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 either cache features (such as cache size, associativity, block replacement policy, etc.) or qualities of the program being simulated (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 types of cache performance uncertainty, including some surprising performance bottlenecks in common coding operations, demonstrating the usefulness of our approach.

Memory performance is often a bottleneck for overall performance, which in turn is important in many situations, such as large-scale programs running on shared supercomputers. We investigate *cache simulation ensembles* in order to yield insight about program memory cache performance. Our work uses cache simulation of program memory reference traces to produce cache performance profiles; by varying features of the simulated caches, and qualities of the programs being run, we are able to create ensembles of such simulations allowing us to look into the effect of such changes on performance. Software features that might change from run to run include choice of algorithms (which may access memory in different ways), data layout (which may affect the performance of access patterns upon that data), and the value and structure of the input data itself (which may cause the algorithm to react one way or another, affecting its memory performance). Cache features that may affect performance include the number and size of cache levels, associativity of each level, and block replacement policies. This poster presents several case studies of cache simulation ensembles that vary in one or more of these features, and explains the insights we gained from these studies.

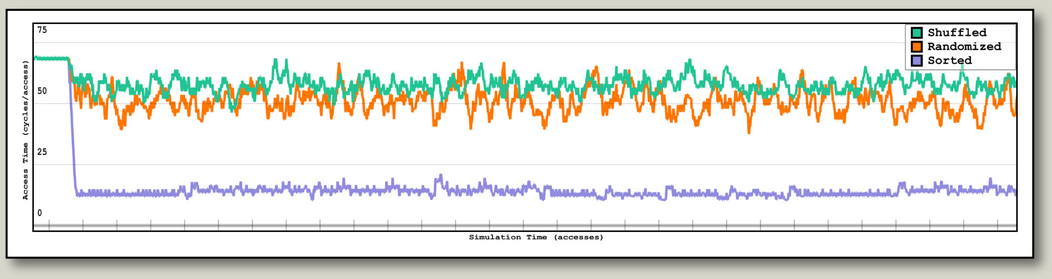


Fig. 1. Triangle rendering with input data ordered different ways. Triangle rendering requires repeated access into a data array describing the geometry of the triangles, making performance dependend on the order of storage. This ensemble shows three sorting orders for the input data: sorted numerically, randomly, and regularly shuffled. The initial phase of the ensemble, in which the data is simply being loaded from disk into memory, shows unanimous agreement. Then the drastic effect of storage order becomes visible. There is more than a twofold increase in performance in going from poorly or randomly sorted data to well-sorted data, and the performance is seen to be relatively stable in time for all three storage policies.

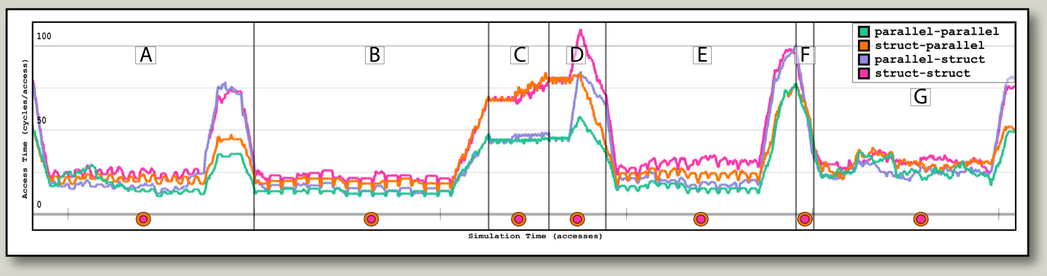


Fig. 2. Data storage policies for material point method. The algorithm keeps track of both particle and grid node data, with two ways to store each: in an array of C-style `structs`, or in several parallel arrays. Because the algorithm requires different patterns of access on the data at different times, we expect one or the other of these policies to perform better. This ensemble shows all four combinations of policies for the two kinds of data, with “parallel-parallel” being seen to perform the best. Note that the spikes of poor performance occur regardless of storage policy, indicating a particular feature of the computation that seems not to depend on storage policy at all.

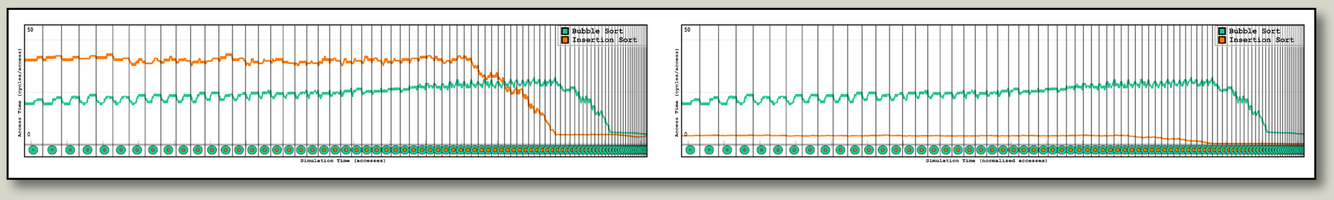
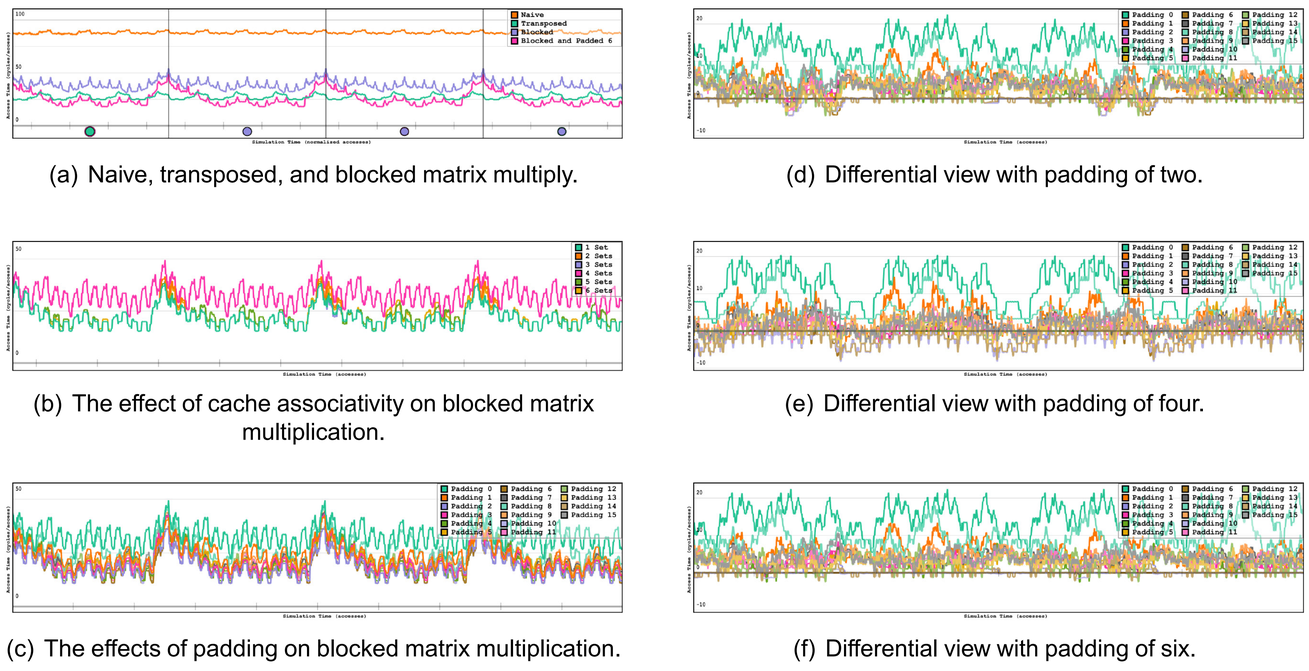


Fig. 3. Bubble vs. insertion sort. These sorting algorithms have very similar computational structure, both making repeated, shrinking sweeps of the list, placing one element in its correct position at the end of each. However, bubble sort makes many more memory accesses to accomplish this, as it repeatedly swaps elements towards the end of the array on each sweep. Because these swaps tend to occur within the cache, bubble sort appears to have better cache performance overall. However, when the two traces are time-matched (top) and appropriately scaled (bottom) by synchronizing the plots at known points in their source code, bubble sort is seen to actually take longer with respect to the cache, because of its overall higher volume of memory access. The circular glyphs at the bottom denote the amount of relative stretching of the traces required to time-match them.



(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. 4. Various incarnations of matrix multiplication. (a) An ensemble several basic forms of matrix multiplication. The naive algorithm (red) performs the worst, due to cache-unfriendly access patterns coming from the right-hand matrix, which is stored in row-major order but must be accessed by its columns. Storing the right-hand matrix in column-major order transposes the poor access pattern to a cache-friendly one, but such transposed matrices cannot be used as left-hand matrices in other computations. A more general solution is to use blocked matrix multiply—curiously, though well-known for its cache efficiency, it is seen not to perform as well as the transposed matrix multiplication. (b) The problem is that in this example, the starting addresses of the matrix blocks cause them all to be mapped into the same associative sets of the cache, disrupting cache performance. This ensemble shows the effect of reducing the associativity of the simulated cache. Note the large effect it has for this example, visible in the spread of ensemble members. (c) A simple solution is to stagger the block addresses by inserting unused padding at the ends of the rows of the matrices. This ensemble shows varying performance for different amounts of padding. (d-f) The same data appearing in (c), but with the curves for padding amounts of two, four, and six subtracted uniformly out, respectively. A padding amount of two staggers the block addresses effectively, while an amount of four seems to has a similar but smaller effect. With padding of six, we combine the effects of two and four, making near-optimal gains.

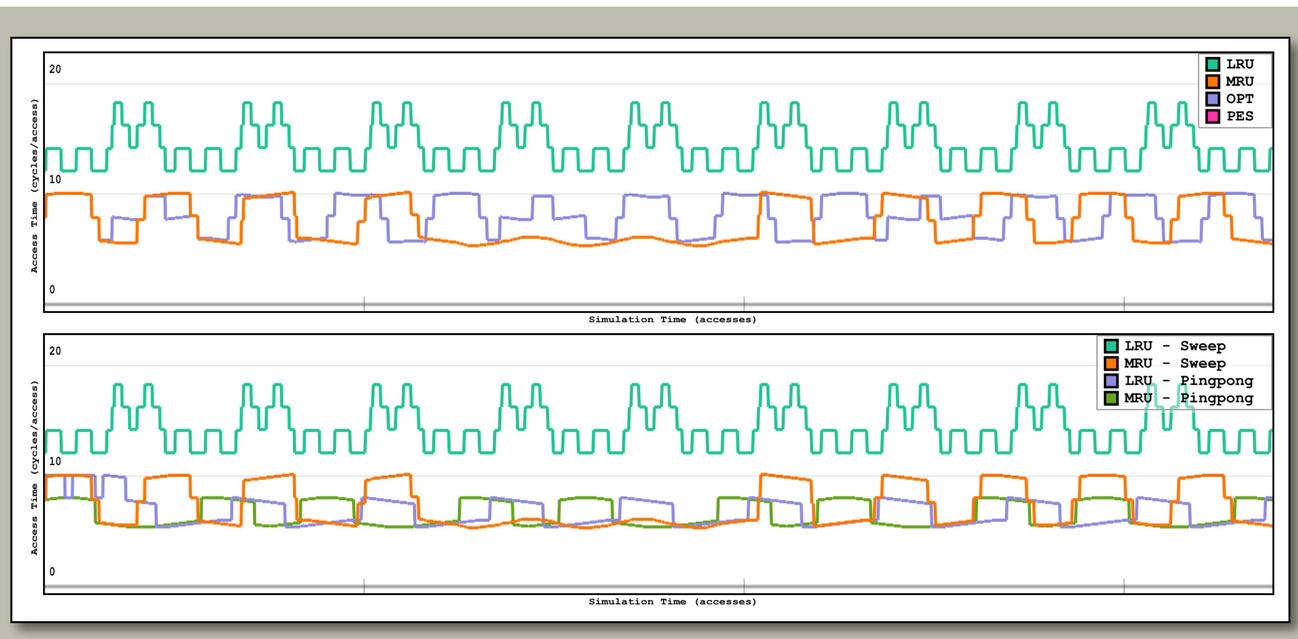


Fig. 5. Block replacement policies. Top: This ensemble shows the effect of block replacement policy on a scientific code that solves the one-dimensional diffusion equation. The algorithm works by making repeated sweeps of the data array, updating it for each computed timestep. Curiously, in this example LRU, a commonly well-performing choice, coincides with PES, the pessimal block replacement algorithm. This is due to the sweeping access pattern, which would actually prefer to evict the *most* recently used item in order to achieve optimality. MRU is seen to have similar performance to OPT, the optimal choice. Bottom: However, we can perform a trick to improve LRU’s performance: by “pingponging” our computation—sweeping from first to last, and then reversing for the next pass—LRU begins to look something like MRU, as the meaning of “least recent” and “most recent” swap roles. Not that LRU with pingponging performs at least as well as MRU. This is a case where the analysis suggests a simple change to the software to dramatically increase its cache efficiency.