Abstract Visualization of Runtime Memory Behavior

A.N.M. Imroz Choudhury
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112
email: roni@cs.utah.edu

Paul Rosen
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112
email: prosen@sci.utah.edu

***Abstract*—We present a system for visualizing memory reference traces, the records of the memory transactions performed by a program at runtime. The visualization consists of a structured layout representing the levels of a cache and a set of data glyphs representing the pieces of data in memory being operated on during application runtime. The data glyphs move in response to events generated by a cache simulator, indicating their changing residency in the various levels of the memory hierarchy. Within the levels, the glyphs arrange themselves into higher-order shapes representing the structure of the cache levels, including the composition of their associative cache sets and eviction ordering. We make careful use of different visual channels, including structure, motion, color, and size, to convey salient events as they occur. Our abstract visualization provides a high-level, global view of memory behavior, while giving insight about important events that may help students or software engineers to better understand their software's performance and behavior.**

I. 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 [**?**], 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 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 we discuss related work. Section III overviews our system while section IV discusses the design decisions we have made in our abstract visualization approach. Section V discusses results and examines a few case studies. Section VI concludes with future directions for this work.

II. Related Work

A. 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

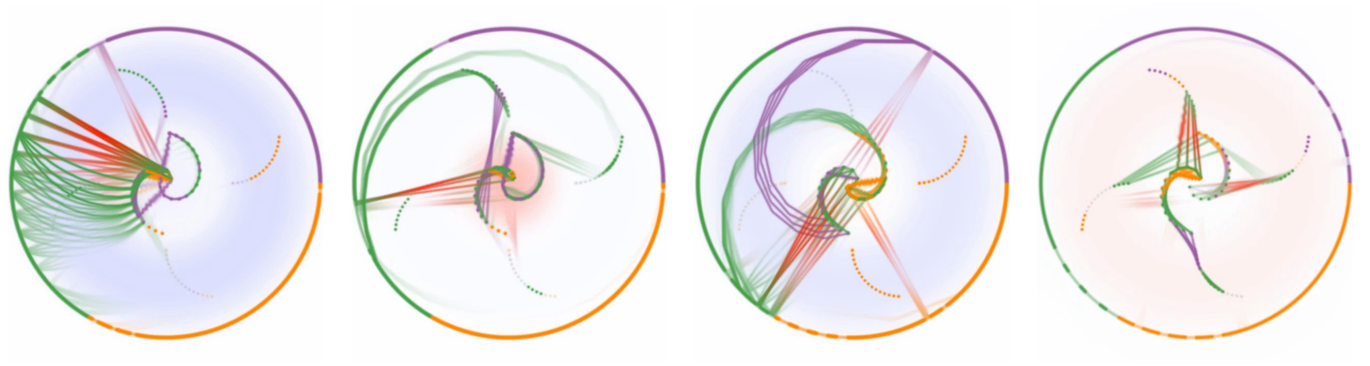
(a) Standard 16×16 matrix multiply. (b) Transposed-storage 16×16 matrix multiply. (c) 16×16 matrix multiply with 4×4 blocking. (d) 12×12 matrix multiply with 4×4 blocking.

Fig. 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.

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 [1]. 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 [2] and Tau [3] 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. [4] 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 [5] 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 [6] shows cache block residency, visualizing cache line contention due to the layout and access patterns of several active data structures. KCacheGrind [7] 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. [8] 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 [9] 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) [10] visualizes a reference trace and performs cache simulation, showing access patterns as they occur, and cumulative cache performance. By contrast, Grimsrud et al. [11] 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.

B. Organic Visualization

Our current work is inspired by organic visualization [12], 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 [13] 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.

III. 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 [14], 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, examining and describing our design choices, and how they add up to provide an insightful visual expression of the data in the reference trace.

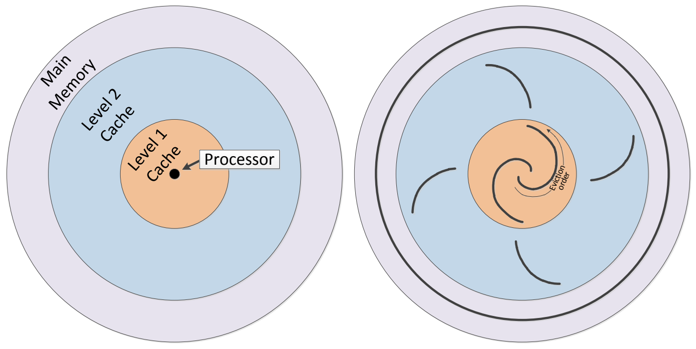

Fig. 2. Left: Our visualizations are structured schematically as concentric rings representing the main memory and levels of cache. The central point represents the CPU with its registers. Increasingly distance from the center are the L1 and L2 caches, with main memory as the farthest ring. Right: Against 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.

IV. 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.

A. Structured Cache Layout

The data glyphs occupy a structured visual space representing the machine architecture under consideration (Figure 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

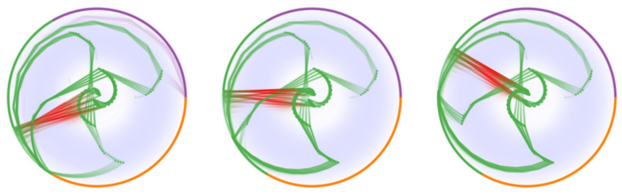

Fig. 3. The common pattern of array initialization, as visualized in our system at three points in time. The red streak lines indicate cache misses for references to the green array. The data comes into L1 and is initialized with a series of write operations. As the next batch of data comes in, the initialized 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 data stream through, visually characterizing this pattern of access.

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 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.

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 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 set to zero. As subsequent data items are processed, d becomes older in the 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, 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.

TABLE I
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

B. Data Glyph Behavior

Figure 2 demonstrates the static structuring we have designed as the space in which our visualization occur. 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 3 demonstrates this idea for a specific kind of memory access pattern).

Within a particular cache level, slower motions 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 the 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.

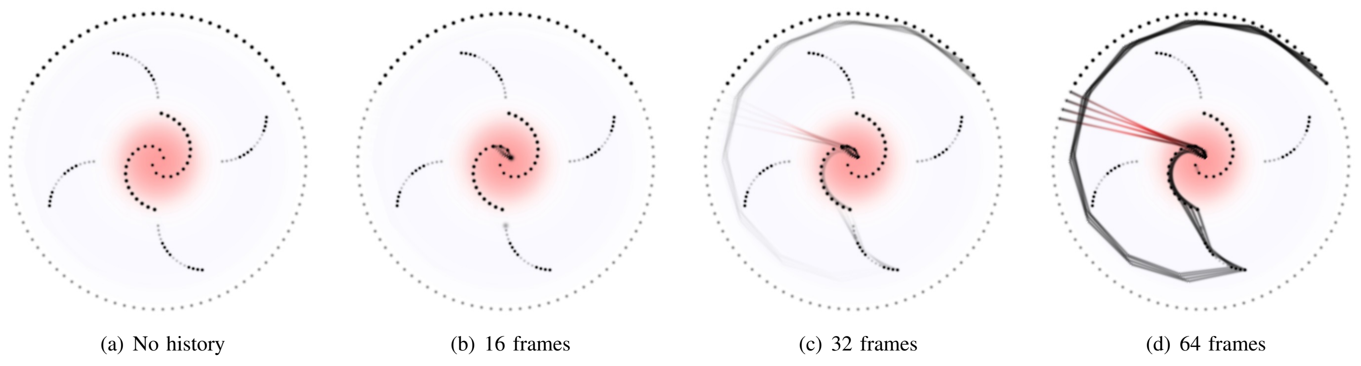

Fig. 4. Demonstration of the effect of history pathlines on the visualization. These four images are of the exact same simulation time, varying only in the amount of history accumulated into the fading trails. In (a) we see only the current animation frame, with no sense of history. In (b) we see the 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.

Color. Each glyph's color reflects the region of memory it comes from. For example, Figure 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 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).

C. Time-Lapse Mode

Memory reference traces can be very large; as such, the visualization produced from it 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., $2\times$, $4\times$, 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 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.

D. 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.

V. Results and Discussion

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

A. *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).

Figure 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 of its data. The lack of reuse in the right 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 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.

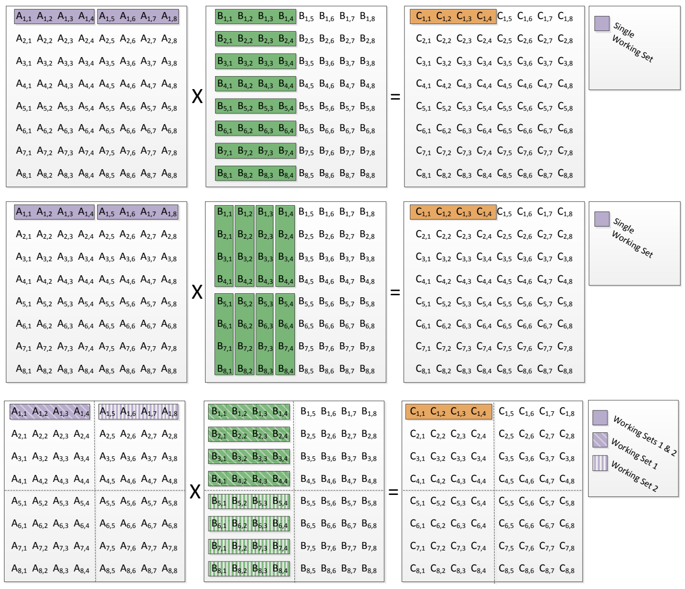

Fig. 5. A schematic view of the cache properties of matrix multiply. Top: The standard algorithm computes dot products of rows of the left hand matrix with columns of the right hand matrix. This requires pulling the indicated cache lines into the cache. Unfortunately, as the columns of the right hand 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, 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.

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 1(c,d) show that the overall volume of cache misses is reduced, and is now 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.

B. *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 $O(n^2)$

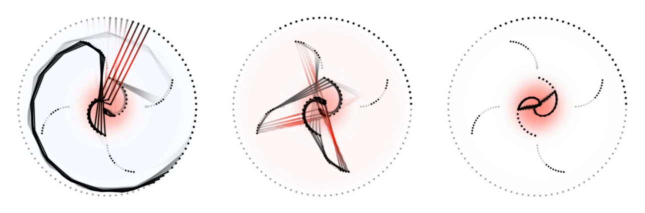

Fig. 6. Bubble sort, a sorting algorithm in which progressive sweeps swap the remaining largest element to the correct location. Because the sweeps become progressively shorter, the size of the working set continuously decreases until it fits first within L2, and then within L1, leading to good cache behavior at the end of the algorithm.

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 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 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.

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 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 7 middle), and then into main memory, with increasingly frequent bursts of cache misses as the merge phase progresses (Figure 7 bottom). At the midway point, the process begins again for the second

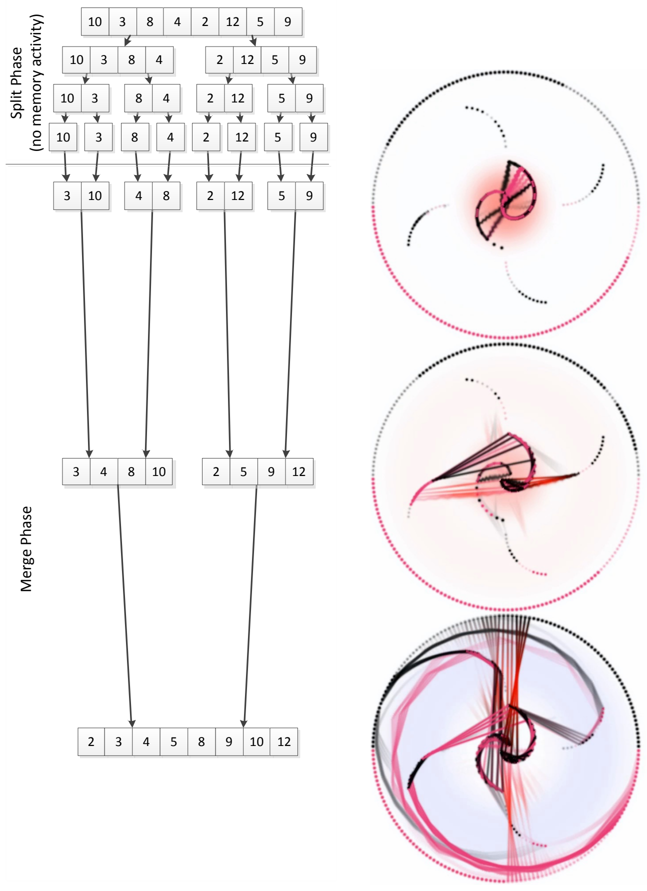

Fig. 7. Left: A schematic view of how merge sort works. In the top half, the sorting function is recursively called on each half of the input. This step simply sets up a tree of computation that will accomplish the sorting, without any memory access. In the lower half, atomic lists of a single element 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.

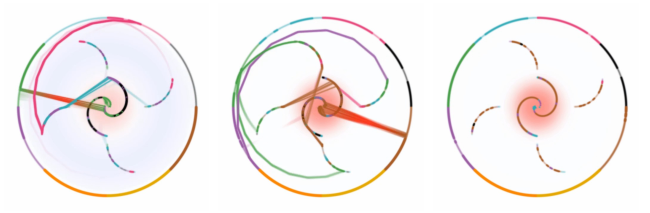

Fig. 8. The material point method (MPM), a particle-based mechanical engineering simulation, in action. Left: Computation of momentum from the mass and velocity data (in the black and green arrays). The algorithm tends to sweep through the values in order, resulting in good cache performance. Middle: 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 uses a larger example with a larger cache to demonstrate the scalability of our visualization system.

half of single-element lists, and the cache behavior recurs once more.

C. Material Point Method

The material point method (MPM) [15] 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 in our visualization system.

Figure 8 shows an MPM timestep at various points. Figure 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 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 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.

VI. 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 explorations 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 how useful uninitiated subjects find them.

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.

References

[1] 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.

[2] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, January 1996.

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

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

[5] 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.

[6] E. van der Deijl, G. Kanbier, O. Temam, and E. Granston, "A cache visualization tool," *Computer*, vol. 30, no. 7, pp. 71–78, July 1997.

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

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

[9] B. Quaing, J. Tao, and W. Karl, "Yaco: A user conducted visualization tool for supporting cache optimization," in *Proceedings of HPCC*, 2005, pp. 694–703.

[10] A.N.M. I. Choudhury, K. C. Potter, and S. G. Parker, "Interactive visualization for memory reference traces," *Computer Graphics Forum*, vol. 27, no. 3, pp. 815–822, May 2008.

[11] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, "Locality as a visualization tool," *IEEE Transactions on Computers*, vol. 45, no. 11, pp. 1319–1326, November 1996.

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

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

[14] 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.

[15] S. G. Bardenhagen and E. M. Kober, "The generalized interpolation material point method," *Computer Modeling in Engineering and Sciences*, vol. 5, no. 6, pp. 477–496, 2004.