UNNAMED PHD THESIS

by

A.N.M. Imroz Choudhury

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing

The University of Utah

March 2012

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

A.N.M. Imroz Choudhury

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Paul A. Rosen, Steven G. Parker

David M. Beazley

Erik L. Brunvand

Christopher R. Johnson

Mike Kirby

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of A.N.M. Imroz Choudhury in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Paul A. Rosen, Steven G. Parker
Chair, Supervisory Committee

Approved for the Major Department

Al Davis
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

Signal relay and adaptation in response to cAMP stimuli in the cellular slime mold *Dictyostelium discoideum* are a model system for the study of signal transduction. Calcium dynamics in different cell types, especially in deutosome eggs and cardiac cells, have attracted a lot of interest lately. This thesis attempts to develop some general theories about second messenger dynamics.

To my parents

Hello

CONTENTS

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to acknowledge joint work with coauthor Hans G. Othmer, part of which appears with minor modification as Chapter 2 and Chapter 4. These materials are included with permission of the publishers.

Professor Kenneth W. Spitzer and Mr. Timothy J. Lewis provided me with much information and fruitful discussions during the development of the models presented in Chapter 6. I also want to express my deep appreciation for the patient help received from Dr. Nelson H. F. Beebe and Mr. Pieter J. Bowman on my computational work and the preparation of the manuscript.

The research in this thesis was supported in part under NIH Grant #GM123456.

CHAPTER 1

INTRODUCTION

1.1 Outline

oposal intro for extra ideas

- Abstraction enables productivity, greatness—sometimes have to peel abstraction layers back to inspect implementation layers.
- If "correctness" includes "runs in a short time", for instance, cannot blindly rely on abstractions.
- One major abstraction—memory as linear array of addressable elements—hides a lot of machinery (cache) and circuitry in name of complicity, uniformity
- Can represent failure of performance
  - Poor use of cache in single-threaded process.
  - Cache invalidation protocol in multi-threaded process.
  - Ignorant remote access (done transparently via access abstraction) in NUMA machines.
  - Other architectures—poor communication patterns in MPI programs; too much CPU-to-GPU transfer, etc., user of memory on GPUs.
- Memory behavior—*simple rules* give rise to *complex behavior*, yet important for *correctness* and *performance.*
- Analysis/simulation for gross numerical results (refer to Christiaan chapter, if possible).

- *Visualization* for understanding events—*reasons* for poor cache behavior.
- **Thesis statement**
  - Notes for construction:
    - * Vis makes activity *concrete* in the form of *visible patterns*, granting *deeper understanding* of program/memory behavior, and enabling both the display of petential memory performance problems, and in many cases suggesting possible solutions.

1.2 Dissertation Roadmap

1.2.1 MTV Paper

- “Literal” visualization—sets *baseline* for all vis work/analysis.
- Literal metaphors, w/only “array” abstraction (matrix as well—FUTURE WORK IDEA: incorporate other geometry into visual encoding—circle, etc.)
- see patterns
- (other summary points from paper)
- Introduces cache simulation as way to drive analysis/visualization (cache behavior is fundamental to understanding program performance).

1.2.2 Waxlamp Paper

- Shed “literal” view in favor of more abstraction.
- Shift setting from memory to cache—retain abstraction of “placement” in memory (i.e. in the outer ring of the radial display).
- Access patterns still visible, but deemphasized.
- *New focus*—data motion within cache.
- New patterns:

  - ∗ bulk miss rates localized in time
  - ∗ eviction order visible
  - ∗ program idioms—"sweeping", "pounding (on a short stretch of memory)" become visible

– Abstract design lends itself to other devices, scales (GPU? MPI? other onboard systems such as I/O? etc.).

1.2.3 Topology Paper

– Forget about cache simulation.

– Focus on *reference trace.*

– Trace is linear, yet program flow (and programs themselves) are non-linear.

– *Recurrence* important in understanding memory behavior—maps to program structure at multiple scales, and to operations that hardware can try to optimize (pre-fetching), and to similar patterns across time or space (restructuring of programs for better cache use, vectorization, etc.).

– Use topology to extract circulat structure, aka recurrent behavior.

– Yields visualization laying out linear trace out in space, encoding time in a spatial dimension also.

– New insight into trace behavior (examples from paper).

– Ancillary vis for other methods (add cache simulation back in, or use as global "map" for other methods, etc.).

1.2.4 Uncertainty Paper

– Longer view—look at multiple traces to yield information from their ensemble behavior.

– Vary cache features to learn about *algorithms.*

sion—Vis workshop submission)

- Model higher-level phenomena? (future work)

CHAPTER 2

BACKGROUND

Lorem ipsum blah blah blah.

CHAPTER 3

RELATED WORK

Lorem ipsum blah blah blah.

CHAPTER 4

VISUALIZING REFERENCE TRACES WITH MTV

Lorem ipsum blah blah blah.

4.1 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" [31] 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 [5]. 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 4.1. 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.

4.1.1 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 nth cache level. A reference *misses* if it is not found in a cache level, and must therefore retrieve data from main

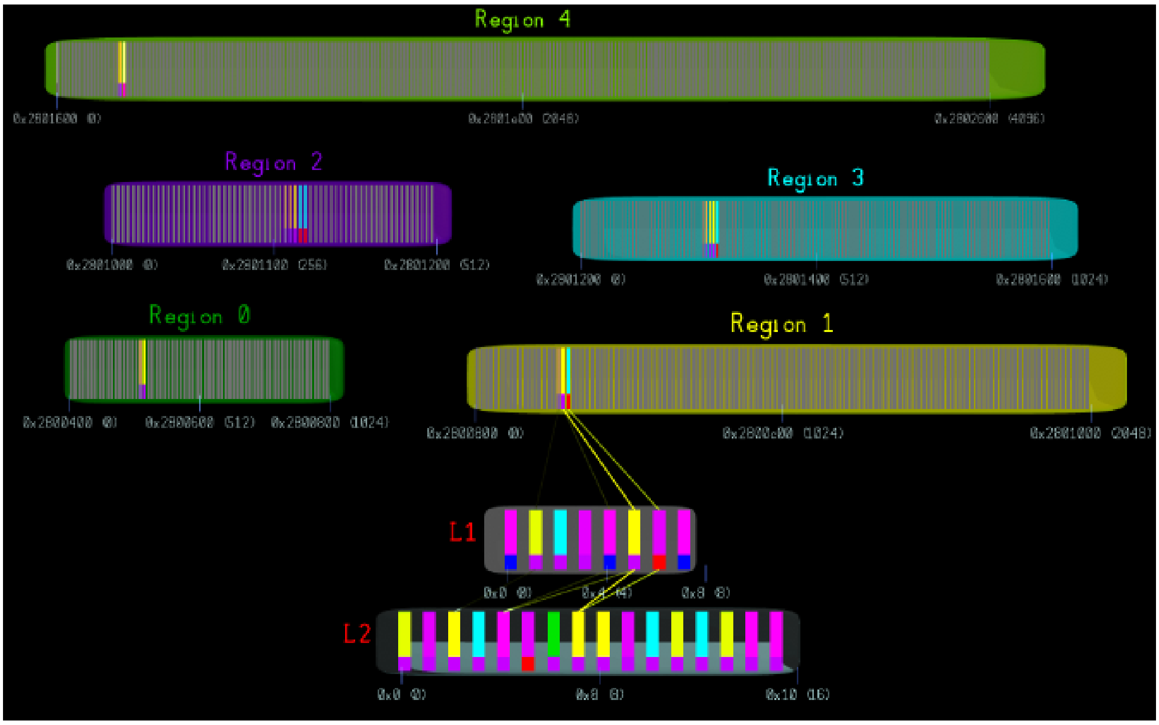


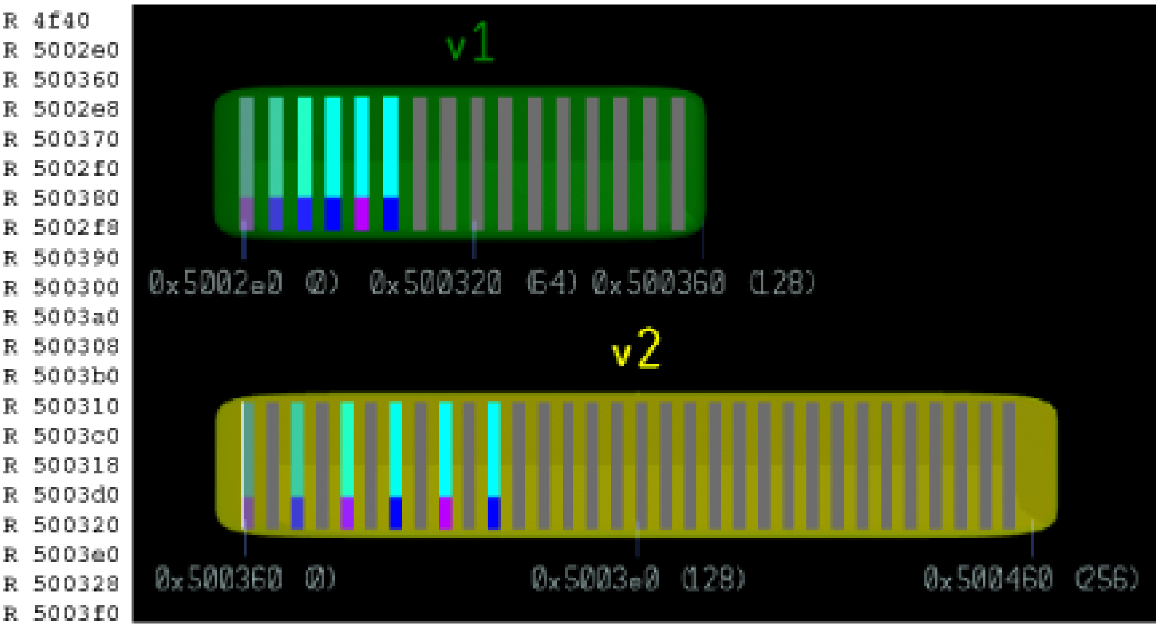
Figure 4.1. Screenshot of the Memory Trace Visualizer.

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*.

4.1.2 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 4.2, 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,



```
R 4f40
R 5002e0
R 500360
R 5002e8
R 500370
R 5002f0
R 500380
R 5002f8
R 500390
R 500300
R 5003a0
R 500308
R 5003b0
R 500310
R 5003c0
R 500318
R 5003d0
R 500320
R 5003e0
R 500328
R 5003f0
```

Figure 4.2. 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).

intercepting the load and store instructions, decoding them, and storing an appropriate reference record in a file. Several tools exist for this task. Pin [11] 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 [2], 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.

4.2 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.

4.2.1 Performance Analysis Tools

Shark [2] 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 [13] and TAU [21] 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.

4.2.2 Cache Simulation

The fundamental way to process a reference trace is to use it as input to a cache simulator [26], 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.

Valgrind [15] 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.

4.2.3 Cache Visualization

The Cache Visualization Tool [28] 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 [30] 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 [33]. 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 [17] 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.

4.2.4 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 [19] and JOVE [20] 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. [23] demonstrate a system that visualizes important processor internals, such as pipeline stalls, instruction dependencies, and the contents of the reorder buffer and functional units.

4.3 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.

4.3.1 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.

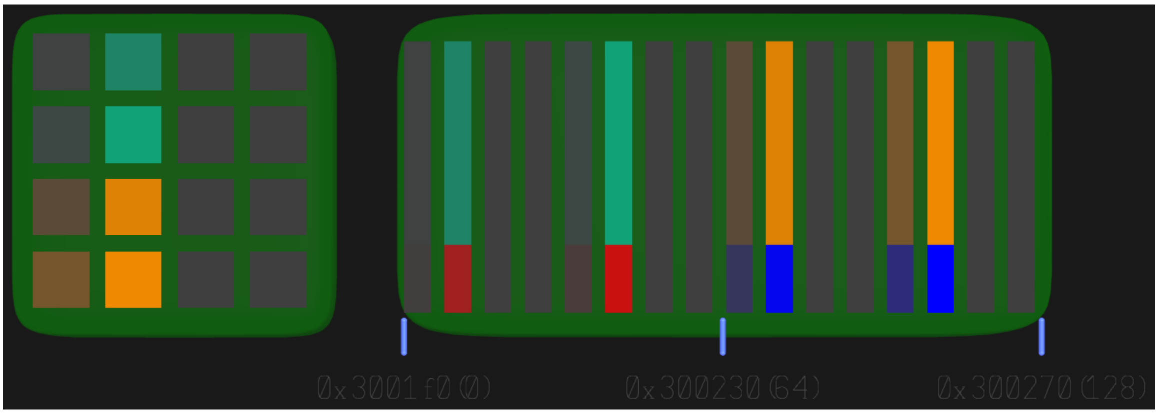


Figure 4.3. A single memory region visualized as a linear sequence in memory (right) and as a 2D matrix (left). The read and write accesses are indicated by coloring the region line cyan or orange. Corresponding cache hits and misses are displayed in blue and red. Fading access lines indicate the passage of time.

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) [10].

4.3.2 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.

4.3.2.1 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 4.3, 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 4.3.2.3).

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 4.3, 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.

4.3.2.2 Address Space

By also visualizing accesses within a process address space, MTV offers a global analog to the region views (Figure 4.4, 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.

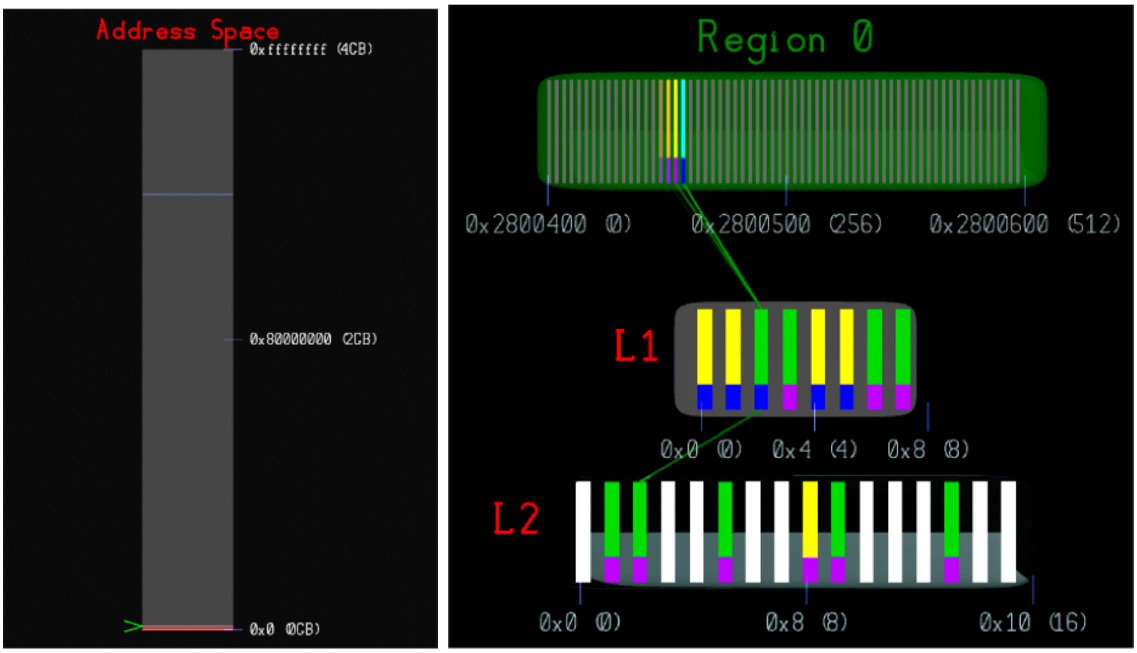


Figure 4.4. Left: A visualization of the entire process address space. Right: A single memory region and the cache (labeled L1 and L2).

4.3.2.3 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 4.4). The cache is composed of multiple levels, labeled L1 (the smallest, fastest level) through Ln (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.

4.3.3 Orientation and Navigation

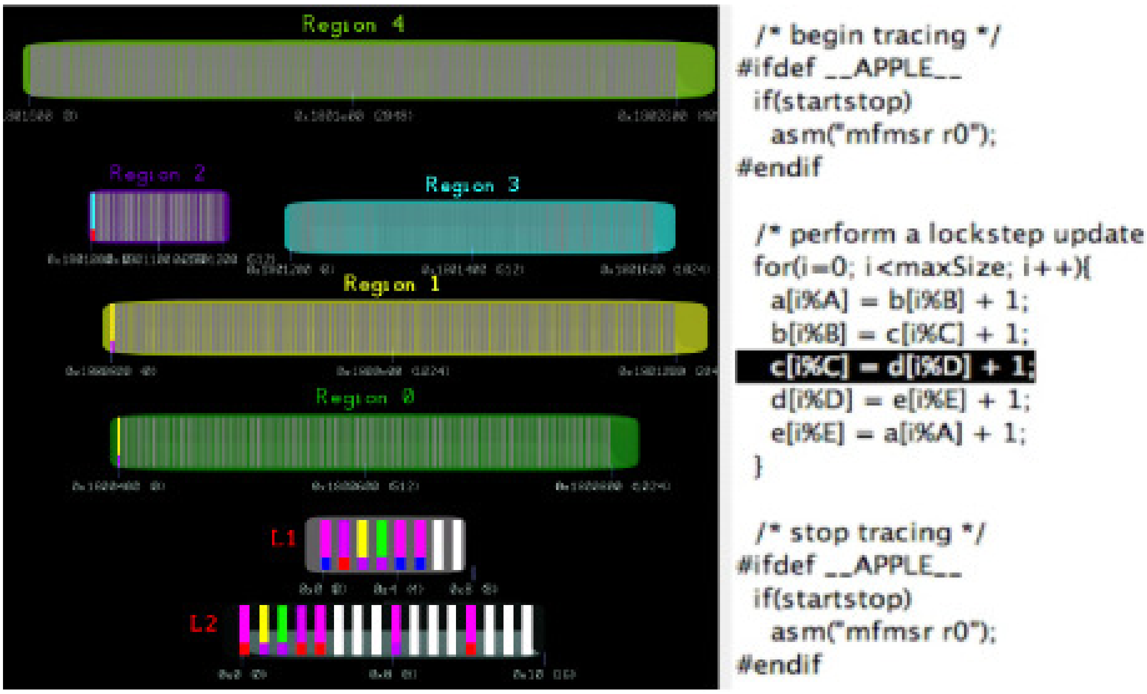
Combining a cache simulation with the tracking of memory access patterns creates a large, possibly overwhelming amount of data. Reducing the visualized data to only important features, providing useful navigation techniques, as well as relating events in the trace to source code is very important to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

4.3.3.1 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 4.4, right).

4.3.3.2 Source Code Orientation

MTV also highlights the line of source code that corresponds to the currently displayed reference record (Figure 4.5), offering a familiar,



```
 /* begin tracing */
#ifdef __APPLE__
 if(startstop)
   asm("mfmsr r0");
#endif

 /* perform a lockstep update
 for(i=0; i<maxSize; i++){
   a[i%A] = b[i%B] + 1;
   b[i%B] = c[i%C] + 1;
   c[i%C] = d[i%D] + 1;
   d[i%D] = e[i%E] + 1;
   e[i%E] = a[i%A] + 1;
 }

 /* stop tracing */
#ifdef __APPLE__
 if(startstop)
   asm("mfmsr r0");
#endif
```

Figure 4.5. The source code corresponding to the current memory access is highlighted, providing a direct relationship between memory access and source code.

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."

4.3.3.3 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. [33]. 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 4.3.2.3). 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.

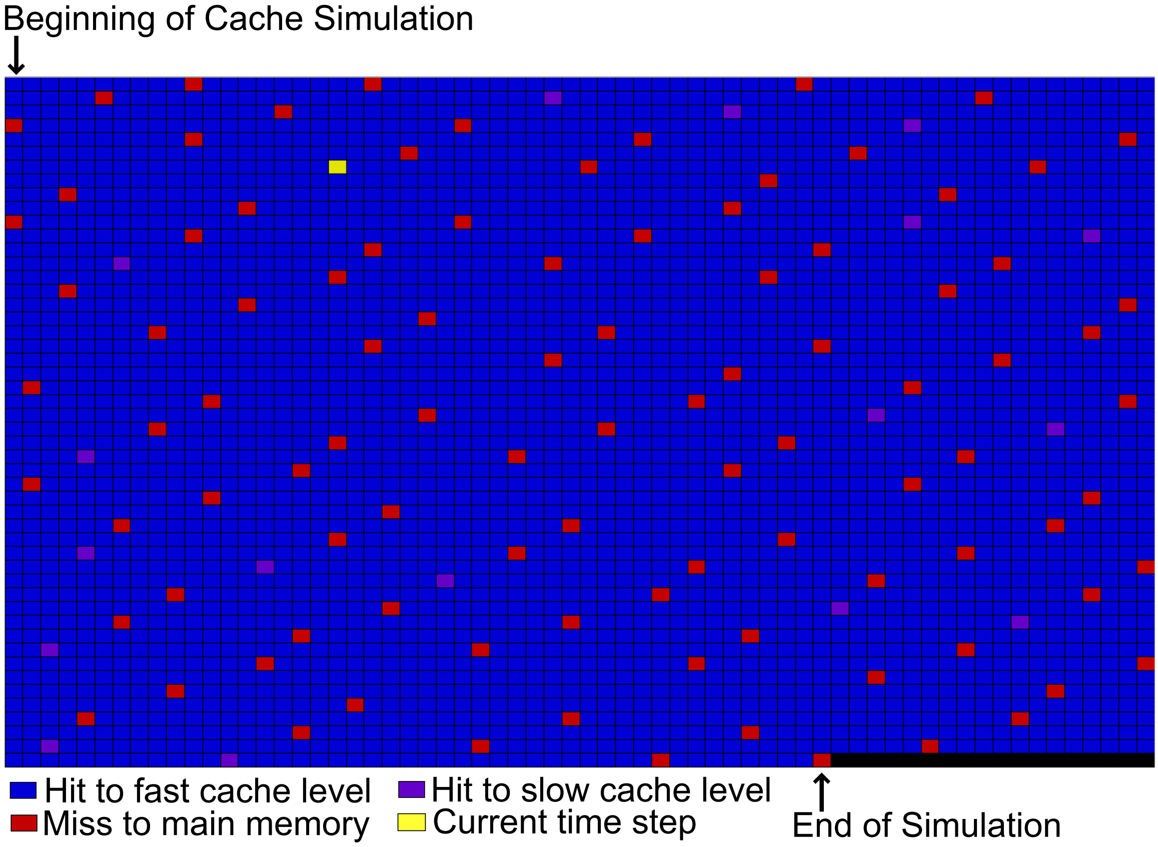


Figure 4.6. The cache event map provides a global view of the cache simulation by showing the cache status for each time step, and a clickable time navigation interface. The time dimension starts at the upper left of the map and proceeds across the image in English text order.

4.4 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.

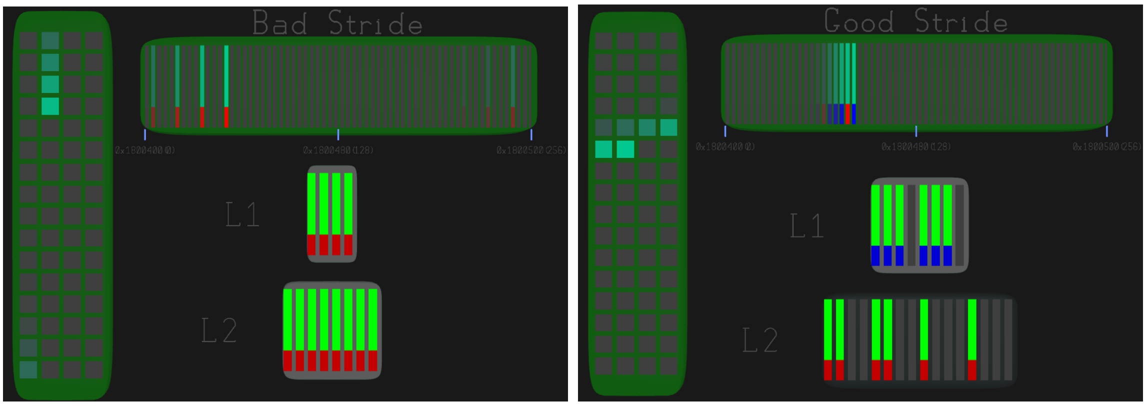


Figure 4.7. Striding a two dimensional array in different orders produces a marked difference in performance.

4.4.1 Loop Interchange

A common operation in scientific programs is to make a pass through an array of data and do something with each data 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* [10], 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 4.4.1). 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.

4.4.2 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 4.8, 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.

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 4.8, 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 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 4.8, bottom).

4.4.3 Material Point Method

A more complex, real-world application of MTV is in investigating the Material Point Method (MPM) [3], 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

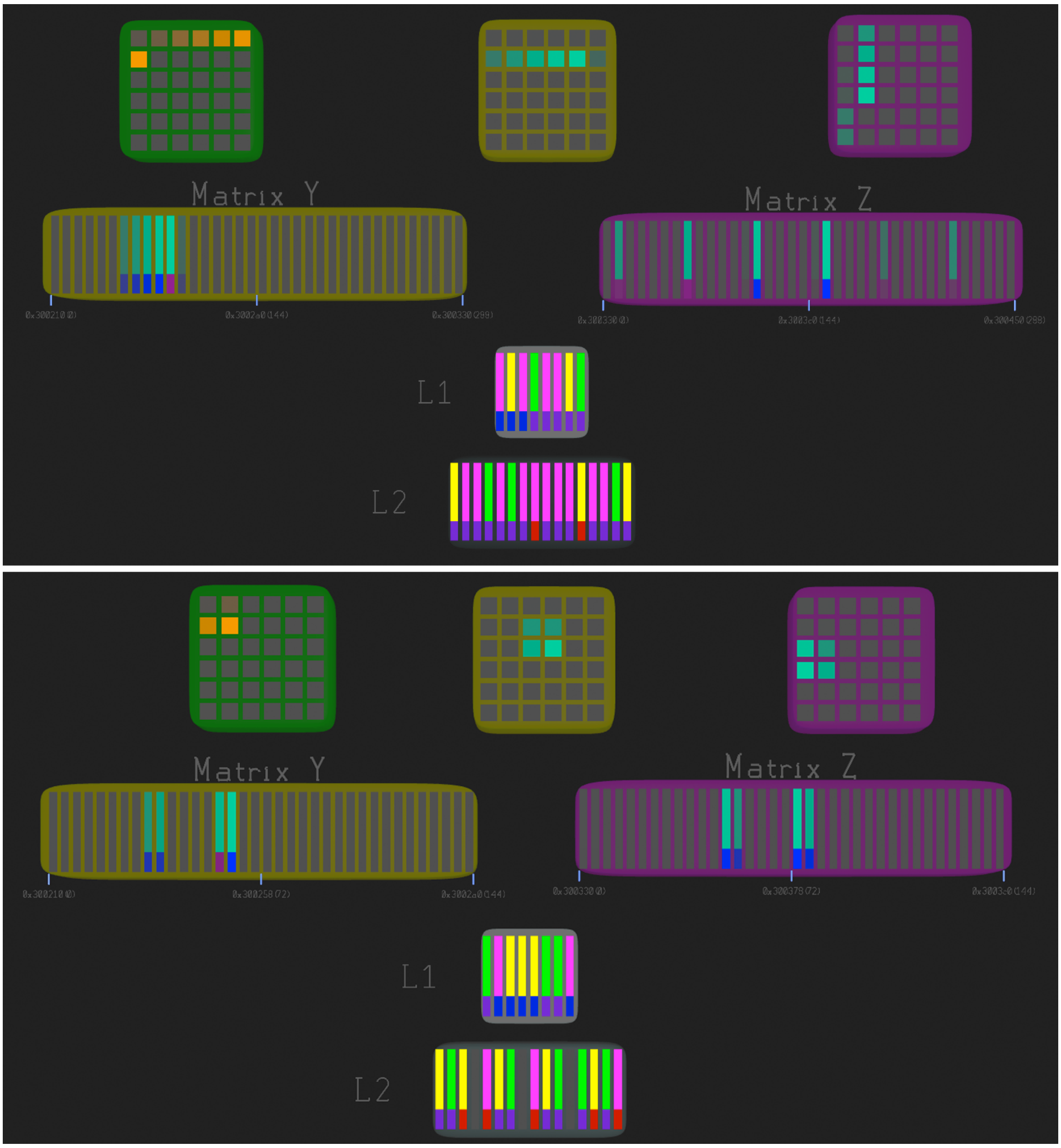


Figure 4.8. Naive matrix multiply (top) and blocked matrix multiply (bottom).

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.

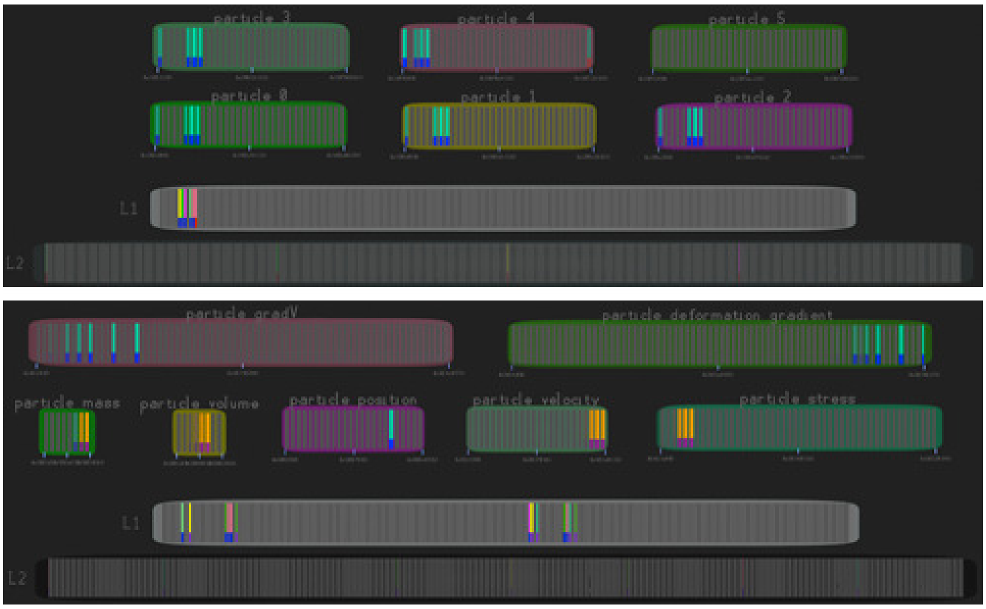


Figure 4.9. MPM Horizontal (top) and Vertical (bottom).

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 5.8, 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 4.4.1). 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 5.8, 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 5.8, 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.

4.5 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.

CHAPTER 5

ABSTRACT REFERENCE TRACE VISUALIZATION

This is an awesome chapter on abstract reference trace visualization.

5.1 Introduction

The interactions between modern hardware and software systems are increasingly complex which can result in unexpected interactions and behaviors that seriously affect software performance costing time and money. To address this issue, students and software engineers often spend a significant amount of their time understanding performance and optimizing their software.

One common performance analysis technique is to track cache activity within an application. This information is usually provided for very coarse time granularity. At best, cache performance is provided for blocks of code or individual functions. At worst, these results are captured for an entire application's execution. This provides only a global view of performance and limits the ability to intuitively understand performance. An alternative to this coarse granularity is to generate a memory reference trace, which can then be run through a cache simulator to produce a fine-grained approximation of the software's actual cache performance.

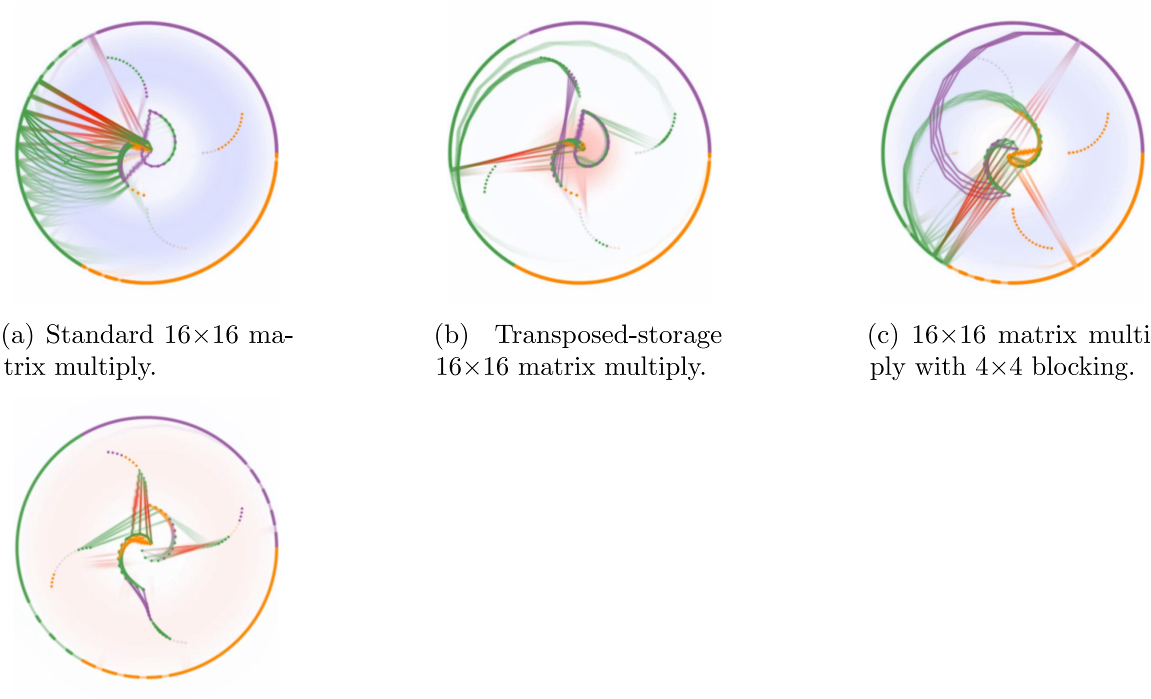
The biggest challenge when using this approach is sifting through the volume of data produced. Even simple applications can produce millions of references, yet this data contains valuable information that needs to be extracted to better understand program performance. The use of statistical methods or averaging simply produces a coarse understanding of software performance, forgoing the detail available in the trace. Static analysis of memory behavior is also possible [6], but limited only to cases where the program behavior can be deduced at compile time.

To address these problems, we propose visualizing the simulated cache and the reference trace, allowing developers to see their software with fine-grained detail, and bring their experience and intuition to bear on understanding software memory performance. We do this by introducing a system that provides an abstract visualization of the cache as the reference trace plays through it.

The goal of the system is to provide an intuitive understanding of how the computer hardware affects software performance, without the need to know or understand every feature of the hardware itself. The resulting visualizations correspond to our intuitive understanding of how caches work, yet are able to convey cache activity that may be difficult to envision or else are surprising in some way. Our approach is not a replacement for other conventional approaches, but rather an additional tool that can assist in software analysis.

Figure 5.1 shows four example images of our system visualizing different versions of the matrix multiply algorithm. Memory locations, represented by point glyphs, are placed on concentric rings based upon their cache residency. Lighter-colored, ghost glyphs are placed in the higher levels of cache (and the main memory region) to indicate duplication of data through the levels of the memory hierarchy. The outermost ring contains items in main memory, the middle ring contains items in the level 2 (L2) cache, and the innermost ring contains items in the level 1 (L1) cache. Our visualization provides an intuitive understanding about how memory is used and evicted from the cache. As locations are referenced, their glyphs move to the center of the visualization, and as they age (and are eventually evicted), they are pushed out towards the next concentric ring.

The remainder of the paper is organized as follows. In the next section



(a) Standard 16×16 matrix multiply.

(b) Transposed-storage 16×16 matrix multiply.

(c) 16×16 matrix multiply with 4×4 blocking.

(d) 12×12 matrix multiply with 4×4 blocking.

Figure 5.1. Matrix multiply in various incarnations. The standard algorithm shows good cache behavior for the left-hand matrix but poor behavior for the right-hand matrix. One solution is to operate on a transposed-storage version of the right-hand matrix, which results in better cache behavior, but a loss of generality in the allowed matrix operations. A common solution between the two is matrix blocking, in which submatrices are operated on to accumulate the final result piece by piece. By operating on small submatrices that fit into the cache, we can improve the cache performance of the multiply while keeping the generality of the standard matrix multiplication algorithm.

we discuss related work. Section 5.3 overviews our system while section 5.4 discusses the design decisions we have made in our abstract visualization approach. Section 5.5 discusses results and examines a few case studies. Section 5.6 concludes with future directions for this work.

5.2 Related Work

5.2.1 Memory Behavior Visualization

Software profilers, programs that observe the runtime behavior of a target application and generate statistics about where that application spent its time, are a basic tool for any study of software performance. Well-known examples include GNU GProf, VTune, and Shark. These programs report the amount of time spent in various functions or lines of code, allowing developers to direct their optimization effort. They are capable of providing, for example, aggregate cache miss statistics from hardware performance counters, but generally they do not provide information about how memory was used during the application's execution. Performance counters can also be accessed from applications by making use of specialized libraries [25]. The visualization provided by profilers is usually limited to graphs of the data that can show where the application spent more time, but not necessarily *why*.

Software profilers generalize to a certain class of visualization tools, exemplified by Vampir [14] and Tau [22] which use runtime profiling information to produce post-mortem, statistically-guided visualizations. They use classical information visualization techniques to show trends in bulk data about, for example, communication patterns between nodes of a cluster, and allow for the developer to identify high-level performance bottlenecks. They are essentially the visual counterparts of traditional code profilers.

More specific visualizations can provide insight about execution and performance, at many levels of detail. At the system level, whole-system data is collected in an attempt to visualize the various parts of the machine as an execution is carried out. Stolte et al. [24] present a system that visualizes important processor internals, such as functional unit utilization and pipeline stalls, and allows for drilling down to show details about certain subsystems. At the application level, runtime data is visualized in the familiar context of source code. For example, Heapviz [1] tracks heap allocations and their pointer dependencies in the Java runtime, displaying the heap's graph structure, allowing developers to see how their data structures develop during the run, possibly finding errors such as misallocations, unbalanced hash tables, etc.

Several approaches deal with the memory subsystem specifically. The Cache Visualization Tool [27] shows cache block residency, visualizing cache line contention due to the layout and access patterns of several active data structures. KCacheGrind [29] is a visual frontend for CacheGrind that visualizes the calling context over time, correlating cache miss costs with lines of source code. Yu et al. [32] use cache simulation to produce a static view of cache behavior over time. Each pixel in an image corresponds to the cache effect (hit or miss) of a single reference; as a whole, the image serves as a time-indexed “map” of cache performance. YACO [18] is a cache optimization tool focusing on performance statistics. Cache misses are counted and plotted in different ways, highlighting performance bottlenecks in lines of code and data structures. In our own earlier work, the Memory Trace Visualizer (MTV) [7] visualizes a reference trace and performs cache simulation, showing access patterns as they occur, and cumulative cache performance. By contrast, Grimsrud et al. [9] use traditional information visualization techniques, developing precise definitions of access locality, and visualizing the resulting measures in surface plots.

These approaches all provide specific insights, but none of them gives an overarching view of the behavior of the memory system and cache, including the elements residing therein, whereas our goal in the current work is to set up a system in which such a global view of many elements of the memory subsystem is possible, leading to insights about large-scale patterns and behaviors.

5.2.2 Organic Visualization

Our current work is inspired by organic visualization [8], an approach that imbues the visual elements with behavioral rules that allow them to self-organize into meaningful visual structures, much as individual cells are able to work together to constitute a whole organism. Codeswarm [16] is an example of the technique as applied to software visualization, in which source code repository data directs visual elements representing files and developers to form groups according to tight relationships between them. For instance, frequent committers associate into circles with their working files. Motion, proximity, color, and size all work together to express the important relationships between the participants. Our current work is inspired by systems such as Codeswarm, as such organic visualization systems are able to handle many visual elements by allowing them to aggregate automatically into higher-level structures—such as levels of a cache and semantically delineated regions of memory—so that their sheer volume does not obscure the insights they try to transmit. Compared to this more organic visualization behavior, our earlier system MTV addresses the same problem of visualizing reference traces, but in a more regimented, litral way. Concrete access patterns are more visible in MTV, while our present work is better able to show cache dynamics and data motion. As with much of the work described here, our system is trace driven, and performs cache simulation to derive some performance statistics that can be associated to the trace. In the next section we detail just how our system works, both in terms of visual element design, and their prescribed behaviors.

5.3 System Overview

In this section we briefly outline the data flow in our visualization system.

Memory Reference Traces. The system relies on memory reference traces collected from running applications as its primary data source. The traces are simply lists of addresses accessed by the application as it runs, together with a code indicating the type of transaction (i.e. read or write). We collect these at runtime using Pin [12], a dynamic binary rewriting infrastructure that allows for arbitrary code to be attached to any instruction at runtime. Collecting a reference trace is relatively straightforward: each load or store instruction is directed to trap to a recording function which writes the read-write code and the effective address to disk. We are also able to use debugging symbols in the executable to record correlations of instructions to line numbers in source code. This allows the visualization to correlate memory activity to the familiar source code context for the visual analysis. In a final step, the log of memory activity is filtered to allow the visualization to only display activity from variables and algorithms of interest to the developer. In this way, we can avoid displaying the many activities application perform which are not important to understanding the application's behavior.

Cache Simulation. We drive our analysis and visualization with cache simulation, so that users may start to understand how their application performance is affected by its interaction with the cache and the memory subsystem. Though there are several cache simulators available for research use, we use a home-grown simulator that allows us to have more control over what kinds of data can be extracted as output. The simulator takes as input individual reference records from the trace and computes their effects on the working sets in each cache level, reporting what level of the cache was hit, and which data items were moved from level to level or were evicted entirely.

Visualization. The results of the cache simulation are fed, step by step, to our visualization system. The system has a structural layout reflecting the simulated memory architecture, over which glyphs representing pieces of data arrange themselves to reflect the ongoing dynamic updates to the cache state as encoded in the reference trace and the cache simulation.

The current work focuses largely on the last component, visualization. Data collection and cache simulation are crucial parts of this effort, however the difficulties and issues they bring are outside the scope of this work. In the next section we describe the visualization system in careful detail,

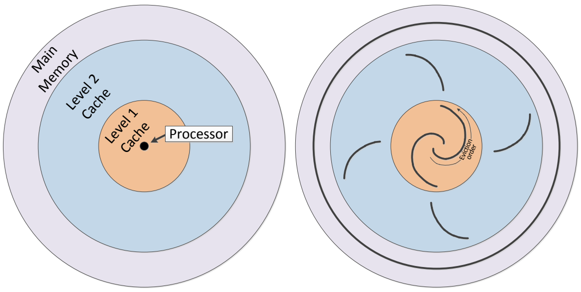


Figure 5.2. Left: Our visualizations are structured schematically as concentric rings representing the main memory and levels of cache. The central point represents the CPU. Increasingly distant from the center are the L1 and L2 caches, with main memory as the farthest ring. Right: Against this backdrop, point glyphs representing data items move from place to place to indicate residency in the various levels of the memory hierarchy. In the cache levels, the glyphs arrange themselves into groupings indicating the associative cache sets, with data on the verge of eviction appearing nearer the boundaries between the levels.

examining and describing our design choices, and how they add up to provide an insightful visual expression of the data in the reference trace.

5.4 Visualizing Reference Traces

In this section we describe the design of our system, focusing on the nature and usage of individual visual channels. In particular, we distinguish time scales in each channel by "frequency," reflecting the time scales over which changes in visual qualities tend to persist. Channels engage a low frequency when visual elements exhibit a longer-term, stable behavior, and a high frequency when they change rapidly. By way of example, we can consider the position of a data glyph—the low-frequency behavior is to settle into a position within a cache level or main memory; the high-frequency behavior is to move from one area to another in response to a cache level eviction event. Generally speaking, we use low-frequency qualities to establish baselines or express average behaviors over a long time period, reserving high-frequency qualities to reflect sudden changes in state, or very important events that need to draw the viewer's attention.

In broad strokes, the visualization system consists of a *structural layout* representing the levels of cache, and main memory, over which *data glyphs*, representing individual addressable pieces of memory, move according to behavioral rules. The positions of these glyphs encode their presence in one or more levels of cache.

5.4.1 Structured Cache Layout

The data glyphs occupy a structured visual space representing the machine architecture under consideration (Figure 5.2, left). Because locality is so important in understanding cache and memory behavior, the visual space encodes both spatial and temporal locality of memory using spatial layout design choices. The design is literally CPU-centric—the physical center of the display represents the computing core, encompassing the operation of functional units as well as the registers containing the working set of data. In radial layers about the center, we reserve space for the levels of cache, from fastest to slowest, while main memory is represented as a final radial layer beyond all the levels of cache. This structuring reflects the idea that as storage levels grow larger as they become slower and more "distant" from the computing core. Visually, it means that data glyphs representing pieces of memory must move from farther distances in order to occupy the CPU.

The glyphs further organize themselves to reflect the operation of particlar cache levels (Figure 5.2, right). For instance, in an L1 two-way cache, there are two sets into which data items may map—these are represented as interlocking spirals emanating from the center of the display. Similarly, the four sets of the L2 cache are represented as spiral arms emanating from the boundary of the L1 region. We choose to show the sets distinctly because this feature of caches is often abstracted away in the thinking of programmers, yet it may matter very much to cache performance. By rendering the distinction visible, we are able to demonstrate the resulting cache behavior and performance directly.

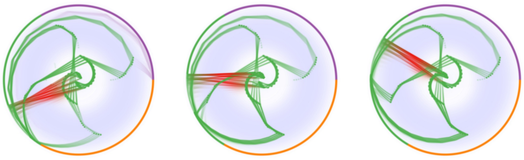


Figure 5.3. The common pattern of array initialization, as visualized in our system at three points in time. The red streak lines indicate cache misses for references to the green array. The data comes into L1 and is initialized with a series of write operations. As the next batch of data comes in, the initialized data becomes "stale" and moves slowly first out of L1 to L2, then out of L2 back into main memory. The bundle of red cache miss lines is seen to rotate through the array as the array items stream through, visually characterizing this pattern of access.

As mentioned above, the placement of the cache sets reflects their progressive "distance" from the CPU core; within each cache set representation, distance also encodes the eviction order, with glyphs that are about to be evicted from the cache positioned further away from the center, on the border with the slower cache level to which they will be sent.

A common cache design uses the "least recently used" (LRU) heuristic in deciding which cache block should be evicted when a new block arrives. Under an LRU block replacement strategy, distance from the center of the display can also be taken to encode *time*, so that glyphs that are more "stale" (i.e., have not been accessed for a long time) tend to appear further from the center. This placement rule renders certain access behaviors clearly visible. For instance, a common memory access pattern is that of *array initialization*, in which a newly created array must have its entries all set to some base value (Figure 5.3). Tracking a single data item d through the cache would reveal that at some point in time, it is brought into L1, where it is initialized. As subsequent data items are processed, d becomes older in L1, so it progressively moves further away along its spiral arm. When it reaches the end of the spiral, and yet another block is brought into L1, it is evicted to L2, where a similar process occurs, finally ejecting d back to its original home in main memory. Because time is, in this way,

Table 5.1. Visual channels engaged in our system

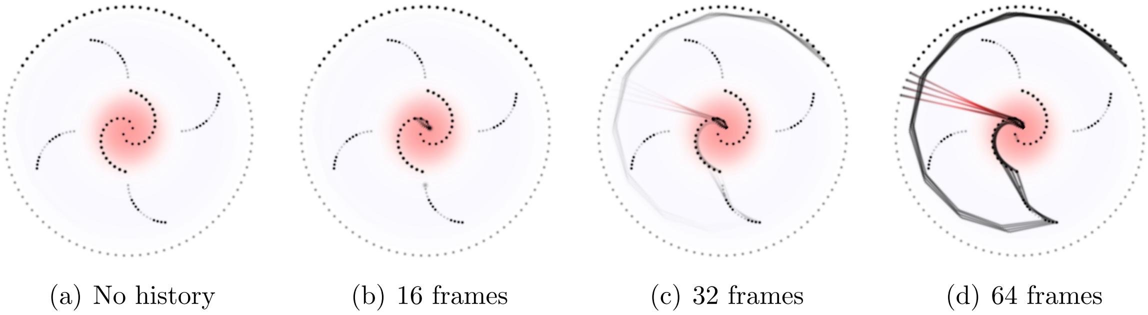
Visual Channel	High Frequency	Low Frequency
Structure	Eviction order	Cache level
Motion	Change in resident cache level	Changes in eviction order within cache level
Size	Access	—
Color	Cache miss	Home memory region

encoded as distance from the center, d moves along a radial path as it ages, eventually leaving the cache altogether. The visual pattern makes clear how the lack of reuse of d makes it both "older" and pushes it "far away" at the same time.

5.4.2 Data Glyph Behavior

Figure 5.2 demonstrates the static structuring we have designed as the space in which our visualization occurs. In fact, almost every dynamic aspect of this visualization occurs via the behavior of the data glyphs. In this section, we describe the visual channels occupied by the glyphs and how they make use of these channels at both low and high frequencies to transmit information about the reference trace.

Motion. One of the glyphs' basic jobs is to move from place to place to express their changing occupancy of different memory hierarchy levels in response to cache events. Because glyphs are alloted the same amount of time for each move, larger distances are covered at higher velocities than shorter ones. Important events such as cache misses and evictions appear as visually striking, higher velocity actions than do cache hits; when a flurry of such events occurs, the effect is a jumble of high-speed activity which appears very clearly and draws the viewer's attention (Figure 5.3 demonstrates this idea for a specific kind of memory access pattern).



(a) No history (b) 16 frames (c) 32 frames (d) 64 frames

Figure 5.4. Demonstration of the effect of history pathlines on the visualization. These four images are of the exact same simulation time, varying only in the amount of history accumulated into the fading trails. In (a) we see only the current animation frame, with no sense of history. In (b) we see the last 16 frames, which show that L1 hits have been taking place in the recent past. In (c) there is evidence of a recent cache miss, and an associated eviction event, while in (d) we see these same events in heavier detail. Note that while the same set of events is visible in (c) and (d), the longer history trail in (d) tends to obscure the L1 activity that is clearer in (b). By providing an interactive control for this feature, the user can select the amount of history that is appropriate for the current visual analysis task.

Within a particular cache level, slower motion to the head of the cache set indicates a cache hit. With many cache hits occurring in a row, the visual character is that of several glyphs vying for the head position in the cache. The volume of activity is again expressed by volume of motion, but the short distances involved serve as a visual reminder that the observed behavior exhibits good locality. This channel is naturally high-frequency, as glyphs cover long distances quickly only when they are evicted from one cache level and enter another—a momentary state change that occurs locally in time. The low-frequency component is simply lack of motion, expressing residency within the current level of cache. Furthermore, we distinguish between data entering the cache (in response to a cache miss), and data leaving the cache (due to eviction)—the former is expressed by fast, straight-line motion, while in the latter, glyphs move in a wider circular motion to suggest fleeing.

As noted before, position also plays an important role in expressing cache performance. The cache levels are arranged so that their distance from the center reflects their architectural distance from the CPU; the distance away from the center in each cache level further reflects how old each access is, as measured from the last time it was accessed. Therefore, data items with poor utilization slowly migrate to the outer edge of their home cache levels, and are evicted by incoming data items at the appropriate time to a farther cache level. By watching this slowly developing positional change, one may learn about the effect of under-utilization of these data items.

Color. Each glyph's color reflects the region of memory it comes from. For example, Figure 5.8 shows several arrays of data from a particle simulation, each containing a certain type of simulation value (mass, velocity, etc.). Using the region identity as the base color for the glyphs allows for understanding the composition of the current working set at a glance. In Figure 5.1(a), L1 is seen to contain elements from the two multiplicand matrices in a particular order.

The region identity occupies the low-frequency component of the color channel; it may also be used to indicate important events at a high-frequency as they occur. For instance, when glyphs move from slower cache levels to faster ones (i.e., "closer" to the CPU), this indicates a cache miss event, which are important to understand in achieving high software performance. Therefore, as the glyphs move to the L1 cache in response to such an event, they flash red momentarily to indicate their involvement in the cache miss event.

Size. Along with color, the size of the glyphs makes up their basic visual composition. The data glyphs all have an equal baseline size (i.e., the low-frequency size channel empty) in order to emphasize the relative composition of the cache levels without singling out any particular data items.

The high-frequency size channel is used to redundantly encode an access to a particular data item. When a data item is accessed, it pulses larger momentarily, with the effect of highlighting it among all the data items present in the cache level along with it. When the data item is not already present in the L1 level, its pulsation can be seen as it moves into L1 in response to the cache miss event, once again highlighting the important event (in this case, the pulsation redundantly strengthens the red glow as discussed above).

5.4.3 Time-Lapse Mode

Memory reference traces can be very large; as such, visualizations produced from them can be intractably long to observe. One option would be to simply speed up the visualization by increasing the speed of trace playback and glyph motion. This approach works until the speed becomes so high that glyph motion is no longer visible.

To address this limitation, we have taken the approach of compressing several timesteps into a single animation frame, encoding the changes in glyph positions through time by using pathlines. First, a fast forward speed is set (e.g., 2×, 4×, etc.), indicating the number of animation frames to skip in visualization. The positions of glyphs are calculated for those skipped frames, and a pathline is used to connect the glyph positions at those intermediate times. When the time-compressed frames are played at a normal speed, simulation time appears to have sped up dramatically, yet the pathlines keep the sense of evolving time coherent.

The pathlines can be controllably extended further into history as desired. Figure 5.4 shows four different settings for the tail length for the same time step. Increasing the tail length shows more events, but also tends obscure individual events—the tradeoff can be managed by the user interactively. Transparency in the pathlines indicates age, older events appearing more transparent, while newer events appear opaque. The time-lapse view therefore shows higher-order temporal patterns in addition to managing the commonly long time scales present in most reference traces.

5.4.4 Summary Views

The structured layout also provides for displaying a general quantity computed from the trace as a whole, allowing, for example, statistical information about the trace to be included in the display. The computed value is displayed in a soft, colormapped disk behind the areas reserved for the cache levels. In our examples, we have computed the "cache temperature," a measure of the proportion of transactions in each cache level resulting in a hit. More precisely, each reference trace record causes a change in the cache: each level may either *hit*, *miss*, or else be *uninvolved* in the transaction. These are assigned scores (a negative value for a miss, a positive value for a hit, and zero for noninvolvement) which are averaged over the last N reference trace records. The assigned scores may vary for different levels; for example, the penalty for a miss is higher for the L1 cache, because once a cache line is loaded into L1, it will have more of a chance to make heavy reuse of the data than a slower level would. In each level, the cache temperature rises above zero when the volume of data reuse exceeds the "break even" point, and falls below zero when there is not enough reuse. When a cache level sits idle (because, for instance, faster levels are hitting at a high rate), its temperature gradually drifts back to zero. The metaphor is that new data are cold, causing a drop in temperature, but accessing resident data releases energy and raises the temperature. Between these extremes, sitting idle allows for the temperature to return slowly to a neutral point.

The cache temperature is displayed as a glowing color behind the appropriate structural elements of the display. We have used a divergent colormap consisting of colors that naturally express relative temperatures: it runs from white in the middle (the neutral color indicating no activity, or a balance of hits and misses) to red at the warm end (indicating a relatively high volume of cache hits), and to blue at the cool end (for a relatively high volume of misses).

The cache temperature glyphs provide a context for the patterns of activity that occur over it. When the cache is warm, the pattern of activity will generally show frequent data reuse, while there may be many patterns to explain a cold cache. The changing temperature colors help to highlight periods of activity leading to both kinds of cache behavior.

5.5 Results and Discussion

In this section we review several case studies, identifying performance and behavioral characteristics that can be seen with our visualization methods.

5.5.1 Matrix Multiply

Matrix multiplication is ubiquitous in many computing fields and as such its caching performance has been of interest to programmers. Here we examine the cache behavior of matrix multiply using our visualization approach.

Standard Algorithm. The standard matrix multiplication algorithm computes dot products of the rows of the left matrix with the columns of the right matrix. This algorithm achieves good cache characteristics for only one of the matrices, since the other must have its elements accessed in an order that does not correspond to its layout in memory. Visually, it can be seen that the cache contains contiguous blocks from one array, and separated blocks from the other; the separated blocks each have a single element that is accessed during each dot product, and these blocks flow in and out of L1 for each column (Figure 5.5).

Figure 5.1(a) demonstrates that the cache misses incurred by the right matrix (in green) are almost constant, whereas the left matrix (purple) is able to achieve much more data reuse. The lack of reuse in the right

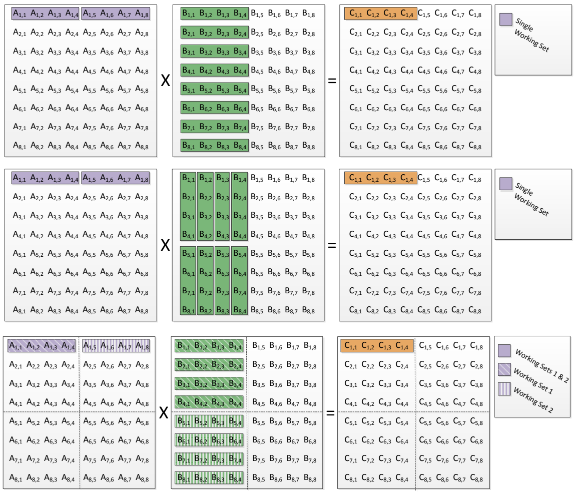


Figure 5.5. A schematic view of the cache properties of matrix multiply. Top: The standard algorithm computes dot products of rows of the left hand matrix with columns of the right hand matrix. This requires pulling the indicated cache lines into the cache. Unfortunately, as the columns of the right hand matrix are accessed, the upper lines will tend to be evicted, causing them to be pulled in again for each column, leading to poor cache performance. Middle: One simple idea for optimizing the multiplication is to compute with the transpose of the right-hand matrix, accessing its rows rather than its columns during the computation. The access patterns for both matrices become spatially coherent, but at the cost of restricting where the transposed matrices may be used. Bottom: By blocking the matrix multiply, we can bring in fewer numbers of cache lines at a time, operating on the full set of data present before bringing in a new block on which to operate. The results are eventually accumulated in the output matrix, and the correct product is computed with better cache behavior than the standard algorithm. Blocking retains some of the locality of the transposed approach, while also keeping the generality of the standard matrix multiply.

matrix is conveyed visually by new data streaming into L1 as older data is ejected from the cache in an almost pipelined manner. The misses come from the ejected data having to re-enter the cache every time a column is traversed.

Transposed Matrix Multiply. The visualization leads to a simple idea: if we stored the *transpose* of the right matrix, then we would improve its caching behavior by accessing its rows instead of its columns. Figure 5.1(b) shows that the number of cache misses is largely reduced. The left matrix (purple) is still seen to have better cache residency and reuse; this is due to the fact that the dot products of a single row from that matrix are computed against all columns of the right matrix, so it tends to reside in the cache for longer.

Blocked Matrix Multiply. Storing transposed matrices restricts the allowed operations performed on them—transposed matrices can only participate as the "right matrix" in any multiplication. A common cache optimization for the standard algorithm is instead to use *blocking*, in which submatrices are repeatedly multiplied and accumulated in the final output. Rather than a single row of one matrix and a single value of one column residing together in the cache at a time, blocking allows for the submatrices to occupy the cache instead, occupying a middle ground between the standard and transposed algorithms, while retaining the generality of standard matrix multiply.

Figures 5.1(c,d) show that the overall volume of cache misses is reduced, and more evenly distributed between the matrices. As the submatrix lines are brought into cache, they remain there relatively longer and get better data reuse than in the naive case.

5.5.2 Sorting Algorithms

Sorting algorithms are a natural choice for demonstrating reference trace visualization, as the algorithms are usually straightforward and simple to implement and understand, and therefore have simple yet important interactions with the cache. In this section we compare two well-known sorting algorithms, uncovering their cache performance characteristics: bubble sort and merge sort. Bubble sort is known for its slow $O(n^2)$ average-case running time, but it has good cache performance characteristics. By contrast, merge sort has a better running time, and we demonstrate its particular cache behavior characteristics.

Bubble Sort. Bubble sort is a well-known sorting algorithm with a very simple implementation, in which repeated sweeps of the array to be sorted cause large items to be swapped to the end. After the ith sweep, the

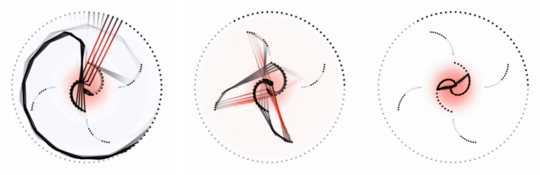


Figure 5.6. Bubble sort, a sorting algorithm in which progressive sweeps swap the remaining largest element to the correct location. Because the sweeps become progressively shorter, the size of the working set continuously decreases until it fits first within L2, and then within L1, leading to good cache behavior at the end of the algorithm.

ith largest element is sorted into place; therefore, the algorithm requires N sweeps of steadily decreasing length in the worst case to sort the entire list. The visualization of the memory behavior of this algorithm (Figure 5.6) shows an interesting characteristic—as the algorithm nears completion, and the size of the remaining elements to sort begins to fit in the cache, cache performance steadily improves. During the first sweep, all elements of the array are accessed in turn, and the visualization shows every block of values entering and then exiting the cache. The L1 cache temperature rises due to the high volume of swaps occuring there, while the L2 cools due to the lack of available data reuse in that level (since each item is accessed at one time during each sweep). However, because fewer and fewer elements are needed in each subsequent sweeps, eventually all of the required data populates the L2—and then L1—cache, and no further evictions take place. This is illustrated by the sustained flurry of activity between L1 and L2, and then later solely in L1, indicated by frequent, localized streak lines and an increase in the observed cache temperatures. The visualization clearly shows the increasing spatial locality inherent in the access patterns associated to bubble sort.

Although bubble sort is famously slow in algorithmic complexity, it does in fact have—at least during certain segments of its execution—desirable cache behavior. Our conclusion from this initial example: though reasoning carefully about bubble sort would lead to the insights about its execution presented here, our visualization makes the insights immediately graspable—its value lies in its ability to quickly, decisively, and *visually* convey those insights, which can then later be confirmed by reasoning about the program.

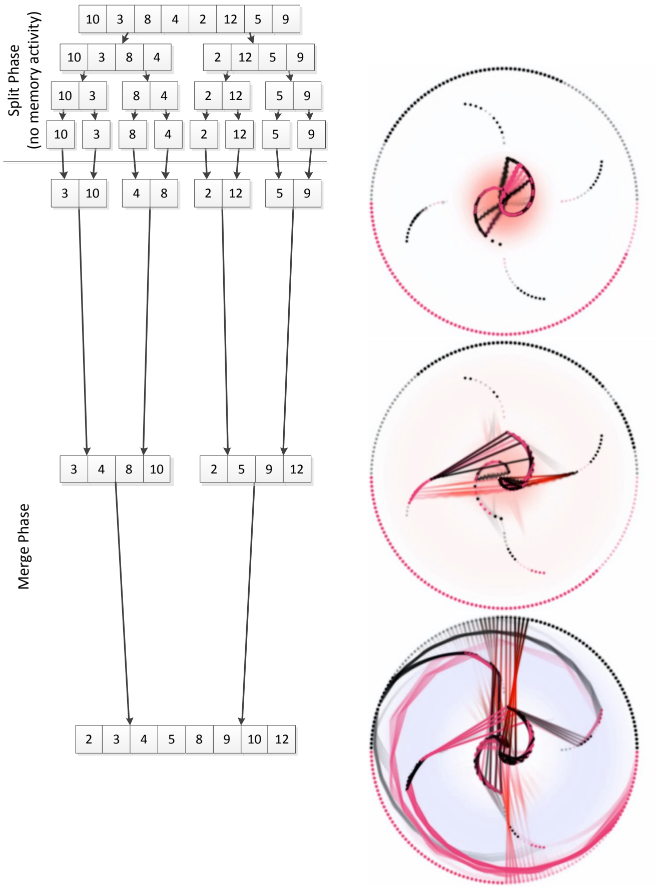


Figure 5.7. Left: A schematic view of how merge sort works. In the top half, the sorting function is recursively called on each half of the input. This step simply sets up a tree of computation that will accomplish the sorting, without any memory access. In the lower half, atomic lists of a single element are combined anti-recursively by merging, resulting in progressively larger, sorted sublists. This stage involves comparisons and movement of elements to a temporary working store, before they are copied back to the input array. Each depicted merging phase matches with a snapshot of our visualization on the right. Right: Visualization of the memory behavior of the merge phase. This has roughly the opposite cache behavior as bubble sort—it begins its memory transactions with small lists that fit entirely in the cache, forming progressively larger lists that eventually overspill the cache levels, leading to poorer cache characteristics near the end of the algorithm.

Merge Sort. Merge sort typifies the "efficient" sorting algorithms—it achieves the $O(n \log n)$ lower complexity bound on comparison-based sorting algorithms. It is a divide-and-conquer algorithm that works by dividing the list into two parts, applying the merge sort procedure recursively to each half, and then reassembling a sorted list by sweeping each list, transferring the appropriate value to the result array.

Though the algorithm has good asymptotic complexity, it may be somewhat surprising to see that its cache behavior is somewhat erratic. In the initial phase of the algorithm, the input is recursively subdivided into a tree of lists of single elements (each of which is trivially already sorted, by definition). In this phase, no memory transactions are performed on the elements, so its cache performance is vacuously neutral. The second half of the merge sort algorithm builds the sorted output by anti-recursively merging the single-element lists, then the two-element lists that result, etc. This phase starts out with good cache performance, as the lists to be sorted are small and fit entirely into L1 (Figure 5.7 top), but as sorted elements begin to move farther and farther distances (as they jump from their current position to the head of a progressively sorted subarray), spatial locality degrades. This can be seen in the spilling over of the working set into L2 (Figure 5.7 middle), and then into main memory, with increasingly frequent bursts of cache misses as the merge phase progresses (Figure 5.7 bottom). At the midway point, the process begins again for the second half of single-element lists, and the cache behavior recurs once more.

5.5.3 Material Point Method

The material point method (MPM) [4] is a particle-based mechanical engineering simulation method in which objects are discretized into collections of points, which undergo loads according to certain rules. Here we demonstrate a running MPM code and highlight some of its cache behaviors. We present it here as an example of a real-world code running

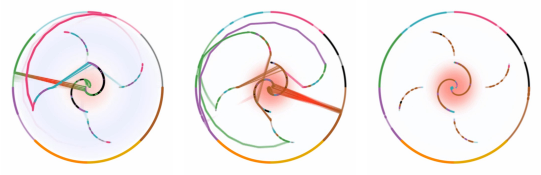


Figure 5.8. The material point method (MPM), a particle-based mechanical engineering simulation, in action. Left: Computation of momentum from the mass and velocity data (in the black and green arrays). The algorithm tends to sweep through the values in order, resulting in good cache performance. Middle: Computation of the particle stress update (brown data array) near the end of the timestep, from various data, including the constitutive model (blue data array). MPM is made up of several phases which tend to access the data in order. The resulting visual pattern is that of data moving into L1, being operated upon a limited number of times, and then slowly migrating first to L2 and then back into main memory, as newer data comes into L1 to be operated upon in turn. Right: This example shows a bigger MPM simulation and a larger cache to demonstrate the scalability of our visualization system.

in our visualization system.

Figure 5.8 shows an MPM timestep at various points. Figure 5.8 left shows an early phase of the timestep, in which the particle momenta (computed from their masses and velocities—the black and purple data arrays, respectively) are interpolated to a background mesh via their positions (the green data array).

In Figure 5.8 middle, we see the particle stress update (the brown data array) taking place, with input from the physical constitutive model (blue data array), using a sweeping access pattern that will engage each particle in the system. As this action continues, the data seen to reside in L2, which is no longer needed during this phase of the timestep, will slowly age and be pushed out by the newer incoming data—the hallmark of a "streaming" style of access, which is embodied by the stress update.

This example contains more data than our previous examples, and we have also quadrupled the size of the simulated cache. As Figure 5.8 right shows, our system is able to scale up to larger sizes. Currently, our bottleneck lies on the data collection side, rather than the visualization side.

5.6 Conclusions and Future Work

We have presented a visualization system for memory reference traces, drawing inspiration from organic visualization approaches, in reaching for the goal of illustrating the large-scale behavior of memory access and caching during the run of a program. Our system includes cache simulation as a way to drive performance analysis, and uses a carefully orchestrated set of visual qualities to convey important information about a program's runtime memory behavior.

We have several ideas in mind for future work. Although we have argued that our design decisions work well to convey information, there is still possible exploration of the visual channels we have discussed. For instance, the low-frequency motion chanel is largely unused in our current approach—mainly because we believe the visualization is more effective this way—but it may be the case that other effects in various visual channels are in fact useful. We would like to prototype several such effects, design a user study, and investigate whether uninitiated subjects find them useful.

There is also no reason to restrict these techniques to just the memory subsystem. A crucial part of the current effort rested in designing a meaningful static structure against which to overlay the dynamically changing data glyphs. We believe that such designs are possible for many different kinds of system architectures, and that with the right kinds of data sources, we could adapt this approach to diverse computing platforms. The generally accepted difficulty of high-performance software enterprises invites approaches such as ours to help developers understand the performance characteristics of their programs.

CHAPTER 6

TOPOLOGICAL ANALYSIS AND VISUALIZATION OF MEMORY REFERENCE TRACES

Lorem ipsum blah blah blah.

CHAPTER 7

ENSEMBLE UNCERTAINTY IN MEMORY REFERENCE TRACES

Lorem ipsum blah blah blah.

CHAPTER 8

RESULTS AND DISCUSSION

Lorem ipsum blah blah blah.

CHAPTER 9

CONCLUSION

Lorem ipsum blah blah blah.

REFERENCES

[1] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer, *Heapviz: interactive heap visualization for program understanding and debugging*, in Proceedings of the 5th international symposium on Software visualization, 2010, pp. 53–62.

[2] Apple Corporation, *Performance and debugging tools overview*. http://developer.apple.com/tools/performance/overview.html.

[3] S. G. Bardenhagen and E. M. Kober, *The generalized interpolation material point method*, CMES, 5 (2004), pp. 477–495.

[4] ——, *The generalized interpolation material point method*, Computer Modeling in Engineering and Sciences, 5 (2004), pp. 477–496.

[5] K. Beyls, E. H. D'Hollander, and Y. Yu, *Visualization enables the programmer to reduce cache misses*, in IASTED Conference on Parallel and Distributed Computing and Systems, Nov 2002, pp. 781–786.

[6] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck, *Exact analysis of the cache behavior of nested loops*, SIGPLAN Not., 36 (2001), pp. 286–297.

[7] A.N.M. I. Choudhury, K. C. Potter, and S. G. Parker, *Interactive visualization for memory reference traces*, Computer Graphics Forum, 27 (2008), pp. 815–822.

[8] B. J. Fry, *Organic information design*, Master's thesis, Massachusetts Institute of Technology, May 2000.

[9] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, *Locality as a visualization tool*, IEEE Transactions on Computers, 45 (1996), pp. 1319–1326.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, third ed., 2003.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, in Programming Language Design and Implementation, Chicago, IL, June 2005, pp. 190–200.

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

[13] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of mpi resources*, Supercomputer, 12 (1996), pp. 69–80.

[14] ——, *VAMPIR: Visualization and analysis of MPI resources*, Supercomputer, 12 (1996), pp. 69–80.

[15] N. Nethercote and J. Seward, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, in Programming Language Design and Implementation, June 2007.

[16] M. Ogawa and K.-L. Ma, *code_swarm: A design study in organic software visualization*, IEEE Transactions on Visualization and Computer Graphics, 15 (2009), pp. 1097–1104.

[17] B. Quaing, J. Tao, and W. Karl, *YACO: A user conducted visualization tool for supporting cache optimization*, in Proceedings of HPCC, 2005, pp. 694–603.

[18] ——, *Yaco: A user conducted visualization tool for supporting cache optimization*, in Proceedings of HPCC, 2005, pp. 694–703.

[19] S. P. Reiss, *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.

[20] S. P. Reiss and M. Renieris, *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.

[21] S. S. Shende and A. D. Malony, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, 20 (2006), pp. 287–331.

[22] S. S. Shende and A. D. Malony, *The tau parallel performance system*, Int. J. High Perform. Comput. Appl., 20 (2006), pp. 287–311.

[23] C. Stolte, R. Bosch, P. Hanrahan, and M. Rosenblum, *Visualizing application behavior on superscalar processors*, in INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization, 1999, pp. 10–17.

[24] C. Stolte, R. Bosch, P. Hanrahan, and M. Rosenblum, *Visualizing application behavior on superscalar processors*, in Information Visualization, 1999, pp. 10–17.

[25] D. Terpstra, H. Jagode, H. You., and J. Dongarra, *Collecting performance data with PAPI-C*, in Tools for High Performance Computing, 2009, pp. 157–173.

[26] R. A. Uhlig and T. N. Mudge, *Trace-driven memory simulation: A survey*, ACM Computing Surveys, 29 (1997), pp. 128–170.

[27] E. van der Deijl, G. Kanbier, O. Temam, and E. Granston, *A cache visualization tool*, Computer, 30 (1997), pp. 71–78.

[28] E. van der Deijl, G. Kanbier, O. Temam, and E. D. Granston, *A cache visualization tool*, Computer, 30 (1997), pp. 71–78.

[29] J. Weidendorfer, *Sequential performance analysis with callgrind and kcachegrind*, in Tools for High Performance Computing, 2008, pp. 93–113.

[30] J. Weidendorfer, M. Kowarschik, and C. Trinitis, *A tool suite for simulation based analysis of memory access behavior*, ICCS, 3038 of LNCS (2004), pp. 440–447.

[31] W. A. Wulf and S. A. McKee, *Hitting the memory wall: Implications of the obvious*, Computer Architecture News, 23 (1995), pp. 20–24.

[32] Y. Yu, K. Beyls, and E. D'Hollander, *Visualizing the impact of the cache on program execution*, in Information Visualization, 2001, pp. 336–341.

[33] Y. Yu, K. Beyls, and E. H. D. Hollander, *Visualizing the impact of the cache on program execution*, in Proceedings of the Fifth International Conference on Information Visualisation, July 2001, pp. 336–341.