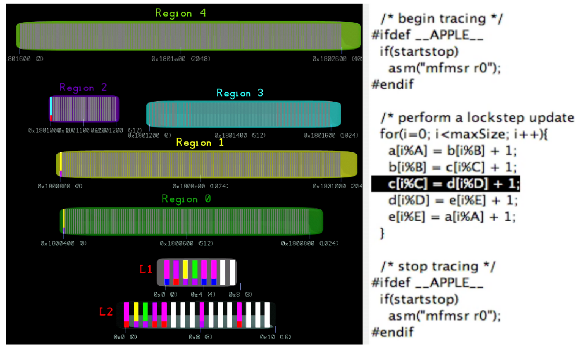

Figure 5: *The source code corresponding to the current memory access is highlighted, providing a direct relationship between memory access and source code.*

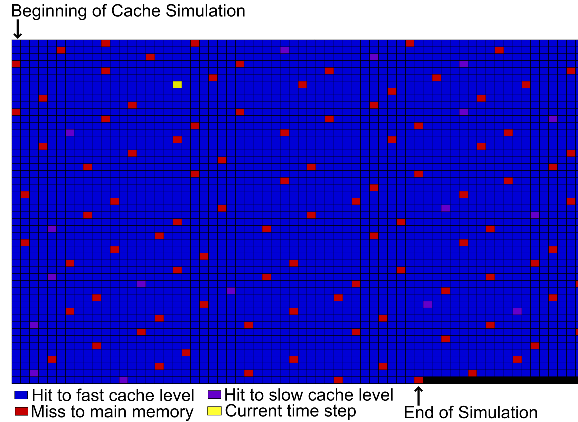

Figure 6: *The cache event map provides a global view of the cache simulation by showing the cache status for each time step, and a clickable time navigation interface. The time dimension starts at the upper left of the map and proceeds across the image in English text order.*

fering a familiar, intuitive, and powerful way to orient the user, in much the same way as a traditional debugger such as GDB. This provides an additional level of context in which to understand a reference trace. Source code directly expresses a program's intentions; by correlating source code to reference trace events, a user can map code abstractions to a concrete view of processor-level events.

Generally, programmers do not think about how the code they write effects physical memory. This disconnect between coding and the memory system can lead to surprising revelations when exploring a trace, creating a better understanding of the relationship between coding practices and performance. For example, in a program which declares an C++ STL vector, initializes the vector with some data, and then proceeds to sum all the data elements, one might expect to see a sweep of writes representing the initialization followed by reads sweeping across the vector for the summation. However, MTV reveals that before these two sweeps occur, an initial sweep of writes moves all the way across the vector. The source code viewer indicates that this view occurred at the line declaring the STL vector. Seeing the extra write reminds the programmer that the STL *always* initializes vectors (with a default value if necessary). The source code may fail to explicitly express such behavior (as it does in this example), and often the behavior may appreciably impact performance. In this example, MTV helps the programmer associate the abstraction of "STL vector creation" to the concrete visual pattern of "initial write-sweep across a region of memory."

3.3.3. Time Navigation and the Cache Event Map

Because reference traces represent events in a time series and MTV uses animation to express the passage of time, only a very small part of the trace is visible on-screen at a given point. To keep users from becoming lost, MTV includes multiple facilities for navigating in time. The most basic time navigation tools include play, stop, rewind and fast forward buttons to control the simulation. This allows users to freely move through the simulation, and revisit interesting time steps.

The cache event map is a global view of the cache simulation, displaying hits and misses in time, similar to the technique of Yu et al. [YBH01]. Each cell in the map represents a single time step, unrolled left to right, top to bottom. The color of each cell expresses the same meaning as the blue-to-red color scale in the cache and region views (see Section 3.2.3). A yellow cursor highlights the current time step of the cache simulation. By unrolling the time dimension (normally represented by animation) into screen space, the user can quickly identify interesting features of the cache simulation. In addition, the map is a clickable global interface, taking the user to the time step in the simulation corresponding to the clicked cell.

4. Examples

The following examples demonstrate how MTV can be used to illuminate performance issues resulting from code behavior. For the first example, a simple cache is simulated: The block size is 16 bytes (large enough to store four single-precision floating point numbers); L1 is two-way set associative, with four cache blocks; L2 is eight-way set associative, with eight cache blocks. In the second example, the cache has the same block size but twice as many blocks in each level. These caches simplify the demonstrations, but much larger caches can be used in practice. The third example simulates the cache found in a PowerMac G5. It has a block size of 128 bytes; the 32K L1 is two-way set associative, and the 512K L2 is eight-way set associative.

4.1. Loop Interchange

A common operation in scientific programs is to make a pass through an array of data and do something with each data

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

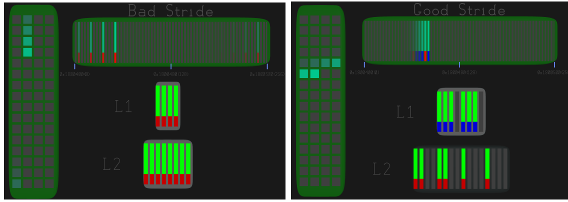

Figure 7: *Striding a two dimensional array in different orders produces a marked difference in performance.*

item. Often, the data are organized in multi-dimensional arrays; in such a case, care must be taken to access the data items in a cache-friendly manner. Consider the following two excerpts of C code:

```
/* Bad Stride (before) */
double sum = 0.0;
for(j=0; j<4; j++)
    for(i=0;  i<32;  i++)
        sum += A[i][j];
```

```
/* Good Stride (after) */
double sum = 0.0;
for(i=0; i<32; i++)
  for(j=0; j <4; j++)
    sum += A[i][j];
```

The illustrated transformation is called *loop interchange* [HP03], because it reorders the loop executions. Importantly, the semantics of the two code excerpts are identical, although there is a significant difference in performance between them.

The above source code demonstrates how MTV visualizes the effect of the code transformation (Figure 4.1). In each case, the A array is displayed both as a single continuous array (as it exists in memory) and as a 2D array (as it is conceptualized by the programmer). The "Bad Stride" code shows a striding access pattern resulting from the choice of loop ordering, while the "Good Stride" code shows a more reasonable continuous access pattern.

The "Bad Stride" code exhibits poor performance because of its lack of data reuse. As a data item is referenced, it is loaded into the cache along with the data items adjacent to it (since each cache block holds four floats); however, by the time the code references the adjacent items, they have been flushed from the cache by the intermediate accesses. Therefore, the code produces a cache miss on every reference. The "Good Stride" code, on the other hand, uses the adjacent data immediately, increasing cache reuse and thereby eliminating three quarters of the cache misses.

MTV flags the poor performance in two ways. First, the poor striding pattern is visually apparent: the accesses do not sweep continuously across the region, but rather make multiple passes over the array, skipping large amounts of space each time. Because the code represents a single pass through the data, the striding pattern immediately seems inappropriate. Second, the cache indicates that misses occur on every access: the shell of the cache glyph stays black, and therefore cold, throughout the run. The transformed code, on the other hand, displays the expected sweeping pattern, and the cache stays warm.

4.2. Matrix Multiplication

Matrix multiplication is another common operation in scientific programs. The following algorithm shows a straightforward multiplication routine:

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++){
    r = 0.0;
    for(k=0; k<N; k++)
      r += Y[i*N+k] * Z[k*N+j];
    X[i*N+j] = r;
   }
```

MTV shows the familiar pattern associated with matrix multiplication by the order in which the accesses to the X, Y, and Z matrices occur (Figure 8, top). The troublesome access pattern in this reference trace occurs in matrix Z, which must be accessed column-by-column because of the way the algorithm runs.

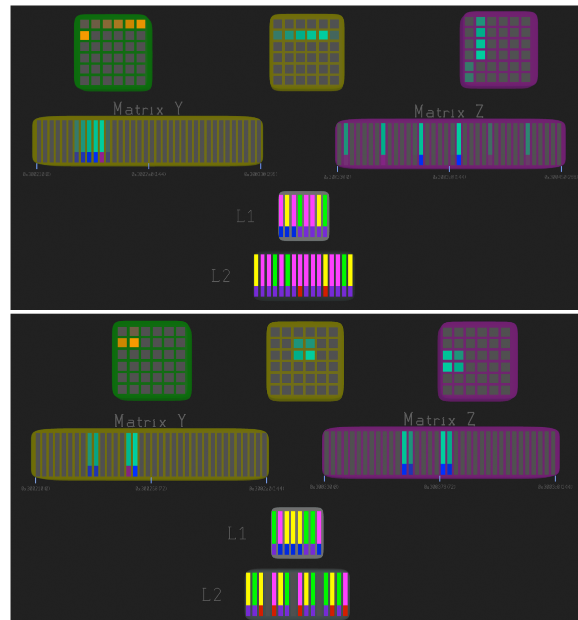

Figure 8: *Naive matrix multiply (top) and blocked matrix multiply (bottom).*

In order to rectify the access pattern, the programmer may transform the code to store the transpose of matrix Z. Then, to perform the proper multiplication, Z would have to be accessed in row-major order, eliminating the problematic access pattern. When certain matrices always appear first in a matrix product and others always appear second, one possible solution is to store matrices of the former type in row-major order and those of the latter type in column-major order. In this example, the visual patterns encoded in the trace (Figure 8, top), suggested a code transformation. This transformation also suggests a new abstraction of left- vs. right-multiplicand matrices that may help to increase the performance of codes relying heavily on matrix multiplication. A

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

more general solution to improving matrix multiplication is widely known as *matrix blocking*, in which algorithms operate on small submatrices that fit into cache, accumulating the correct answer by making several passes (Figure 8, bottom).

4.3. Material Point Method

A more complex, real-world application of MTV is in investigating the Material Point Method (MPM) [BK04], which simulates rigid bodies undergoing applied forces by treating a solid object as a collection of particles, each of which carries information about its own mass, velocity, stress, and other physical parameters. A simulation is run by modeling an object and the forces upon it, then iterating the MPM algorithm over several time steps.

Because each material point is associated with several data values, the concept of a particle maps evenly to a C-style struct or C++-style class. The collection of particles can then be stored in an array of such structures. Accessing particle values is as simple as indexing the array to select a particle, and then naming the appropriate field. Although this design is straightforward for the programmer, the scientific setting around MPM demands high performance.

MTV's visualization of a run of MPM code with the array-of-structs storage policy demonstrates how the policy might cause suboptimal performance (Figure 9, top). The region views show that the access pattern is broken up over the structs representing each particle, so that the same parts of each struct are visited in a sort of lockstep pattern. Though these regions are displayed in MTV as separate entities, they are in fact part of the same contiguous array in memory; in other words, the access pattern is related to the poorly striding loop interchange example (Section 4.1). The visual is confirmed by the MPM algorithm: in the first part of each time step, the algorithm computes a momentum value by looking at the mass and velocity of each particle (in Figure 9, top, the single lit value at the left of each region is the mass value, while the three lit values to the right of the mass comprise the velocity). In fact, much of the MPM algorithm operates this way: values of the same type are needed at roughly the same time, rather than each particle being processed in whole, one at a time.

MTV demonstrates a feature of the MPM implementation that is normally hidden: the chosen storage policy implies a necessarily non-contiguous access pattern. The simplest way to rearrange the storage is to use parallel arrays instead of an array of structs, so that all the masses are found in one array, the velocities in another, and so on. Grouping similar values together gives the algorithm a better chance of finding the next values it needs nearby. This storage policy results in a more coherent access pattern, and higher overall performance (Figure 9, bottom).

This particular observation and the simple solution it suggests are both tied to our understanding of the algorithm. By making even more careful observations, it should be possible to come up with a hybrid storage policy that respects more of the algorithm's idiosyncrasies and achieves higher performance. The example also stresses the value of abstraction, and in particular, the value of separating interfaces from implementations. By having an independent interface to the particle data (consisting of functions or methods with signatures like `double getMass(int particleId);`), the data storage policy is hidden appropriately and can vary freely for performance or other concerns.

5. Conclusions and Future Work

The gap between processor and memory performance will be a persistent problem for memory-bound applications until major changes are made in the memory paradigm. We have described a tool that is designed to explicitly examine the interaction between a program and memory through visualization of detailed reference traces. Our work provides a technique for the rich yet inscrutable reference trace data by offering visual metaphors for abstract memory operations, leading to a deeper understanding of memory usage and therefore opportunities for optimization.

In the future, we hope to mature our techniques by making them more automatic; we want to make the process of collecting, storing, and analyzing a reference trace transparent to a user, so that MTV can become as useful as interactive debuggers are today. A way to reduce or eliminate the need for runtime instrumentation (or at least, render it completely transparent) would help meet this goal, for instance.

We are also seeing a relatively new trend in computing—multicore machines are on the rise, and programmers are struggling to understand how to use them effectively. To fully realize their potential, we need ways to keep all of the cores fed with data: it is a central problem, and as yet, an unsolved one. We believe visualization and analysis tools in the spirit of MTV have an important place among multicore programming techniques.

Whether processors continue to get faster, or more of them appear in single machines (or both), memory will always be a critical part of computer systems, and its careful use will be critical to high-performance software. We hope that MTV and the ideas behind it can help keep the growing complexity of computer systems manageable.

References

[App] APPLE CORPORATION: Performance and debugging tools overview. http://developer.apple.com/tools/performance/overview.html.

[BDY02] BEYLS K., D'HOLLANDER E. H., YU Y.: Visualization enables the programmer to reduce cache

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

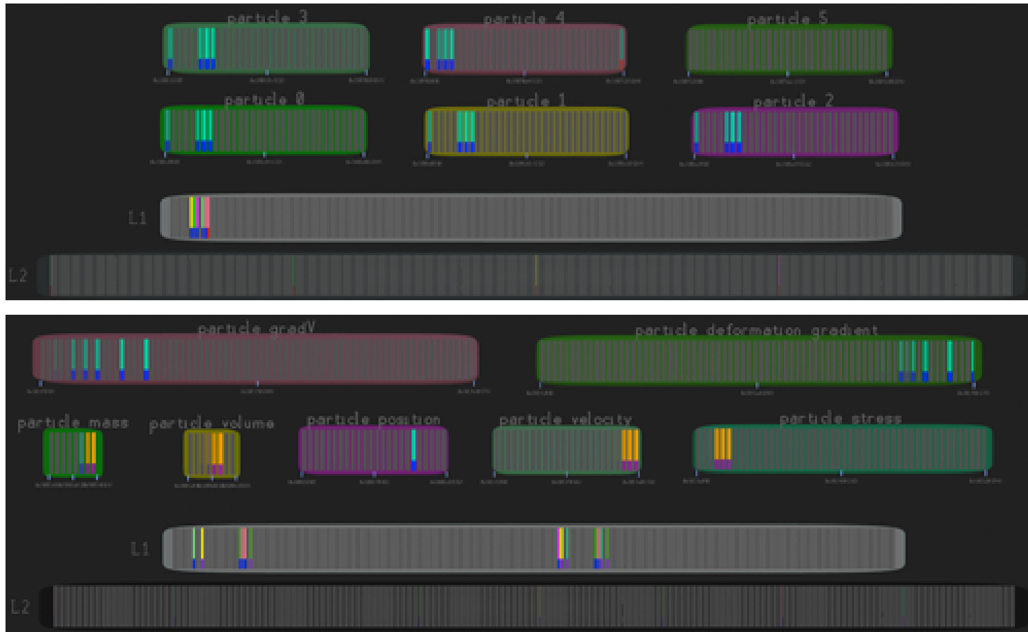

Figure 9: *MPM Horizontal (top) and Vertical (bottom).*

misses. In *IASTED Conference on Parallel and Distributed Computing and Systems* (Nov 2002), pp. 781–786.

[BK04] BARDENHAGEN S. G., KOBER E. M.: The generalized interpolation material point method. *CMES 5*, 6 (2004), 477–495.

[HP03] HENNESSY J. L., PATTERSON D. A.: *Computer Architecture: A Quantitative Approach*, third ed. Morgan Kaufmann Publishers, 2003.

[LCM*05] LUK C.-K., COHN R., MUTH R., PATIL H., KLAUSER A., LOWNEY G., WALLACE S., REDDI V. J., HAZELWOOD K.: Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[NAW*96] NAGEL W. E., ARNOLD A., WEBER M., HOPPE H.-C., SOLCHENBACH K.: VAMPIR: Visualization and analysis of mpi resources. *Supercomputer 12*, 1 (1996), 69–80.

[NS07] NETHERCOTE N., SEWARD J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation* (June 2007).

[QTK05] QUAING B., TAO J., KARL W.: YACO: A user conducted visualization tool for supporting cache optimization. In *Proceedings of HPCC* (2005), pp. 694–603.

[Rei03] REISS S. P.: Visualizing java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization* (New York, NY, USA, 2003), ACM, pp. 57–ff.

[RR05] REISS S. P., RENIERIS M.: Jove: java as it happens. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization* (New York, NY, USA, 2005), ACM, pp. 115–124.

[SBHR99] STOLTE C., BOSCH R., HANRAHAN P., ROSENBLUM M.: Visualizing application behavior on superscalar processors. In *INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization* (1999), pp. 10–17.

[SM06] SHENDE S. S., MALONY A. D.: The TAU parallel performance system. *International Journal of High Performance Computing Applications 20*, 2 (2006), 287–331.

[UM97] UHLIG R. A., MUDGE T. N.: Trace-driven memory simulation: A survey. *ACM Computing Surveys 29*, 2 (June 1997), 128–170.

[vdDKTG97] VAN DER DEIJL E., KANBIER G., TEMAM O., GRANSTON E. D.: A cache visualization tool. *Computer 30*, 7 (July 1997), 71–78.

[WKT04] WEIDENDORFER J., KOWARSCHIK M., TRINITIS C.: A tool suite for simulation based analysis of memory access behavior. *ICCS 3038 of LNCS* (2004), 440–447.

[WM95] WULF W. A., MCKEE S. A.: Hitting the memory wall: Implications of the obvious. *Computer Architecture News 23*, 1 (1995), 20–24.

[YBH01] YU Y., BEYLS K., HOLLANDER E. H. D.: Visualizing the impact of the cache on program execution. In *Proceedings of the Fifth International Conference on Information Visualisation* (July 2001), pp. 336–341.

© 2008 The Author(s)
Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.