Cache Ensemble Analysis for Understanding Algorithmic Memory Performance

A.N.M. Imroz Choudhury and Paul Rosen

Abstract—We present an approach to studying ensembles of cache simulations which vary in cache features (such as size, associativity, block replacement policy, etc.) or qualities of the program (such as choice of algorithm, data storage layouts, etc.). Through the qualities of variation between their members, these ensembles can reflect the computational structure of programs and expose their performance characteristics, leading to better understanding of code and performance improvements that can be valuable in conserving scarce computing resources. We include several case studies looking at various cache performance scenarios, including some surprising performance bottlenecks in common programming operations, demonstrating the usefulness of our approach.

Index Terms—Cache performance, ensembles, visualization.

1 Introduction

The scarcity of high-performance computing resources has made performance a primary design goal in many applications. Hardware performance prediction [9] and software analysis [13] can be used to help guide new hardware deployments and software optimization efforts, but these approaches are approximate, because of the many complex interactions among computer subsystems. These subsystems, including the storage, network, functional, and memory units, are all simultaneously dealing with important events, such as incoming messages, interrupts, and shifting computational work loads which affect their performance. These conditions can vary across machines, operating systems, executions, and even from moment to moment within a particular run producing performance uncertainties.

The processing capabilities of computers has increased significantly faster than memory speeds, creating a imbalance in which keeping the processors fed with data from memory remains an important challenge. Memory caches are used to manage this speed difference, but careful use of the cache is required to achieve high performance. This makes the memory subsystem one major source of performance bottlenecks and uncertainty with many contributing factors. While some factors, such as algorithm selection, are controllable, many remain outside of a software developer's control, including the size and composition of data, end user hardware, and resource sharing with other system processes.

In this work, we present an approach for analyzing the performance uncertainties induced by design decisions in algorithms and memory caches, with the insight gained about memory performance eventually leading to software optimizations. By varying execution conditions, such as cache configuration parameters or algorithmic implementation details, we obtain several memory reference traces and simulated caches that, when combined, yield a *simulation ensemble*. Visualizing these ensembles can yield insight about cache performance characteristics of both software and hardware, and the relationship between the two. Figure 1 shows example of this approach for the bubble sort algorithm where the three graphs represent different ensembles for caches of different (a) total size, (b) L2 size, and (c) L1 size. In this example, we can clearly see the effect of cache size on performance as the size of the data working set shrinks in later portions of the execution.

2 Related Work

We briefly review the visualization of memory cache behavior and ensemble data.

- *A.N.M. Imroz Choudhury (roni@cs.utah.edu) and Paul Rosen (prosen@sci.utah.edu) are with the University of Utah*

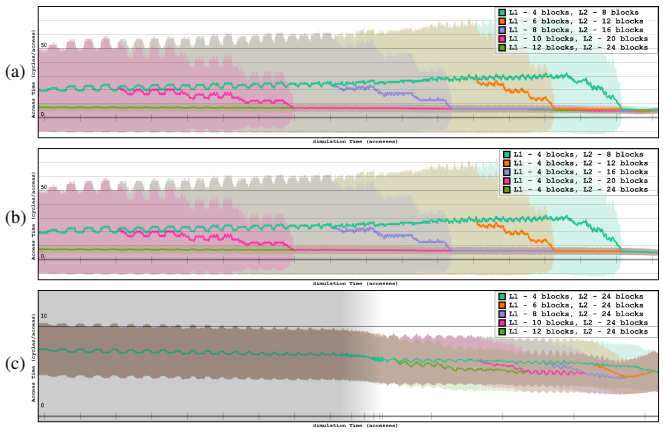


Fig. 1. An overview of our approach ensembles for bubble sort run through simulated caches which differ in size. (a)The total cache size is varied and the simulations show equal performance until the working set fits into the cache. (b) The size of the L2 cache is varied such that we can see similar results to varying the total cache size. (c) Only the L1 cache size is varied and, after some time, working sets begin to fit into L1, while the other is still only fit in L2. This graph also shows the use of a focus-and-context technique for zooming in on the end of the graph.

2.1 Memory and Cache Visualization

High performance software is important in many situations, and cache performance is integral component. There are several approaches to visualizing performance data and memory cache behavior. Vampir [11] and Tau [17] are visual profilers that collect broad runtime performance data, producing postmortem graphs using standard information visualization techniques. At the specific level of the memory subsystem approaches are available for visualizing runtime memory allocations along with their structural dependencies [1] and changing cache block residency [19], tracking cache misses and plotting them to help optimize software [15], visualizing the access patterns to various regions of memory [4] and cache dynamics, including data motion between levels of cache and evictions [5].

These techniques all report on a single execution with a fixed cache configuration. In the current work, we investigate the problem of analyzing variation in algorithms and multiple cache configurations by using ensembles. In particular, the ensembles are used to emphasize the differences between configurations generating insight about program behavior.

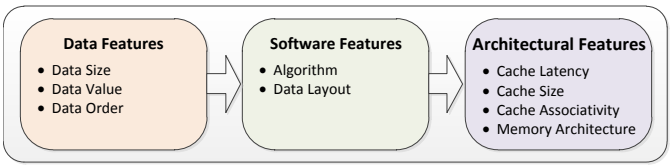


Fig. 2. Sources of cache performance uncertainty. The arrow indicates the flow of information through running software, from the end user, through software developers, and to the hardware that runs it.

2.2 Ensemble Visualization

Ensembles capture uncertainty by running a particular computation multiple times using different conditions and coordinating the results. A common example is in weather forecasting where each model is run multiple times with different initial conditions [14, 16]. By identifying where the ensembles agree and disagree, it becomes easier to make accurate forecasts and derive insight about why particular models might differ in their predictions. General techniques for visualizing uncertainty, whether from an ensemble or another source, include glyph-based approaches, colormaps, and overlaying uncertainty over the domain to be visualized [12].

In the current work, the model is an abstract representation of the levels of a cache, which the simulation outputs as changes to the state of the cache. As such, we cannot directly apply concrete techniques and instead opt for plotting statistics over various measures of the simulation results. Through these approaches, we can begin to bring insight about program behavior from differences between cache simulation ensemble members.

3 Cache Performance Uncertainty

There are three major sources of memory performance uncertainty, data, software, and architectural features (Figure 2). Most of these sources remain outside of the control of the software developer, yet the decisions made by a software developer can effect how robustly applications perform in uncertain conditions.

3.1 Data Features

The layout of data in memory can have a significant effect on cache performance. In most applications, the data should be stored in the same order that it will be accessed at runtime to promote good cache performance. However, the runtime access pattern may be variable or even unpredictable.

The first feature of the data which software developers must account for is the size. For example, if a developer chooses an algorithms which maintains a minimal memory footprint by streaming data off the disk, they are likely to see consistent performance across multiple platforms for data of any size. However, this might come at a significant performance cost in cases where all data fits on-core and a non-streaming algorithm can be used.

The values of input to a program can also affect computation. For instance, different sorting algorithms may have very different memory access patterns when sorting a particular kind of list, such as one that is in reversed order or one that is already sorted. By comparing ensembles that differ in the particular data values being operated on, such differences and similarities may be uncovered. This is similar to formalized 'best/worst/average case" algorithmic analysis.

3.2 Software Features

The only aspect of runtime execution under the control of the software developer is the design of the software. There are several software features that can give rise to ensembles, which in turn can help developers evaluate the memory performance characteristics of their choices.

Most computational problems can be solved in many ways, meaning that the developer must choose one of several algorithms to execute the solution. We demonstrate this idea by comparing different algorithms for sorting and matrix multiplication, focusing on their memory performance. For example, a naive matrix multiply is well suited for small matrices while blocked matrix multiply was designed with the purpose of improving memory performance for arbitrarily large matrices.

Aside from algorithmic differences, there are usually many ways to layout data in memory. For instance, matrices can be stored in row-major or column-major order. Similarly, data can be stored in multiple arrays or interleaved in structure. The patterns of access to memory can have a big impact on performance making data layout important to performance analysis.

3.3 Architectural Features

The various architectural features of caches, though normally outside the control of the developer, can still be useful to study. Varying the configuration can provide insight about why a particular program achieves poor performance in some scenarios and good performance in others. In some cases the insight gleaned may even suggest how to modify the code to be more robust to cache architectures and achieve better performance across more platforms.

Caches are made up of several *cache levels*, each of which contains a subset of the data in the next level, and which may independently vary in their attributes. Each cache level has a *size* dictating how much data the level can hold. Each level has a *block replacement policy*, which determines eviction to make room for new data. Finally, levels may have an *associativity*, which divides caches into subsets. We use the variation between different cache attributes to form simulation ensembles that in turn can expose the structures and behaviors of our programs, leading to insights about how those programs might be improved.

Though we focus on these particular features in our simulations, the variability in real-world cache systems is much larger. For example, programmable GPUs have a complex memory system including a large global memory and smaller banks of shared memory, visible subsets of threads. Some supercomputers have a "non-uniform memory architecture", in which accesses to different parts of the memory space may result in drastically different access times. Supercomputing clusters lack an explicit memory subsystem at the cluster level, instead relying on transfers of data between individual nodes to carry out their work. Our ensembles approach would certainly apply to such systems as well, as they have their own modes of variability, such as network topology, interconnect speed, etc., though these systems are significantly more complex to model.

4 Visualizing Cache Simulation Ensembles

This work is mainly interested in directly comparing simulation results by varying some parameter or quality of the simulations. Using straightforward visualization have been valuable in deriving insights about our case studies. Our approach plots performance as a function of time for all the ensemble members, on the same set of coordinate axes. Such a plot easily transmits the degree of agreement or dispersion between the ensemble members, and allows for quickly judging the difference in performance between several ensemble members. Clustering, which indicates insensitivity to the changing simulation parameters, is also easily visible.

Data Collection. We use Pin [10], a dynamic binary rewriting framework, to collect a memory reference trace—a record of all memory transactions made by a program at runtime. That memory trace record is then used as input to a home-grown cache simulator, which yields performance data. The cache simulator reports hit and miss information as memory transactions occur. That information is then used for the visualizations. To emphasize the much larger cost of accessing more distant memory, the hit level data is weighted by time to yield a cache service time for that simulation step. We used typical values for a real-world cache of 3 CPU cycles for L1 access, 15 cycles for L2 [6], and a much larger 300 cycles for main memory.

Ensemble Plots. In our plots, each ensemble member is represented by a curve showing the cache access time as a function of simulation time. There are few cache levels with little natural continuity from one reference trace record to the next, leading to high-frequency plots as cache misses are mixed in with cache hits. A moving average

is applied to the data to help reduce these effects and better investigate trends in the data.

Standard Deviation Plots. In some cases, we plot the standard deviation over the data as lighter-colored envelopes extending above and below the mean. This statistic is meant to capture the high frequency activity within the averaging window. In practice, high frequency activity generally means more cache misses, which inflates both the mean and the standard deviation, due the high access time to main memory. The standard deviation is therefore a redundant encoding of the information carried by the mean, and only serves to reinforce the qualities of the ensemble (for example, in Figure 1).

Difference Plots. Alternatively, it can be useful to investigate the general difference between ensemble members, as opposed to within a single member. In such cases, the standard deviation of the member values is a measure of dispersion or disagreement among the members (e.g. Figure 7), or single ensemble member can be used as a baseline value to find an optimal configurations (e.g. Figure 6 (d-f)). Generally speaking, other types of plots are also possible, but we leave an investigation of the full power of statistical techniques to future work.

Time Matching. When the ensembles differ in, for example, choice of algorithm, the ensemble members may represent different numbers of total memory accesses. For example, Figure 5 compares bubble and insertion sort which have similar computational structure but engage in numbers of memory accesses. In order to effectively visualize the differences between the simulations, they can be transformed into a common timeframe. This is done by using the source code to identify milestones at which to synchronize the simulations. For instance, bubble and insertion sort are synchronized at the ends of the sweeps through the data. This casts the simulation time into "source code time," in which source code is taken as the measure of elapsed logical time. Figure 5 has vertical lines to indicate the synchronization points. The insertion sort has been stretched to fit in the same timeframe as bubble sort, which is reflected in the relative size of the color-coded circular glyphs at the bottom.

User Control. Several aspects of plotting naturally fall under user control, mediated by user interface elements. For instance, the size of the moving average window can be changed at plotting time, giving users control over the smoothing effect of averaging and allowing them to search for performance features at different scales. We also use a focus-plus-context technique (Figure 1(c)) for zooming in on a section of the graph while deemphasizing the remainder, allowing for examination of details while still keeping a handle on the larger context.

5 Case Studies

We now illustrate our approach with several case studies, using various kinds of ensembles. Our case studies are divided into four groups: data features, algorithmic features, architectural features, and second-order ensembles.

5.1 Comparing Data Features

5.1.1 Data Order: Rendering Triangle Meshes

Triangle mesh rendering is a primitive action in many graphics applications. Meshes are often composed of point data describing vertices and indices describing edges and faces. The order of the indices determines the access order of the vertices directly effecting cache performance during triangle processing [20]. Figure 3 shows the cache performance during input assembly, using the indices to sweeping through the vertices for the Utah Teapot model. Ensemble members differ only in the order of the indices. The poorest performing run uses a deliberately shuffled dataset, while the middling performance is randomized, and the best member uses a sorted ordering. The initial part of the ensemble, in which all three algorithms give the same performance, comes from loading the data into memory. The remainder demonstrates how important good triangle ordering is to this application—a deliberately poor sorting order performs only slightly worse than a randomized list of triangles, while sorting the triangles, as expected, improves the cache performance roughly four times.

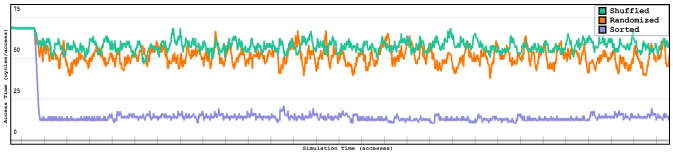


Fig. 3. Triangle rendering with input data ordered different ways. Randomized data (orange) and an adverse shuffling (green) perform poorly, with a roughly fourfold improvement upon sorting the data (blue).

5.1.2 Data Layout: Material Point Method

The material point method (MPM) [18] is a method for simulation of mechanical engineering problems, in which solid bodies are discretized into collections of particles that move, in response to forces, over a background grid. The grid nodes interpolate particle data to compute gradients. Each particle represents a bit of material, carrying attributes such as mass and velocity. The particles can be stored in memory as an array of structures, each storing the attributes of one particle, or in parallel arrays, each storing one attribute from every particle. There is a similar choice of storage policy for the data on the grid nodes as well.

Figure 4 shows an ensemble of the various policies, for both particles and grid nodes. Simulation time is partitioned into the phases of the MPM timestep, so that different performance behaviors can be correlated to source code. The ensemble shows two distinct cache behaviors—stretches of relatively good performance interrupted by bursts of poor performance. The better performance reflects process of the grid nodes, which can always be done in a fixed order, leading to sweeping access patterns with heavy data reuse. The short bursts reflect processing of the particles as they interact with nearby grid nodes. Since particles can move about the domain, such processing represents scattered accesses to the grid nodes, implying much poorer data reuse and therefore performance. Such bursts are short because the particles are much fewer than the grid nodes. However, the bursts also show a large variance in performance due to storage policy. In addition to the already scattered nature of memory access during the bursts, the struct storage policy tends to spread out these accesses across memory, further compounding the poor data reuse, leading to even worse performance.

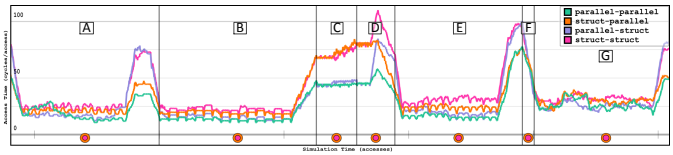


Fig. 4. Data storage policies for MPM. Particle and grid node data can each be stored in an array of C-style structures or in several parallel arrays. This ensemble shows all four combinations, with "parallel-parallel" being seen to perform the best.

5.2 Comparing Algorithm Features

5.2.1 Number of Memory Accesses: Sorting

Bubble and insertion sort both work by making repeated shrinking sweeps of the list to be sorted, leaving the largest of the remaining elements in its correct position at the end of each sweep. Figure 5 compares bubble and insertion sort, which have similar computational structure, but use memory differently. Bubble sort uses many write operations, as it uses repeated swaps to move elements, whereas insertion sort only performs a single swap per sweep, moving the largest element into place. While bubble sort appears to have much better cache performance than insertion sort (because all of its extra writes result in cache hits), insertion sort may actually have better overall performance simply because it performs fewer memory accesses.

In Figure 5, the traces have been time-matched to the ends of their sweep operations (top), with attendant scaling of the average access times (bottom). When accounting for bubble sort's extra memory

accesses, insertion sort appears to have better memory performance overall. This analysis demonstrates how a simple report of "cache hit rates" may be misleading when comparing alternate approaches that compute the same result.

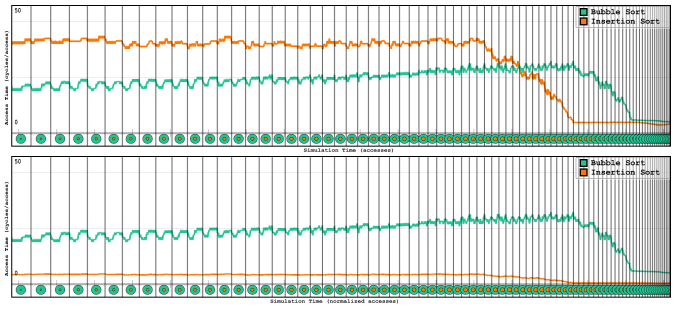


Fig. 5. While the bubble and insertion sort algorithms have similar computational structure, bubble sort makes many more in-cache memory accesses, giving the appearance of better cache performance (top). When the traces are appropriately scaled (bottom), bubble sort is seen to actually take longer because of its higher volume of memory accesses.

5.2.2 Data Access Patterns: Matrix Multiplication

Matrix multiplication is an important operation in many scientific programs. The naive algorithm for matrix multiply computes dot products of the rows of the left-hand matrix and columns of the right-hand matrix, introducing a cache-unfriendly access pattern for one matrix or the other depending the data storage orientation (row or column major). A simple solution is to store the left-hand matrix in row-major and the right-hand matrix in column-major order, transposing its access pattern to a cache-friendly one. Figure 6(a) shows that the "transposed multiply" has dramatically better cache performance.

Mixing data storage orders means that swapping a left-hand and right-hand matrix will lead to extremely poor access patterns. The more general cache-friendly solution is to use *matrix blocking*, in which submatrices are repeatedly multiplied, accumulating their products into the final result. The block sizes are chosen so they fit into cache, improving the reuse of the appropriate entries. Figure 6(a) also shows the relative performance of blocked matrix multiply compared to the naive algorithm and transposed algorithms. Interestingly enough, our results show that blocking performs worse than the transposed multiply. This is puzzling and counterintuitive, as blocking is known to be an efficient technique for matrix multiplication.

5.2.3 Effect of Associativity: Matrix Multiplication

When a program's access patterns show certain kinds of regularity, the cache may perversely end up exclusively using a limited number of its associative sets, leaving others idle, and leading to an increased miss rate. This is exactly the behavior we observe in blocked matrix multiplication. Our example uses a 16×16 matrix, with blocks of size 4×4, ***giving the starting address of each block the same modulo-4 address, thereby causing the start of every block to map to the same associative set in any cache using four sets (as our simulated cache does), roughly quadrupling the miss rate, and slashing performance in half (Figure 6(b)). – i don't think this is exactly true***

The visualization and analysis suggest a simple fix, stagger the mapped sets in each block of the matrix by simply inserting some amount of padding after in each row. With rows of 16 elements, amounts of padding from zero to fifteen excess elements per row are possible. Studying how much padding to use in this case is a prime example of the usefulness of cache ensembles. Figure 6(c) includes an ensemble member for each of the sixteen options. Figure 6(d-f) show differential versions of Figure 6(c), each one subtracting out a particular curve from all the ensemble members, allowing for direct comparison of value. As expected, a padding of two elements does a better job of staggering the block addresses than a padding of four. Padding with six extra elements does even a better job of redistributing the addresses throughout the sets of the cache, and seems to be the best solution in this case.

5.3 Comparing Architectural Features

5.3.1 Cache Size: Sorting

The size of a cache is a very important parameter—generally speaking, a larger cache suffers fewer cache misses, due to less contention for space. Simulation ensembles in which the cache size varies can be used to expose the size of the *working set* of an application, the total amount of memory the application requires during a given phase of its run. For example, consider the bubble sort ensemble in Figure 1. Each ensemble shows the average cache service time as a function of simulated cache time, while each curve represents a single ensemble member differentiated by size (total size for (a), L2 size for (b), and L1 size for (c)). Bubble sort works by making repeated, shrinking sweeps over the list to be sorted. At some point during each run, the working set becomes small enough to fit entirely inside the cache, at which point the cache performance improves dramatically. The effect of a changing cache size can be seen in this ensemble as the location of the sudden drop off in access time.

The pattern in Figure 1 is well-known and occurs in simple analyses of working set size in standard reference texts [7]. A more complex example can be seen in merge sort (Figure 7, top), which first works on small sublists, assembles them into larger sublists, and then recursively assembles those into yet larger sublists, etc. This algorithm therefore admits different working set sizes at different times during its run. By forming an ensemble of merge sort running with several different cache sizes, these working set sizes become visible. The relative distribution of the performance values may give insights about when relatively good or bad memory performance can be expected from such an algorithm. Figure 7 shows several peaks during the sort, corresponding to various sizes of sublists. The poor cache performance results from the incoherent access pattern of having to access two sublists simultaneously. The spectrum of cache sizes differentiates the various sizes of sublists, i.e. the varying working set size of the application. As the working set becomes large, the ensemble members begin to disagree in a regimented way about the cache performance. In fact, this notion generalizes to the standard deviation of the ensemble members (Figure 7, bottom), which summarizes the disagreement between the members, which in turn reflects the size of the working set in this case.

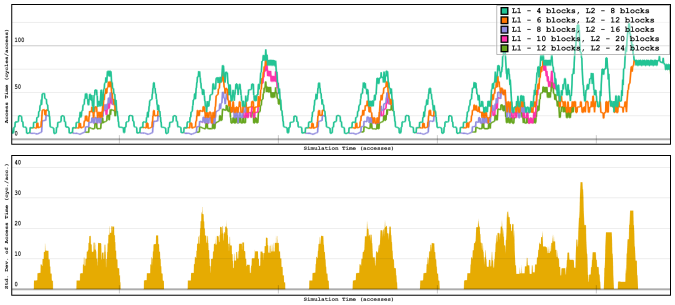
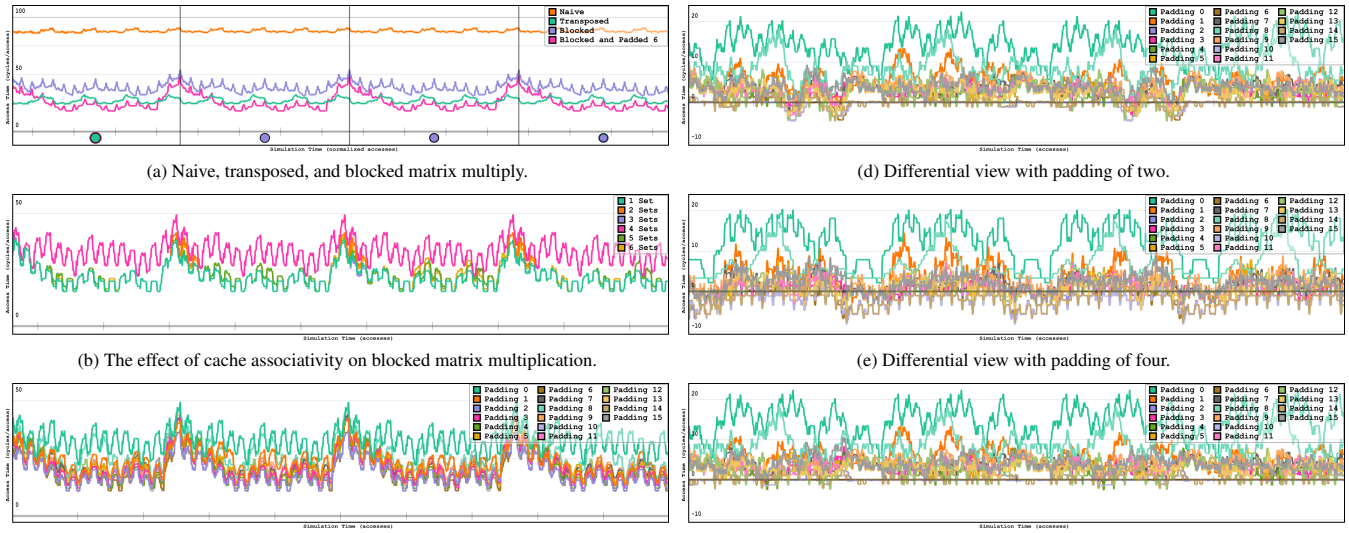


Fig. 7. Merge sort with different cache sizes. Top: The blue curve represents a simulation of a very small cache which cannot hold even the smallest merge lists. The peaks of poor performance in this member reflect the computational structure of the algorithm. With its fuller spectrum of cache sizes, the ensemble differentiates different working set sizes by disagreement between the its members. Bottom: The standard deviation of the ensemble members. The value falls to zero when all of the members agree, with positive values denoting disagreement. This statistic therefore summarizes the degree of disagreement, and therefore the working set sizes of the algorithm.

5.3.2 Block Replacement Policy: Diffusion Equation

The block replacement policy has an important impact on cache performance since it selects the block to evict when new blocks enter the



(a) Naive, transposed, and blocked matrix multiply.

(b) The effect of cache associativity on blocked matrix multiplication.

(c) The effects of padding on blocked matrix multiplication.

(d) Differential view with padding of two.

(e) Differential view with padding of four.

(f) Differential view with padding of six.

Fig. 6. Cache performance analysis of matrix multiplication. (a) The naive algorithm (orange) performs the worst, due to cache-unfriendly access patterns. The other members show different algorithms with better performance, but surprisingly, blocked multiply algorithm does not perform as well as expected. (b) Highlighting the effect of cache associativity on block matrix multiplication, accounting for the reduced performance. (c) Various padding inserted at the ends of the matrix rows helps redistribute cache activity around the associative sets. (d-f) The same data appearing in (c), but with the curves for padding amounts of two, four, and six subtracted uniformly out, respectively, with a padding of six seeming to offer the best gain.

cache. The best block replacement strategy, known as OPT [2], is impossible to implement in practice [1], an implementable approximation must be used instead. A commonly used policy, *least-recently used (LRU)*, evicts the block which was least recently accessed. LRU works well in practice, and is so common that Hill's groundbreaking work on analysis of caches and cache behavior [8], assumes it as a base feature of caches. However, there are cases where LRU is not the best policy, due to the structure of computations. Forming ensembles in which the replacement policy varies, it is possible to see the effect of different policies on the miss rate of a program. In general, OPT and its inverted, pessimal counterpart PES (which evicts the soonest-needed block, to cause as many replacement misses as possible) bound the performance of replacement policies, giving some idea of how much damage could be caused by the replacement policy.

Figure 8, top, shows an example of a case in which LRU may not be the best option. The program in this case is a diffusion equation solver that repeatedly sweep a data array, updating values using finite differences. For repeated sweeps of this type, *most-recently used (MRU)* is the optimal replacement strategy [3]. Figure 8 bears this out, as the ensemble member for the MRU cache closely matches that of the OPT cache.

The replacement strategy is out of the control of the developer. This means the diffusion equation solver is stuck with LRU, but the analysis suggests a possible corrective course of action. The trouble with LRU is that it tends to evict blocks that will be needed in the near future—if we were to reverse the order of updates periodically, we could try to "trick" LRU into behaving more like MRU. The program can be modified to use "pingpong" sweeps, proceeding from the first element to the last, then make the next sweep from last to first. By reversing the direction at intervals, LRU now tends to throw out blocks that will be needed *furthest* in the future. Figure 8, bottom, shows that LRU with the pingpong strategy performs about as well as MRU did with the original program: i.e., we found an optimal setting under the constraint of having to use LRU in the cache. This is a case where investigating the possibilities—even when they are out of our reach in practice—led to a vastly improved practical solution.

[1]Both OPT and PES block replacement strategies require full knowledge of future accesses to perform evictions.

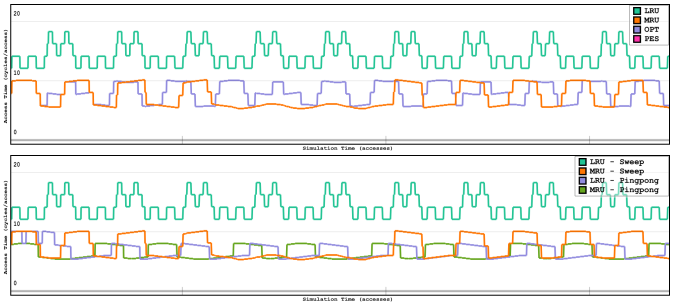


Fig. 8. Diffusion equation solver with different block replacement policies. Top: LRU performs poorly in this example, due to the repeated sweeping access pattern used in this algorithm. Bottom: By recasting the algorithm with an alternating "pingpong" sweep pattern, LRU now performs near-optimally.

5.4 Second-Order Ensembles

We have also experimented with combining members of an ensemble in meaningful ways to yield a new, second-order ensemble that offers its own insights about program behavior. Consider the ensemble of varying cache sizes in Figure 1. As discussed previously, this ensemble tells us something about the application working set over the program's run. However, we might also consider the varying sizes to reflect how much of the cache is alloted to the program during its run. Real computer systems are forced to share resources—for example, two concurrent programs must share the available cache. While one of them runs, it may evict blocks that belong to the other, inducing cache misses when the other program is scheduled to run again.

In some sense, this situation is reflected in the cache size ensemble. When a thread is scheduled to run, it may appear as though its cache allotment has been (temporarily) reduced—i.e., the performance profile will seem to have jumped from one ensemble member to a different one reflecting a smaller cache. To model such a situation, we can *combine* different ensemble members in a particular way. For example, we can take pairs of ensemble members whose cache sizes add up to some constant value; all such pairs of members can represent a two-thread model sharing a cache of that combined total size. Each

pair can be combined into a single performance curve representing two concurrent threads, and these combined pairs are then members of a new, second-order ensemble. Additionally, each second-order member is plotted with a light gray envelope behind it, representing the maximum deviation from the plotted value when the two first-order members are shifted from each other by some fixed amount of time. This represents the possible scheduling orders for the two threads, while the envelope bounds the performance of such orders. When many such envelopes are plotted over each other, the plotted color is a darker gray wherever many envelopes overlap. The darkness indicates how likely that regime of performance is to occur, in essence, bounding the performance of a bundle of ensemble members.

Figure 9 (a) shows such an ensemble. The values in each curve come from pooling the access time data from the pair of atomic ensemble members and treating it as though it came from a single run. The interesting feature in this example is the cluster formed by the large majority of ensemble members. Only when the size allocation is extremely lopsided (representing a "starvation" for one of the threads) does the performance change significantly, with most of the members forming a tight band of relatively well-performing curves. The essential insight is that this program seems to be robust to changes in its cache allocation—if it were made to be multithreaded, the cache would not present much of an obstacle to high performance.

The process generalizes to more threads, resulting in an ensemble like the one in Figure 9 (b). Here, we have combined four of the atomic ensemble members to create a varying four-way breakdown of the total cache space. There are too many ensemble members to enumerate their cache size breakdowns, but there are clearly four groupings of ensemble members forming four clusters. On further inspection, it turns out that these four clusters can be identified by the number of starvations present. The pink cluster, with the worst performance, allocates just eight cache blocks to three of the threads, allowing the rest of the cache to the fourth. Similarly, the blue cluster allocates two such small threads, the orange cluster only one, and the green cluster represents the optimal case where no single thread receives such a small allocation. It may seem obvious that starving a single thread leads to poor performance, but the real insight in this example is the *stability* of performance with respect to such extreme allocations. As long as no thread receives the minimal amount of allocation, the performance for all other combinations remains in a narrow band, which in turn indicates a kind of cache stability.

This is just one example of a higher-order ensemble that can be used to model and reason about more complex behaviors possible within computer systems. It is not meant to be an accurate model of the actual execution of such a behavior, but rather a way to reason about what might be expected of, for instance, cache performance, when such execution is set up. Due to the many uncertainties of concurrent execution, formulating a model of concurrent multithreaded execution using data from a single-threaded run is a powerful approach, and it therefore suggests that other complex systems behavior might be modeled from such data.

6 Conclusion

We have demonstrated an approach to investigating cache performance uncertainty by using cache simulation ensembles and techniques from information visualization to display them. By varying different features, we are able to elucidate reasons for poor performance, and suggest methods of changing the software to eliminate such performance problems. Our case studies clearly show some surprising results in unexpected places, and suggest practicable solutions to such problems where they arise.

One major avenue of future work lies in refining and improving our ideas about modeling multicore execution. Though the model we presented in this paper is not meant to be accurate, it will be useful to confirm some of our findings about, for example, robustness of multicore programs to varying cache allocation. Another area of future work is in designing specific visual encodings for the various types of patterns we have found in this initial work. It is promising that just with basic information visualization techniques, we are able to glean some insight about program performance; with dedicated, pattern-specific visualization solutions, we expect that our methods will become even more useful.

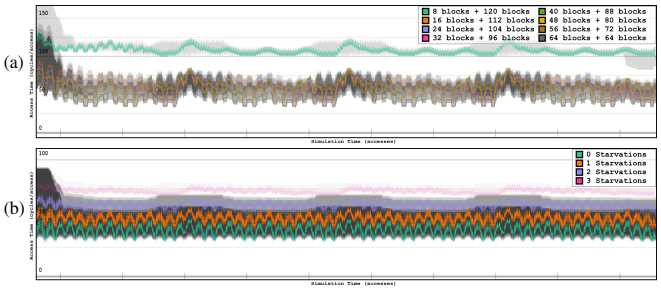


Fig. 9. Modeling multithreaded cache contention in matrix multiply, with (a) two threads and (b) four threads. This is a second-order ensemble formed from members of a simple cache size ensemble, by combining multiple first-order members whose sizes sum to a given total cache size. This ensemble therefore expresses the relative performance of differing allocations of available cache, modeling the sort of cache contention that might occur in a multithreaded code.

Uncertainty analysis has turned out to be an illuminating approach to cache performance analysis, making us think about program performance analysis in new ways, leading to hopefully more and more insights that will help run our programs more efficiently and effectively, increasing the value we derive from them. We see no reason this approach cannot be applied to other areas of performance analysis, hopefully delivering insights about even larger systems and the programs that run on them.

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*, pages 53–62, 2010.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, 1966.

[3] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In A. Pirotte and Y. Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 127–141. Morgan Kaufmann, 1985.

[4] A.N.M. I. Choudhury, K. C. Potter, and S. G. Parker. Interactive visualization for memory reference traces. *Computer Graphics Forum*, 27(3):815–822, May 2008.

[5] A.N.M. I. Choudhury and P. Rosen. Abstract visualization of runtime memory behavior. In *VISSOFT 2011*, 2011.

[6] A. Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, August 2011.

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

[8] M. Hill and A. Smith. Evaluating associativity in cpu caches. *Computers, IEEE Transactions on*, 38(12):1612 –1630, dec 1989.

[9] D. J. Kerbyson, A. H. J., A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 37–37, New York, NY, USA, 2001. ACM.

[10] 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*, pages 190–200, 2005.

[11] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[12] A. T. Pang, C. M. Wittenbrink, and S. K. Lodh. Approaches to uncertainty visualization. *The Visual Computer*, 13:370–390, 1996.

[13] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5:31–62, March 1993.

[14] K. Potter, A. Wilson, P.-T. Bremer, D. Williams, C. Doutriaux, V. Pascucci, and C. Johnson. Ensemble-vis: A framework for the statistical visualization of ensemble data. In *Data Mining Workshops, 2009. ICDMW '09. IEEE International Conference on*, pages 233 –240, dec. 2009.

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

[16] J. Sanyal, S. Zhang, J. Dyer, A. Mercer, P. Amburn, and R. Moorhead. Noodles: A tool for visualization of numerical weather model ensemble uncertainty. *IEEE Transactions on Visualization and Computer Graphics*, 16:1421–1430, 2010.

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

[18] D. Sulsky, Z. Chen, and H. Schreyer. A particle method for history-dependent materials. *Computer Methods in Applied Mechanics and Engineering*, 118(1-2):179 – 196, 1994.

[19] E. van der Deijl, G. Kanbier, O. Temam, and E. Granston. A cache visualization tool. *Computer*, 30(7):71–78, July 1997.

[20] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Trans. Graph.*, 24:886–893, July 2005.