Optimization-Oriented Visualization of Cache Access Behavior

Jie Tao and Wolfgang Karl

Institut für Rechnerentwurf und Fehlertoleranz,
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{tao, karl}@ira.uka.de

Abstract. Cache performance is regarded as a critical issue that significantly influences the overall performance of a computer system. Hence, optimization with respect to cache behavior forms an important topic in both research and commercial areas. This work aims at helping the user in the task of cache behavior analysis with a comprehensive visualization tool. The resulted cache visualizer is capable of presenting the various aspects of cache accesses, and shows the performance data in a high-level, user-readable way. More specially, it directly leads the user to the data structures where excessive cache misses occur. It then further depicts the reason of theses misses. In addition, it helps the user to decide important parameters concerning some optimization techniques, e.g. padding stride and size of cache blocks. Examples of initial optimization based on the visualizer have proven the feasibility of this visualization tool.

1 Introduction

Cache memories have been long introduced in the computer system to relax the widening gap between the memory and processor speed. However, due to poor cache locality, applications still face performance problems caused by long memory access latency. Locality optimization is therefore becoming increasingly important in current systems.

As a result, various approaches have been proposed for enhancing the cache efficiency. These approaches can be roughly divided into two categories: compiler based data/code restructuring and explicit optimization within source codes. The former is done transparently at compile time through code transformation [4], data structure padding [6], or both of them [1], based on heuristic analysis and mathematical frameworks; while the latter [2, 5] is achieved by manually rewriting the program code using the same techniques as compilers.

Since compiler-based approaches have to integrate both analysis and optimization into the complex structures of modern compilers, programmers prefer to using explicit optimization that can be directly performed in the source code. This, however, needs a full understanding of applications' runtime access patterns with respect to the cache behavior, which can be quite tedious due to the complex data structure of applications and the memory hierarchy.

Therefore, a set of cache visualization tools have been developed in order to help the user in this task. These tools, however, either are not optimization-oriented or provide

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 182–190, 2005.

only limited information, and hence can not really indicate what and where the cache problems are and how to tackle these problems.

We designed and implemented a new cache visualizer with the goal of showing the various aspects of cache accesses and behavior. It displays the performance data in a high-level, user-readable way and can lead the user to the data structures where excessive cache misses occur. It then depicts the reason of theses misses and helps the user to deduce reasonable values for parameters used in different optimization schemes. More concretely, this cache visualizer can:

- depict the bottlenecks where excessive references are performed on the main memory. This view allows both an overall observation of the complete working set and a focus on specific memory regions or code segments, like individual functions, loops, and iterations.
- show the different access behavior in various program phases using 3D views. This allows the user to have a deeper understanding of the cache performance and further to perform more detailed optimization on a per-phase basis.
- exhibit the sources of cache misses in terms of data structures and give information that allows the user to decide parameters for array and code transformation.
- illustrate the impact of optimizations, which helps the user to deploy appropriate optimization techniques or to improve the existing schemes.
- show the behavior of cache line invalidation and false sharing for shared memory multiprocessor systems. This allows additional performance gain that can be introduced by optimizing cache coherence protocols.

Overall, we have developed a comprehensive cache visualization tool. It focuses on the presentation of cache problems in a user-understandable way. Users benefit from it in easy detection of access hotspots and deduction of corresponding optimization schemes.

The remainder of this paper is organized as follows. Section 2 gives a short introduction of related work in this research area. This is followed by a brief description of the design requirements for a comprehensive cache visualizer in Section 3. Then in Section 4 the resulted cache visualizer is described in detail. The paper concludes in Section 5 with a summary and some future directions.

2 Related Work

In the field of cache visualization, several tools have been developed. The cache visualizer described in [9] is an example. Similar to our cache visualizer, this tool aims at helping the user to optimize the source code towards less cache misses. For this, it provides two graphical views showing the cache access behavior. The first view is called "cache miss pattern" and is used to illustrate all memory references, each presented with a colored pixel that depicts whether an access is a cache hit or a kind of cache miss. The other view shows the reuse distance which is defined as the number of distinct memory lines fetched between two memory accesses to the same memory line. Again all memory references are illustrated within a single diagram, using different colors to represent

the reuse distance for each access. This view aims to show the feature of capacity misses for fully associative caches, helping the user thereby to detect an appropriate cache size for the applications. However, both views are low-level and not possible of leading the user to form a picture about how the cache behaves at the runtime.

CVT [8] is another cache visualization tool with the focus on showing the behavior of interference misses. It provides two windows, one for showing statistics on cache misses in terms of arrays, and the other, its main window, is a grid illustrating the content of cache lines holding the elements of a working array. The goal of the latter is to allow the detection of elements in different arrays that are mapped to the same cache line causing conflicts. However, this approach is restricted to directly mapped caches, which do not represent the technology of modern processors.

The Intel VTune Performance Analyzer [3] is regarded as a useful tool for performance analysis. It provides several views, like "Sampling" and "Call Graph", to help identify bottlenecks. For cache performance it only shows number of cache misses. Even though this information can be viewed by process, thread, module, function, or instruction address, it is not sufficient for optimizations with respect to cache behavior.

Hence, we implemented a comprehensive, optimization-oriented cache visualizer capable of showing the different aspects of runtime cache behavior. We use high-level, user-understandable graphical views to allow not only the presentation of access hotspots, miss reason, and impact of optimization, but also the deduction of important parameters for code and data transformation. In the following, we first discuss the design requirements for a comprehensive cache visualization tool and then describe this cache visualizer in detail.

3 Design Challenges

A visualization tool, which targets at helping the user to understand the behavior of cache operations and the runtime activity of the code, has to fulfill a set of user requirements. First, it must provide users with both an overview of the complete application access behavior, as well as views of the detailed behavior within specific execution regions, such as a single loop nest, individual function, or specific data segments. This allows the user to begin with a high-level behavioral view, and then to successively refine the focus on hotspots and bottlenecks.

An optimization-oriented tool also must map the gathered behavioral data back to data structures within the source code. Since memory addresses are not visible within an application program, this correlation is necessary for the user to directly perform the optimization on the arrays within the source code.

In addition, access behavior often changes in different phases of an application, and the visualization of cumulative behavior alone may actually hide these differences. A tool therefore needs to show access patterns in different program phases, which may either be explicitly specified or may be identified by barriers, locks, or other synchronization mechanisms. This allows the user to detect the specific locality requirements of each execution phase, and hence enables more comprehensive optimization strategies.

Finally, a cache visualization tool must be not only capable of showing access bottlenecks but also able to give the reason of those cache misses. Only this enables the

detection of exact target to optimize and the selection of appropriate optimization techniques. However, most cache visualization tools fail to do this.

4 The Cache Visualizer

According to the design requirements described above, we implemented an optimization-oriented cache visualizer. In this section, a toolset for acquiring cache performance data is first introduced and then we go into the details of related issues with this visualization tool.

4.1 Source of Performance Data

We deploy CacheIn [7], a monitoring and simulation framework, to acquire performance data. CacheIn, at the top level, is comprised of a cache simulator, a cache monitor, and a software infrastructure. The former takes memory references as input and then models the complete process of data searching in the whole memory hierarchy with multilevel caches. According to the search result, it generates events for the cache monitor, e.g. hit/miss, load, replacement, and invalidation events.

The cache monitor defines a generic monitoring facility capable of being connected to any location of the memory system. It provides mechanisms to filter and analyze the events and then generates low-level monitoring data. More specially, it offers a set of working modes allowing the acquisition of different kind of cache performance data. This data is further delivered to the software infrastructure which produces higher level information in statistical form like histograms and total numbers. This forms the basis for the cache visualizer to exhibit the cache accesses in different aspects.

4.2 Visualization Infrastructure

The cache visualizer uses a client-server infrastructure to coordinate the interoperation between the target system, the data generator (CacheIn), the users, and the visualization facilities.

As the core of this infrastructure, the server receives the users' view selections and the monitoring data. According to the views, it then processes and prepares the data into required form and initiates the visualizer to generate the specified graphical view. The server is capable of handling user requests concurrently to processing the delivered monitoring data. Built on top of the server, the visualizer shows the required information in well understandable representations.

In addition, a graphical user interface is deployed for execution control, view selection, and on-line steering. This GUI consists of two different parts: the first part allows the specification of commands and parameters for executing an application, while the second part allows the user to select a graphical view of interest. The control information for execution is delivered to the target system and the visualization requests are transfered to the server. This means that the machine running the application under observation can be run separately from the machine performing the actual visualization allowing the use of special rendering equipment wherever it is available.

4.3 Available Views

The cache visualizer provides a set of views to show the various aspects of the cache behavior. This includes a few initial views based on virtual addresses and several higher level representations concerning data structures and dynamic accessing patterns. Due to restricted space, only a few sample views will be depicted in the following after the view description.

Access Histogram. First, a 2D diagram shows the overall access histogram covering the complete working set. It uses different colors to present the locations (cache or memory) of all memory references to a single memory line or array element. This allows hence to highlight the regions where large amount of accesses to the main memory are performed.

Phase Information. A 3D view shows a set of access histograms, each for a single program phase that can be identified by iterations and synchronization operations, or for an individual code region that is specified by functions and routines. Corresponding to each phase, a 2D view, like described above, is combined. The 3D view hence enables a detailed insight into the access pattern of different phases and directs a per-phase optimization capable of handling the different access behavior existing in different program phases and functions. For example, the first program phase usually handles the initialization of working arrays and hence varies from other phases in access behavior. Other phases can also behave differently. A specific optimization with respect to individual phase can therefore further introduce performance gains.

Cache Miss Reason. The previous views allow the user to find access bottlenecks. This is further supported by another 2D view which is used to exhibit the reasons causing the bottlenecks. For each critical memory region it shows the proportion of three main miss sources: first reference, replacement, and cache line invalidation. This aids the user to choose an adequate technique for tackling the bottlenecks. The following two views supply more information for this.

Address Series: This view shows the repeated series of access addresses in different lengths for a user specified start address. According to the frequency of each series users can decide which access addresses should be grouped and placed chronologically in the main memory. Such optimization can significantly reduce cache misses because loading the first address in a series also bring the next required data into the cache.

Memory Operation Histogram: This view shows the operations performed on a specified cache area, normally the critical region where inefficiency occurs. The operations include loads, hits, and replacements. Loads represent the loading of a line into the cache, hits imply cache reuse, and replacements indicate the data exchange. Users can specify a histogram either for showing all kinds of events or a single. The change of status with each cache line within the displayed area is highlighted in different color, allowing to detect those lines with heavy traffic – data is frequently loaded into and removed from. Actually, these lines shall be held in the cache during the whole execution. In addition, this view also presents the overmapping of reused data, aiding the user to

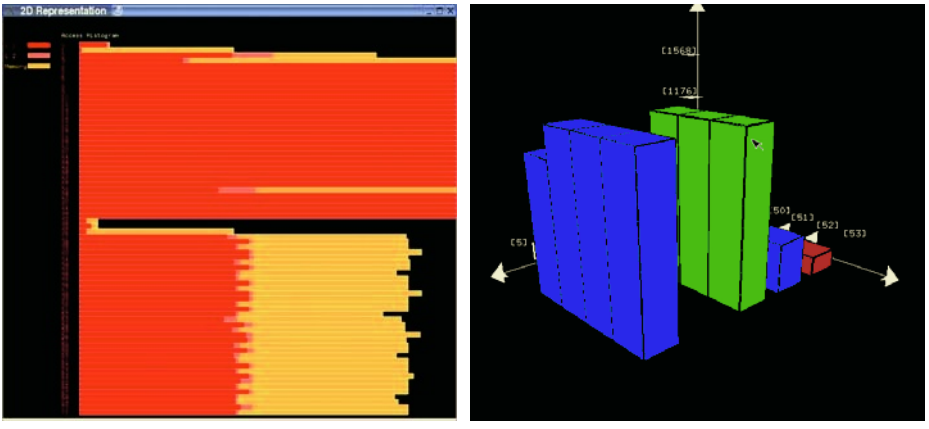

Fig. 1. Sample views: 2D overview (left) and 3D phase (right)

decide some parameters for optimization such as padding stride, size of blocking, and depth of code permutation.

False Sharing. This view shows the operations to each word within a specified cache line. Together with the accessed word, the type of operation and the processor performing this operation are also presented. Relying on the operations presented one after another, it is easy to find false sharing where a cache line is first written by a processor and then read by another processor but targeting a different word within this cache line. In this case, the corresponding line in the cache of the second processor has not to be invalidated. Therefore, this view helps to reduce cache line invalidations and thereby also cache misses.

Figure 1 depicts a 2D Access Histogram and a 3D Phase Information. As mentioned, the Access Histogram gives an overview of the access distribution on the memory system. This can be based on virtual memory lines or data structures. The concrete diagram in Figure 1 is an example of the former and presents an access histogram on a memory hierarchy with two caches. Corresponding to each virtual memory line, which is labeled left to the diagram, three colored bars show the number of accesses performed at each location. Access hotspots, where a large amount of accesses are fulfilled on the main memory, like the last several lines, can be highlighted in this way. Similarly, the 3D Phase Information also shows the access histograms, but in a per-phase fashion with each phase presenting the access behavior within a single program phase or an individual function. Hence, it contains a set of 2D histogram views. For the concrete diagram shown in Figure 1, each 2D view displays the miss distribution to the chosen data structure used in a single routine. This allows to detect concrete cache problems within a specific code region, which probably could not be exhibited by a global overview. By zooming into a single phase, the corresponding 2D diagram is shown in more detail, allowing a deeper examination of the problems in this phase. Hence, the 3D view enables a more comprehensive cache optimization.

Fig. 2. Sample view: Memory Operation Histogram

Figure 2 illustrates a sample Memory Operation Histogram created when executing the Fast Fourier Transformation (FFT) code on a system with a 4-way, 8k L1 cache and a 4-way, 128K L2 cache. This view contains three components for showing the mapping of main data structures in the cache, the runtime cache actions, and the memory location of the data structures respectively. The first component, as depicted in the left of Figure 2, gives the location of array blocks in the cache. For example, the first block of array X is mapped to the second set of the first level cache. The memory addresses of the arrays are presented in the right component. As the primary part of this view, the cache actions are shown in the middle. It depicts the changes of line states and contents in the user-specified cache sets. The status of a cache line can be replaced, hit, loaded, or empty, and is identified with colors. The visualization is synchronized with the actual cache operation, i.e. it shows the changes step by step according to the runtime data accesses. Users can also observe the complete operation series within a single set by clicking at the set area. Hence, this view helps to find overmapping between array blocks and to deduce optimization parameters, such as the padding stride, for eliminating these mapping conflicts.

4.4 Optimization: A Case Study

We use the LU code as an easy-to-follow example to depict how the visualization tool helps to improve the cache behavior. LU factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense matrix is divided into blocks to exploit temporal locality on submatrix elements. We executed this program and simulated the cache behavior with CacheIn. The first CachIn reports show high cache misses.

According to the visualizer's presentation, we found that these cache misses are mainly caused by accessing the matrix being decomposed. Besides this, the visualizer

shows that theses misses are caused by cache line replacement on the matrix itself. Based on the Memory Operation Histogram view, we then found that several matrix blocks of LU, which are needed for a single computation, are mapped into the same cache sets at the runtime leading to replacement. According to this detection, we optimized the LU code with a better data allocation to prohibit these overmappings. We executed LU and simulated the cache again but with the optimized code. An improvement with cache locality was achieved.

5 Conclusions

In this paper, we introduce a visualization tool for presenting the cache access behavior. In order to overcome the deficits of existing visualizers in supporting locality optimization, this cache visualizer is specially designed. Unlike the conventional visualization tools which show "what happened", our toolset shows the problem, the reason, and the solution, e.g. "where the bottleneck lies", "what results this bottleneck", and "how to correct it". For this, it deploys a variety of graphical views with different dimension, granularity, and data basis. A sample optimization based on this tool has proven its feasibility and effectiveness.

In the next step of this research work, we will use the visualization platform to further study the impact of cache locality optimizations with complicated, large applications. Then the optimization will be moved to realistic applications and real hardware systems. This step also brings potential extensions to the established cache visualizer.

References

1. C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. *ACM SIGPLAN Notices*, 34(5):229–241, May 1999.
2. C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, 10:21–40, 2000.
3. Intel Corporation. Intel VTune Performance Analyzer. available at http://www.cts.com.au/vt.html.
4. N. Megiddo and V. Sarkar. Optimal Weighted Loop Fusion for Parallel Programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, New York, June 1997.
5. J. Park, M. Penner, and V. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 32–33, April 2002.
6. G. Rivera and C. W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998.
7. J. Tao and W. Karl. CacheIn: A Toolset for Comprehensive Cache Inspection. In *Proceedings of ICCS 2005*, May 2005.

8. E. van der Deijl, G. Kanbier, O. Temam, and E. Granston. A cache visualization tool. *Computer*, 30(7):71–78, July 1997.
9. Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the Impact of the Cache on Program Execution. In *Proceedings of the fifth International Conference on Information Visualization*, pages 336–341, July 2001.