Visualization Enables the Programmer to Reduce Cache Misses

Kristof Beyls and Erik H. D'Hollander and Yijun Yu
Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, Gent, Belgium
email: {kristof.beyls,erik.dhollander,yijun.yu}@elis.rug.ac.be

Abstract

Many programs execution speed suffer from cache misses. These can be reduced on three different levels: the hardware level, the compiler level and the algorithm level. Much work has been done on the hardware level and the compiler level, however relatively little work has been done on assisting the programmer to increase the locality in his programs. In this paper, a method is proposed to visualize the locality which is not exploited by the cache hardware, based on the reuse distance metric. Visualizing the reuse distances allows the programmer to see the cache bottlenecks in its program at a single glance, which allows him to think about alternative ways to perform the same computation with increased cache efficiency. Furthermore, since the reuse distance is independent of cache size and associativity, the programmer will focus on optimizations which increase cache effectiveness for a wide range of caches. As a case study, the cache behavior of the MCF program, which has the worst cache behavior in the SPEC2000 benchmarks, is visualized. A simple optimization, based on the visualization, leads to consistent speedups from 24% to 48% on different processors and cache architectures, such as PentiumII, Itanium and Alpha.

KEY WORDS
Data cache, program visualization, reuse distance, program optimization, software tools

1 Introduction

The execution time of many programs is dominated by cache stall time on current processors. In the future, this is going to aggravate due to the increasing gap between processor and memory speed. The processor speed is increasing by 60% per year, while the memory speed only increases at about 7% per year[6]. This leads to a memory wall which doubles every two years. Currently, a processor can typically execute a thousand instructions while fetching data from main memory.

Therefore, in order to keep the processor from being data-starved, it must be assured that the data locality in the program is exploited maximally by the data cache hierarchy. The two most occurring types of misses are the conflict and the capacity misses. The conflict misses are those misses that occur because the associativity of the cache is too small. The capacity misses are those that exist because the size of the cache is too small.

The optimization of the cache hierarchy utilization can be performed at three different levels:

- At the *hardware level*, the cache hardware could be improved. Most of the proposed techniques in the literature focus on reducing conflict misses by cheaply increasing the effective associativity of the cache. In order to decrease the capacity misses, the size of the cache should be increased. However, increasing the cache size makes it slower. Therefore, a tradeoff must be made between cache size and its response time. Currently, processors have a number of different cache levels, where the first cache level is small and fast and the levels below are increasingly larger and slower.

- Since the capacity misses are hard to resolve at the hardware level, they should be focused at the *compiler level*. At the compiler level, the conflict misses are diminished by improving the data layout, and capacity misses are handled by increasing the locality of capacity misses. However, previous research[1] has shown that state-of-the-art compiler technology removes 30% of the conflict misses and only 1% of the capacity misses in numerical programs such as those in SPEC95fp.

- A lot of cache misses exist, even after the hardware level has been optimized and the compiler has taken great effort to reduce them. The final optimization level is the *algorithm level*, which is controlled by the programmer.

In contrast to the extensive literature on cache hardware optimization and compiler optimizations for cache behavior, relatively little work has been performed on helping the programmer to optimize its programs cache behavior. Therefore, in this paper, we focus on supporting the programmer in his effort to reduce cache miss bottlenecks.

Several studies on different benchmark suites have shown that *capacity misses* are the most dominant category of misses[9, 1, 3]. However, as discussed above, at the hardware level and the compiler level, mostly the *conflict misses* are targeted. At the hardware level, capacity misses can only be reduced by making the cache larger, and generally, slower. At the compiler level, capacity misses can be reduced, but only for regular array-based loops. Little compiler work has been proposed to eliminate capacity misses for pointer-based irregular programs.

Because it is hard or impossible for the compiler to analyze or optimize a programs cache behavior, the job is delegated to the programmer. Of course, in order to be effective, the following objectives should be reached:

1. The cache behavior is not obvious from the source code. Therefore, a tool should show the programmer where the real cache bottleneck in the program lays. The visualization of the cache behavior by the tool should be program-centric[14], so that the programmer can relate the cache misses to program constructs. Also, if possible, the tool should give hints to the programmer about how to resolve the bottlenecks.

2. In the ideal case, the optimization should not be specific to a single platform, but it should result in improved execution speed, irrespective of the precise cache structures or processor micro-architecture the program runs on.

In order to reach the first goal, a tool should be devised which visualizes the cache behavior of the program. However, the amount of information about the cache behavior that can be recorded is huge. For example, each access to the memory could be recorded as a cache hit or a cache miss. However, since a program typically accesses the memory hundreds of millions of times per second, it is unfeasible to throw all this information unfiltered to the programmer. Instead, the cache behavior should be measured by a metric which allows to describe the cache bottlenecks accurately in a concise way. Ideally, the programmer should be able to identify the cache bottlenecks at a single glance.

Furthermore, in order to reach the second goal, the metric which is used to visualize the cache behavior must indicate the cache behavior bottlenecks, independent from the precise cache structure implemented in the hardware. For example, it should be displayed irrespective of the precise associativity of the cache or its exact size. These properties are found in the reuse distance, which indicates cache behavior, independent from cache parameters such as associativity or size.

The reuse distance metric is further discussed in section 2. The measurement and the visualization of the cache behavior, based on the reuse distance metric is presented in section 3. As a case study, the cache behavior of MCF, the program with the highest cache miss bottleneck in the SPEC2000 benchmark is shown in section 4. Based on the visualization, a small number of program transformations are proposed, which lead to a speedup of up to 48%, on Itanium, PentiumII and Alpha-processors. A comparison to related work is made in section 5, and a conclusion follows in section 6.

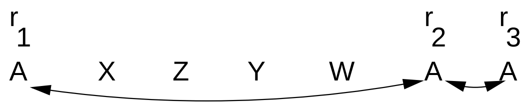


Figure 1. A memory access stream with indication of the reuses. A, W, X, Y and Z indicate the accessed memory location. The accesses to X, Z, Y and W are not part of a reuse pair, since W, X, Y and Z are accessed only once in the stream. The reuse distance of $\langle r_1, r_2 \rangle = 4$. The reuse distance of $\langle r_2, r_3 \rangle = 0$. The backward reuse distance of $r_1 = \infty$, the backward reuse distance of $r_2 = 4$.

2 Reuse Distance

Since the capacity misses are the dominant source of misses, and the hardware and compiler cannot reduce them very effectively, the programmer should focus on resolving those misses. The number of conflict misses is very dependent on both cache details, such as cache associativity, line size and cache size, and on compiler details such as how the data is layed out in the memory. Furthermore, the conflict misses can be reduced substantially by the hardware and the compiler level. Therefore, we wish to only present the potential capacity misses to the programmer.

The capacity misses can be represented by the reuse distance, irrespective of the actual cache size. The reuse distance is defined within the framework of the following definitions.

Definition 1. *A* **reference** *is a read or a write in the source code, while a* **memory access** *is one particular execution of that read or write.*

A **reuse pair** $\langle r_1, r_2 \rangle$ *is a pair of memory accesses in a memory access stream, which touch the same memory location, without intermediate accesses to that location. The* **reuse distance** *of a reuse pair* $\langle r_1, r_2 \rangle$ *is the number of unique memory locations accessed between references* r_1 *and* r_2.

Definition 2. *Consider the reuse pair* $\langle r_1, r_2 \rangle$. *The* **backward reuse distance** *of* r_2 *is the reuse distance of* $\langle r_1, r_2 \rangle$. *If there is no such pair, the backward reuse distance of* r_2 *is* ∞.

Example 1. *Figure 1 shows two reuse pairs in a short memory access stream.*

The reuse distance has the following property, which makes it an interesting metric for detecting capacity misses:

Lemma 1. *In a fully associative LRU cache with* n *lines, an access with backward reuse distance* $d < n$ *will hit. An access with backward reuse distance* $d \geq n$ *will miss.*

Proof. In a fully-associative LRU cache with n cache lines, the n most recently referenced memory lines are retained. When a reference has a backward reuse distance d, exactly d different memory lines were referenced previously. If

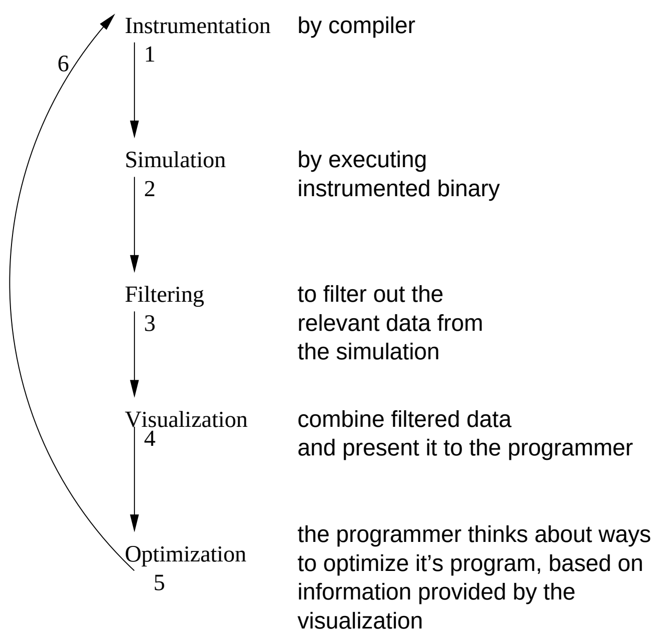


Figure 2. Overview of the measurement, visualization and optimization cycle for cache optimization.

$d \geq n$, the referenced memory line is not one of the n most recently referenced lines, and consequently will not be found in the cache. $\square$

Since all the cache misses in a fully associative cache are capacity misses, the backward reuse distance indicates what cache size is needed for a particular memory access to be a cache hit instead of a capacity miss. Furthermore, [1] showed that the reuse distance is also a good predictor of cache behavior for less associative caches, and even for direct mapped caches. Therefore, the reuse distance is a simple metric which, irrespective of cache parameters such as associativity or size, indicates the cache behavior of memory accesses.

3 Reuse pair visualization

The different steps in the visualization and optimization process are shown in figure 2. First, the program that needs to be optimized is instrumented to measure the reuse distances during the execution. In the second step, the instrumented program is executed and the reuse distance is measured. In the third step, the reuse distance information from the simulation is filtered, so that only those reuses leading to capacity misses are retained. In the fourth step, these long reuses are shown to the programmer, who can then start to think about ways to reduce the distance between use and reuse, in order to transform the capacity misses into cache hits. After optimizing his program, the programmer can measure it again, and try to resolve any left-over cache bottlenecks. The different steps are discussed in detail below.

Instrumentation

First, the memory access stream generated by the program is needed, so that the reuse pairs can be extracted from it. Furthermore, for every memory access, it is necessary to know which reference generated it, so that it can be tracked back to the source code. In our implementation, we extended the ORC-compiler[5], so that for every instruction accessing the memory, such as loads, stores and prefetches, a function call is inserted. The memory address accessed and the identification of the instruction generating the memory access are given to the function as parameters.

This instrumentation makes sure that the function is called for every memory access, and the necessary information about the memory location and the reference is given.

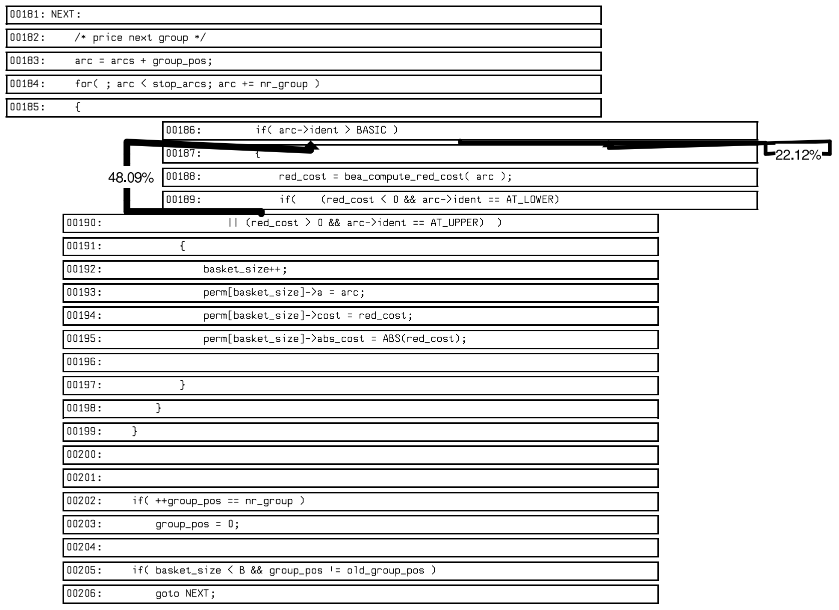
Simulation

The instrumented program is linked with a library which implements the function which is called on every memory access. This function could just store the memory trace to disk. However, this would lead to an enormous trace file on disk, since typical programs access the memory billions of times. Therefore, the trace is processed online. The backward reuse distance is calculated for the access, and the previous reference accessing the same location is looked up. For every pair of references in the program it is recorded how many reuse pairs with which reuse distance were measured during the execution of the program. Only this histogram of reuse distances per pair of references retained, which reduces the amount of data needed to be stored on disk. In our implementation, the data is stored in an XML-format. A short example of the XML-data is shown in fig. 3.

Filtering & Visualization

Lemma 1 indicates that only those reuses which are larger than the cache size generate capacity misses. Therefore, the reuse pairs with a short reuse distance are eliminated, so that only the long reuses are leftover. It is exactly those long reuses which generate cache misses. The filtering will filter out those reuse distances which do not fit into the cache size. This is easily implemented with an XSLT-filter[13] which transforms the raw XML-data measured in the simulation step. The result of the filter on the example data in fig. 3 is shown in fig. 4.

After this, exactly the interesting information for the programmer has been extracted from the program. In order to increase the efficiency of the programmer, the data should be shown directly in the source code. In this way, the programmer can easily analyze the long reuse distances and the program constructs which lead to those long reuse distances. The long reuse distances are shown graphically by arrows between source and sink in the source code. In our prototype implementation, the VCG-graph layout



```
00181: NEXT:
00182:     /* price next group */
00183:     arc = arcs + group_pos;
00184:     for( ; arc < stop_arcs; arc += nr_group )
00185:     {
00186:         if( arc->ident > BASIC )
00187:         {
00188:             red_cost = bea_compute_red_cost( arc );
00189:             if(    (red_cost < 0 && arc->ident == AT_LOWER)
00190:                 || (red_cost > 0 && arc->ident == AT_UPPER)  )
00191:             {
00192:                 basket_size++;
00193:                 perm[basket_size]->a = arc;
00194:                 perm[basket_size]->cost = red_cost;
00195:                 perm[basket_size]->abs_cost = ABS(red_cost);
00196:
00197:             }
00198:         }
00199:     }
00200:
00201:
00202:     if( ++group_pos == nr_group )
00203:         group_pos = 0;
00204:
00205:     if( basket_size < B && group_pos != old_group_pos )
00206:         goto NEXT;
```

Figure 5. Visualization of long-distance reuses in MCF, as produced by VCG. The visualization is a zoom of the locations where the majority of the long reuse distances occur. For 48.09% of all long distance reuse pairs, the first access is generated by `arc->ident` on line 190 and the second access is generated by `arc->ident` on line 186. Furthermore, For 22.12% of the long distance reuse pairs, the first and second accesses are both generated by `arc->ident` on line 186. So, 70.21% of all capacity misses occur on the access of the `ident` field of the variable pointed to by `arc` on line 186.

```
<reference id="pbeampp.c/primal_bea_mpp:21">
  <reuse>
    <log2distance>16</log2distance>
    <fromid>pbeampp.c/primal_bea_mpp:21</fromid>
    <count>3310601</count>
  </reuse>
  <reuse>
    <log2distance>17</log2distance>
    <fromid>pbeampp.c/primal_bea_mpp:21</fromid>
    <count>109607</count>
  </reuse>
  <reuse>
    <log2distance>18</log2distance>
    <fromid>pbeampp.c/primal_bea_mpp:21</fromid>
    <count>513041</count>
  </reuse>
  <reuse>
    <log2distance>19</log2distance>
    <fromid>pbeampp.c/primal_bea_mpp:21</fromid>
    <count>13477191</count>
  </reuse>
  <reuse>
    <log2distance>20</log2distance>
    <fromid>pbeampp.c/primal_bea_mpp:21</fromid>
    <count>7218189</count>
  </reuse>
</reference>
```

Figure 3. Example of some reuse distance data, recorded for the MCF program, before filtering. Only those reuses for which both the first and second access are generated by the 21st memory instruction in the `pbeampp.c` source file are shown here. The `log2distance` field shows the $\log_2$ of the measured reuse distance, the `count` field shows the number of times the reuse felt into this category.

```
<reference id="pbeampp.c/primal_bea_mpp:21">
  <reuse>
    <log2distance>15</log2distance>
    <fromid>pbeampp.c/primal_bea_mpp:21</fromid>
    <count>24628629</count>
  </reuse>
</reference>
```

Figure 4. The same reuse distance data as in figure 3, after filtering with reuse distance ≤ 15 has been applied. This data leads to the edge with the 22.12%-label in the visualization in fig. 5

tool[7] was used to draw the long reuse distance arrows. An example of the resulting graph is shown in fig. 5.

Program Optimization

The previous steps were all automatically performed by the computer. Now, based on the measured reuse distances, it must be tried to reduce the distance between use and reuse for long reuse distances, which decreases the number of capacity misses. In the introduction, it has been argued that the compiler or the hardware cannot do this effectively. Therefore, programmer interaction is needed, since he knows how his program works, and how he can restructure the program in order to reduce long reuse distances. An example of an optimization is shown in the case study, in the next section.

```
  for( ; arc < stop_arcs; arc += nr_group )
  {
    /* prefetch arc!!*/
#define PREFETCH_DISTANCE 8
    PREFETCH(arc+nr_group*PREFETCH_DISTANCE);
    if( arc->ident > BASIC )
    {
      red_cost = bea_compute_red_cost( arc );
      if( bea_is_dual_infeasible( arc, red_cost ) )
      {
        basket_size++;
        perm[basket_size]->a = arc;
        perm[basket_size]->cost = red_cost;
        perm[basket_size]->abs_cost =
                             ABS(red_cost);
      }
    }
  }
```

Figure 6. The optimized code for the MCF program. A single prefetch instruction was inserted.

processor	L1 (size,assoc)	L2 (size,assoc)	L3 (size,assoc)
PentiumII	(16KB, 4)	(256KB, 4)	not present
Itanium	(16KB, 4)	(96KB, 6)	(2MB, 4)
Alpha 21264	(64KB,2)	(8MB, 1)	not present

Table 1. Cache sizes and associativity for the different processors.

processor	original (seconds)	optimized (seconds)	speedup
PentiumII	147s	105s	40%
Itanium	98s	66s	48%
Alpha21264	56s	45s	24%

Table 2. The execution times and speedup of the original and the optimized MCF-program, on three different processor architectures.

4 Case Study

Here, the long reuse distances for the MCF program are shown and the program is optimized. MCF is the program from the SPEC2000 benchmark which has the highest cache bottleneck. On an Itanium processor, even after full compiler optimization, this processor is stalled waiting for data to return from the memory about 90% of the execution time.

In fig. 5, the majority of the cache misses are shown in the code. The figure shows that about 70% of the capacity misses are generated by a single load instruction on line 186. The best way to solve those capacity misses would be to shorten the distance between use and reuse. However, after analyzing the code a bit further, it is obvious that the reuse of arc-objects do not occur within a single iteration of the `for`-loop on line 184. Additionally, the reuse doesn't even occur between iterations of the outermost loop which goes from line 181 to line 206. The reuse occurs between different invocations of this function. So, bringing use and reuse together would need a thorough understanding of the complete program, which we do not have, since we didn't write the program ourselves. Therefore, instead of removing the capacity misses, we tried to hide them using data prefetching. We decided to try to prefetch the data that is touched by the `arc`-pointer on line 186. The optimized code is shown in figure 6.

The optimized code was compiled on 3 different processor architectures: PentiumII, Itanium and Alpha21264. For the PentiumII and the Itanium, the Intel compiler was used, for the Alpha the Alpha compiler was used. For all the experiments, the highest level of compiler optimization was chosen. The execution times and speedups of the original and optimized codes are shown in table 2. The table shows that the insertion of two lines into the source code was able to speed up the program between 24% and 48% on CISC(PentiumII), RISC(Alpha) and EPIC(Itanium) processors.

5 Related Work

Most work on visualizing performance bottlenecks for the programmer has been done for parallel programs[15, 4, 11, 8, 10]. These visualizations mostly focus on visualizing the communication patterns between the parallel parts in the program. In contrast to visualization for parallel programs, relatively little work has been proposed to visualize cache bottlenecks. In [2], the cache behavior is visualized through statistical histograms of the cache lines. The histograms show which cache lines are most frequently used. In [12], the cache lines are visualized, and the contents of that cache line are indicated by a color. Every time the contents of an address is copied into a cache line, the color of that cache line is updated so that it represents the cached address. Both [2] and [12] visualize the cache behavior cache-centric, i.e. the underlying cache structure and its operation is visualized. This doesn't allow to clearly visualize the cache behavior of the whole program, because the cache contents is frequently refreshed and the huge data space of a program is observed through the tiny cache window. This problem is avoided in [14], where the cache behavior is visualized program-centric. The program locality is shown by assigning a single pixel to every memory access. The color of the pixel indicates whether the corresponding access was a hit or a cold, conflict or capacity miss. Furthermore, it is possible to relate the visualized memory trace with the source code. However, this visualization is only feasible for programs which generate short memory access traces. Furthermore, since the hits and misses are recorded for a particular cache, the programmer is not steered to optimize the locality independent of the cache parameters. In contrast, this work is able to visualize memory access traces of arbitrary length. Furthermore, since the reuse distance is independent of the precise cache parameters, it allows the programmer to clearly see the cache bottlenecks common to a wide range of caches.

6 Conclusion

The discrepancy between processor and memory speed affects processor performance substantially. On top of that, the speed difference is doubling every two years. Therefore, all possible means must be used to diminish the speed degradation due to cache misses. In the past, much work has been done on improving hardware and compiler techniques to reduce cache misses. However, there are still a substantial number of cache misses left over. Especially the capacity misses are hardly reduced.

In this paper, it is proposed to complement the hardware and compiler techniques with programmer-driven program optimizations to improve the data locality. However, the cache misses are not obvious from the source code, and therefore a tool must be devised which clearly indicates the causes of poor cache behavior in the source code. In order to make sure that the indicated cache bottlenecks are the bottlenecks for a wide range of cache configurations, the reuse distance was used to measure the programs data locality. It has the advantage that it is independent of cache size and associativity, and it predicts cache behavior for a wide range of cache architectures. The visualization of the long reuse distances steers the programmer to locality optimizations which are independent of the underlying cache. As a case study, the MCF program from SPEC2000 was studied. A simple optimization, based on the visualization, resulted in a speedup between 24% and 48%, on CISC, RISC and EPIC processors with different underlying cache architectures, even after full compiler optimization. This shows that the reuse distance visualization gives a good insight in the poor locality patterns in the program, and enables portable and platform-independent cache optimizations.

Acknowledgements

This research was supported by the Flemish Institute for promotion of scientific and technological research in the industry (IWT).

References

[1] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, 2001.

[2] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A flexible environment for computer systems visualization. *Computer Graphics-US*, 34(1):68–73, Feb. 2000.

[3] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[4] W. M. Jr., T. J. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in carnival. In *ACM SIGMETRICS Symp. on Parallel and Distributed Tools*, page 1, May 1996.

[5] Open research compiler. http://sourceforge.net/projects/ipf-orc.

[6] D. A. Patterson and J. L. Hennessy. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1995.

[7] G. Sander. Graph layout through the vcg tool. In *DIMACS International Workshop GD'94, Proceedings, Lecture Notes in Computer Science 894*, pages 194–205, 1995.

[8] S. R. Sarukkai, D. Kimelman, and L. Rudolph. A methodology for visualizing performance of loosely synchronous programs. *Journal of Parallel and Distributed Computing*, 1993.

[9] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In B. D. Gaither, editor, *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, volume 21-1 of *Performance Evaluation Review*, pages 24–35, New York, NY, USA, May 1993. ACM Press.

[10] B. Topol, J. Stasko, and V. Sunderam. Pvanim: A tool for visualization in network computing environments. *Concurrency: Practice & Experience*, page 1197, 1998.

[11] S. J. Turner and W. Cai. The 'logical clock' approach to the visualisation of parallel programs. In *Proceedings of Workshop on Monitoring and Visualization of Parallel Processing System*, 1992.

[12] E. Vanderdeijl, O. Temam, E. Granston, and G. Kanbier. The cache visualization tool. *IEEE Computer*, 30(7):71, 1997.

[13] W3C. Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt.

[14] Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of cache on the program execution. Ingezonden naar Information Visualization 2001.

[15] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.