Supporting Cache Locality Optimization with a Toolset

Jie Tao and Wolfgang Karl

Institut für Technische Informatik
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{tao, karl}@ira.uka.de

Abstract. Cache performance significantly influences the computation power of modern processors. With the trend of microprocessor design for both general use and embedded systems towards chip-multiple, cache performance becomes more important because an off-chip access is rather expensive in comparison with on-chip references. This means cache locality optimization remains a hot research area for the next generation of computer architectures.
In this paper we present a tool environment aiming at providing the programmers sufficient support in the task of optimizing source codes for better runtime cache behavior. This environment contains a set of tools ranging from profiling, analysis, and simulation tools for gathering performance data, to visualization tools for graphical presentation and platforms for program development. Together, these tools establish a feedback loop for tuning cache performance on current and emerging uniprocessor and multiprocessor systems.

1 Introduction

Processor speed is growing at an exponential rate. In contrast, the increase of memory speed is rather lower. This results in an ever widening gap and the continuously growing memory wall. Uniprocessor chips usually rely on large cache size to overcome the problem, however, this solution is often not efficient due to the complexity in both application access pattern and the memory system. In addition, such solutions can not be commonly applied to chip-multiprocessor machines, while the increase in number of on-chip processors usually results in the decrease of per-processor caches.

For a deeper insight into the influence of cache memories we examine a simple, easy-to-understand example. Suppose a program generate 180 millions memory accesses during execution. If each memory access takes 50ns, 9 seconds are needed to access the memory. Would 20% of the data be found in cache, which is 20-fold faster than the main memory, the time for memory operation takes 7.29 seconds, forming an improvement of 19%. If the cache access behavior could be optimized and 80% of the data was acquired from the cache, the time for data accesses would only account for 4.95 seconds, which is a 56% improvement to the no-cache version.

This example depicts that cache performance can significantly affect the overall execution behavior of an application. It also demonstrates the necessity of code optimization with respect to cache performance. A prerequisite for this kind of optimization is the cache access behavior and the access pattern of applications. For this knowledge, programmers must rely on the help of tools, since static analysis of the source code usually show inefficiency and tedious work.

In this case, a visualization tool is needed to present such access pattern and the runtime behavior in user understandable manner. This tool, in turn, requires performance data, potentially not at high-level but in detail and covering various aspects.

Actually, modern processors offer a set of performance counters to collect specific events including those about caches. This helps to locate performance bottlenecks, however, does not suffice for a comprehensive understanding of the access pattern and behavior. For example, it does not present the reason of misses; it also say nothing about the affinity among data accesses. Therefore, due to this limit, most existing visualization tools are only capable of providing an environment for programmers to analyze the problem of the code and to detect the regions or data structures responsible for the poor performance. It is difficult or even impossible for users to detect an appropriate optimization scheme with their help.

In this case we implemented a set of tools with the goal of efficiently helping the users in locality optimization by presenting both the problem and hints for solutions. This toolset contains the following tools:

- A visualization tool that displays the performance data in user-understandable way. In contrast to existing tools, this cache visualizer not only shows the updates of cache contents but also, and with more endeavor, depicts the reason of cache misses enabling the detection of optimization strategies.
- A profiling tool that utilizes performance counters to collect information about individual events like cache hit/miss, total memory access, and access latency.
- A pattern analyzer that detects repeated address sequence and access stride. The former helps to allocate data with spatial locality in the same cache block, while the latter can be used to guide software prefetching.
- A cache simulator that models the runtime cache activities and analyzes the feature of cache misses.
- A program development environment that establishes a platform for application design.

All these tools work on both sequential and parallel applications, hence can be used to optimize the data locality on both uniprocessor and multiprocessor architectures.

In the following, Section 2 depicts the whole infrastructure of the toolset. This is followed by a detailed description of individual tools, first the visualization tool in Section 3, then the tools for data acquisition in Section 4, and lastly the development environment in Section 5. In Section 3 we also demonstrate

how to apply the visualized information to optimize applications towards better runtime cache performance. The paper concludes in Section 6 with some current and future directions.

2 Infrastructure of the Toolset

The infrastructure of the toolset is depicted in Figure 1. As shown on the most right side of the figure, the program development environment is the core of this infrastructure. From there execution commands can be issued and performance profiling, pattern analyzing, cache simulation, and visualization can be started. The profiling has to be done during the execution of an application because it relies on the performance counters to acquire performance data. A memory reference trace is also generated at the runtime. This trace records all memory accesses performed during the program run. The trace is used as input of both the pattern analyzer and the cache simulator which in turn produce access patterns and cache miss information individually. Performance data from the profiler and the cache simulator is displayed by the visualization tool, where the profiler contributes statistical numbers, while the simulator provides cache miss analysis and other detailed information about the cache behavior like the runtime cache operations. The visualization of the output from the Analyzer, however, is integrated in the development environment due to the combination with source codes.

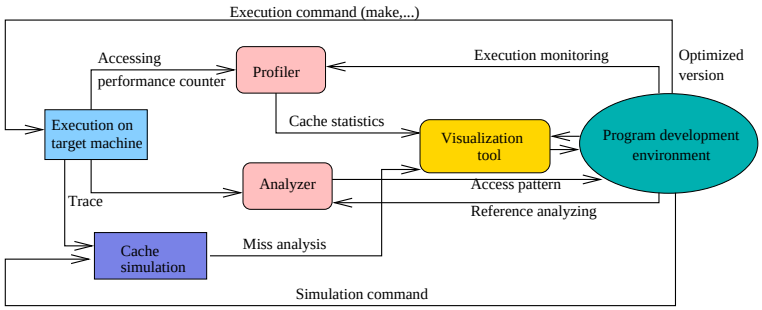

Fig. 1. Infrastructure of the toolset

Figure 1 also demonstrates the feedback loop for tuning cache performance. This loop contains four steps: running the program, gathering performance data, cache visualization, and optimization in source code. After the optimization, users can execute the optimized version of the program and examine the optimization result using the development environment. If cache problems still exist, another feedback loop may be started and further optimizations can be conducted.

3 Cache Visualization

Core of the tool infrastructure is a visualization tool, called YACO [6], which shows both bottlenecks and the reason for them. Following the common optimization process, YACO uses a top-down approach to direct the user step-by-step to detect the problem and the solution. First, users acquire an overview about the cache access behavior shown by the chosen program. Based on this overview, users can determine whether an optimization is essential. In the next step, the access hotspots can be located. After that, the reasons and interrelations between memory references can be detected using the presented information about runtime cache operations. This information also enables the user to select appropriate optimization scheme and related parameters.

Specific feature of YACO, in comparison with existing cache visualization tools like CACHEVIZ [11], lies in its ability of presenting the miss reason. This information can guide the user to deploy effective optimization schemes. Actually, there exist a number of strategies for optimizing the cache locality. However, individual scheme is usually effective to certain cache miss. For example, array padding can tackle conflict miss, but not capacity miss. Hence, in order to adequately apply theses techniques users have to know the reason of cache misses. For this, YACO depicts the runtime activities of both data structures and the caches enabling the detection of the cache miss reason.

Now we use a matrix multiplication code to demonstrate how YACO helps the programmers to perform data locality optimization. The code contains mainly the following loop for computation:

$$
\begin{array}{l}
for(i = 0; i < N; i\ ++) \\
\quad for(j = 0; j < N; j\ ++) \\
\quad\quad for(k = 0; k < N; k\ ++) \\
\quad\quad\quad a(i * N + j) = c[i * N + j] + a[i * N + k] * b[k * N + j];
\end{array}
$$

First, YACO's Performance Overview shows an L1 miss ratio of 54%, rendering an optimization as necessary. For optimization we first use the Variable Miss Overview to locate the optimization target. As illustrated in Figure 2, this view presents the miss behavior of all data structures in the program, where for each data structure four columns show the statistics on total misses and each miss category. Absolute numbers of misses are depicted at the left bottom.

It can be seen that all three matrices introduce cache misses, especially matrix b. It can be also seen that most misses with a and b are capacity miss, while with c mapping conflict is the main miss reason.

For a and b this view indicates an insufficient number of cache lines for holding all the required data within an iteration. Examining the code region above, it can be observed that each k loop calculates a single element of matrix c and for this it needs a whole row of a and a whole column of b. More importantly, the row of a is reused for computing the next element. The capacity miss with a means that these elements have to be evicted from the cache before being reused. This problem can be tackled with loop blocking, an efficient approach

Fig. 2. Variable Miss Overview

for reducing capacity miss [8]. With blocking we generate the following code:

$$
\begin{aligned}
&for(block = 0; block < N; block += N/2)\\
&\quad for(i = 0; i < N; i\ ++)\\
&\qquad for(j = 0; j < N; j\ ++)\\
&\qquad\quad for(k = block; k < block+N/2; k\ ++)\\
&\qquad\qquad c(i * N + j) = c[i * N + j] + a[i * N + k] * b[k * N + j];
\end{aligned}
$$

The difference with the original code lies in that the innermost loop (loop k) does not perform the whole work for generating an element of matrix c; rather it does only a half of the whole work. The additional loop with *block* guarantees the whole work to be covered.

This optimization introduces a significant performance gain, which can be observed in Figure 3, which illustrates two Performance Overview of YACO showing the overall cache performance of both unoptimized version and the version with blocking. Observing the second column of both diagrams it can be noted that the number of cache hits is significantly increased with the optimization. As a result, the cache hit ratio raises from 46% to 62%.

For matrix c, however, a further study on conflict is needed in order to decide how to optimize the code. For this, we examine the Cache Set view of YACO, which shows the content update in the cache.

As depicted in Figure 4, this view contains several horizontal blocks, in this case 2 blocks, which corresponds to the cache lines in a cache set. These blocks demonstrate the operations and content update in the specific set. The operations are presented in chronological order with the right followed by the left. Figure 4 shows the operations of initializing the last few elements of matrix b and the begin of calculating the first element of matrix c.

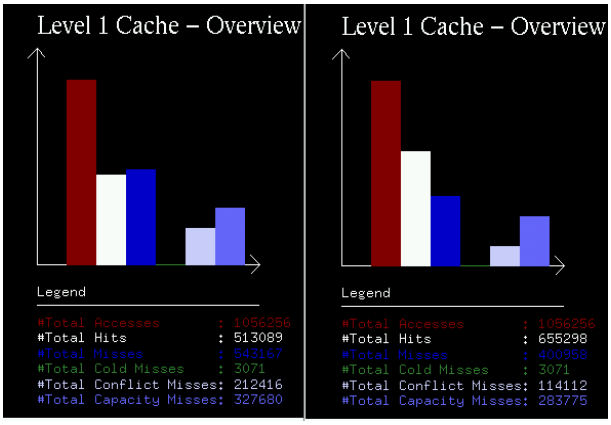

Fig. 3. Optimization with loop blocking

For each operation, the operation type, which can be a load, a replacement, or a hit, and the operation target, i.e. a variable or an element in an array, are presented. Examining line 1 it can be seen that the same element of matrix c, here $c\,0/0$ in the form of block/element, is reused but replaced by elements of matrix a every time after the access. Actually, this corresponds to the computing process where first a multiplication is performed and then the result is accumulated to the c element. The multiplication is performed on different elements of a and b, but the accumulation targets on the same element of c. Hence, better performance can be achieved if the elements of c can be kept in cache.

An efficient approach to tackle conflict miss is padding [5]. The idea of this approach is to additionally allocate a memory buffer to change the mapping behavior of data structures in the cache. Following this approach we add a pad of one cache line between matrix b and c like this:

$$a = (double*)malloc(sizeof(double) * N * N)$$
$$b = (double*)malloc(sizeof(double) * N * N)$$
$$d = (double*)malloc(sizeof(double) * 4)$$
$$c = (double*)malloc(sizeof(double) * N * N)$$

From the Cache Set views for this code version we see that with the padding the mapping of matrix c has been changed to set 1 rather than set 0. This introduces cache hits not only with c but also with a. As a result, this optimization shows a 40% of cache miss reduction which results in a raise of 23% in cache hit ratio.

All these performance gains have to be a contribution of the visualization tool that allows the user to detect the miss reason and further the optimization strategy.

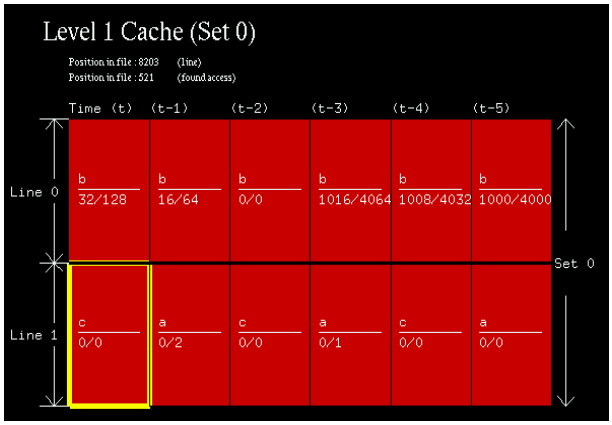

Fig. 4. The Cache Set view of set 0 in L1

4 Tools for Gathering Cache Performance Data

A prerequisite for any visualization is cache performance data. We currently rely on three tools to acquire this data with one extending another. Fist we deploy performance counters to collect information of global cache events. This information can be used to detect access bottlenecks. We then use a pattern analyzer to acquire access patterns, such as spatial relationship among references and access strides. This information shows the user how to accurately use certain optimization schemes, such as grouping and prefetching, to reduce cache misses. Finally, we deploy a cache simulator to deliver information about miss feature and runtime cache activities. This information enables the application of more efficient optimization schemes like code transformation and array padding.

Data Structure Profiling with Performance Counters. Performance counters are specially provided by modern architectures to gather performance information with limited overhead. This feature has to be utilized to first detect access hotspots where a large number of cache misses occur. For user-level optimization, however, the hotspots have to be related to code regions and data structures. This requires the counters to deliver access addresses together with the miss report. In this case, we developed a novel data profiler on Itanium 2 [4]. However, this profiler can be ported to other architectures with the requested counter feature.

Based on the *pfmon* kernel interface [3], we are able to control and access the performance counters, and generate a trace file for captured cache miss events. For each event, this file stores the address of the instruction issuing this access, the data address, and the latency for acquiring the missing data. For mapping the addresses to variables in the source code, we rely on the debugging information in the binary for static data structures and a self-made library for dynamic data structures. This library is used to instrument the *malloc* calls and collect the

mapping information. Based on this mapping information and also the source code, miss events in the trace file are ordered to individual data structures and as a result miss statistics on them are generated. These statistics can cover the whole problem, a single function, or even a code line, allowing hence to exactly locate the concrete access hotspots.

Pattern Analysis. The second tool for collecting performance data is a pattern analyzer [10] that provides address groups, access strides, and push back distances. The base of this tool is a memory reference trace, which is acquired with a code instrumentor. The instrumentor takes an assembly code as input, marks all memory references, inserts calls for acquiring thread IDs, and generates an extended version of the code. The execution of this version results in a creation of the reference trace.

Based on the trace, the analyzer applies appropriate algorithms to detect the regularity among the references and to give hints about optimization possibilities. For detecting address groups it applies Teiresias [7], an algorithm often used for pattern recognition in Bioinformatics, to find accesses which repeatedly occur together but target different memory locations. This helps to pack the target addresses of these accesses in the same data block so that the needed data can be loaded into the cache with the data for the first access, guaranteeing that the rest accesses are all cache hit. This kind of optimization is called grouping, another useful technique to enhance the cache performance. However, for deploying this scheme the knowledge about reference affinity is necessary. In comparison with common-used compiler-based approach [9], our analyzer is simple and based on runtime references.

For detecting access strides the analyzer uses an algorithm similar to that described in [2] to find the access regularity within a data array. The results are a set of records that describe each detected stride with three parameters: start address, access distance, and number of repeating. This information can be used for efficient prefetching, because it shows the address of the next required data.

Additionally, this analysis tool determines for each memory access whether it is a cache hit or a cache miss by computing the reuse distance and set reuse distance. For a cache miss, it also calculates the push back distance which shows how many steps a miss access must be shifted in order to achieve a hit.

The Cache Simulator. Simulation has been generally used to evaluate architecture design, to understand the behavior of applications and target machines, to deliver state information, and to optimize system behavior including the cache performance [1]. Similarly, we developed a cache simulator.

The cache simulator models the runtime cache operations and gathers information about cache activities and information exhibiting cache miss reason. It also uses the memory reference trace as input. For each record in the trace, it simulates the search process in a real multiprocessor cache hierarchy and stores the result in an operation sequence file.

For a cache miss, the simulator analyses the reason for it, i.e. whether it is a cold miss, a conflict miss, or a capacity miss. While cold miss can be simply identified by examining if an access is a first reference, identifying capacity and

conflict miss has to rely on reuse and set reuse distance, which need specially designed algorithms to compute efficiently.

Besides, the cache simulator provides specific functionality for multiprocessor systems. For example, it models a variety of cache coherence protocols and can detect false-sharing across processors.

5 Program Development Environment

In order to simplify the user's task in code optimization, we implemented a programming environment which builds a platform for code development and also for integrating all tools.

This environment is mainly composed of three components: window for code development, window for visualization of access patterns and for comparison of cache performance by different runs, and window for information like runtime output.

The first component offers a platform for modifying, compiling, analyzing, and executing an application. It uses a menu concept to enable the issuing of execution command or the starting of those tools for data collection and visualization. The second component provides a platform for displaying the access pattern, in this case the address group and access stride, the hit/miss behavior of each memory reference, and the execution comparison between transparent and optimized version of the same program. Address groups are presented using virtual addresses. However, by clicking an individual group the corresponding variables to the addresses in the group are immediately marked in the source code. Similarly, for the view showing the hit/miss behavior users can select any reference and in the source code all occurrence of the corresponding variable is marked and in case of a cache miss the position is highlighted, where the reference has to be issued in order to achieve a cache hit for it. The access stride, however, is directly presented within the arrays that are depicted using a diagram showing the array elements holding this stride. In addition, the hit/miss behavior of the access to the elements is also presented.

6 Conclusions

In this paper we introduce a set of tools developed for supporting users in the task of cache locality optimization. This toolset contains components for acquiring cache performance data, tools for behavior visualization, and platform for program development. Based on theses facilities, users can detect cache problem and optimization strategies, and further perform the optimization directly in the source code.

A limitation of this toolset is the workload size of examined applications because both the simulator and the pattern analyzer use a reference trace as input. This trace could be rather large for realistic applications. Currently we are working on different approaches to reduce the trace size. First, we use smaller workload size to detect optimization strategies and then deploy these strategies

to realistic applications. This means also a corresponding reduction of the cache size, which indicates additional burden to users. The second approach is to first use counter information to find critical functions and then only instrument these functions. However, this can still generate large trace file, while most applications show considerate accesses even within a function. Hence, our third approach is to generate a reduced trace which only stores the accesses in the first few iterations of all loops. All theses approaches have to be examined in detail with respect to accuracy and the trace size.

We also intend to investigate compiler-level optimization. This means the gathered cache information is delivered to compilers which based on this information transparently perform the optimization during the generation of executables. This will introduce better performance than approaches using heuristic analysis, because we provide the runtime access pattern for compilers to make optimization decisions.

References

1. L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Supercomputing'02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, 2002.
2. T. Mohan et. al. Identifying and Exploiting Spatial Regularity in Data Memory References. In *Supercomputing 2003*, Nov. 2003.
3. HP. Perfmon Project Web Site. available at http://www.hpl.hp.com/research/linux/perfmon/.
4. Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*, volume 1–3. 2002. available at http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm.
5. K. Ishizaka, M. Obata, and H. Kasahara. Cache Optimization for Coarse Grain Task Parallel Processing Using Inter-Array Padding. In *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003*, volume 2958 of *Lecture Notes in Computer Science*, pages 64–76, 2003.
6. B. Quaing, J. Tao, and W. Karl. YACO: A User Conducted Visualization Tool for Supporting Cache Optimization. In *High Performance Computing and Communcations: First International Conference, HPCC 2005. Proceedings*, volume 3726 of Lecture Notes in Computer Science, pages 694–703, Sorrento, Italy, September 2005.
7. I. Rigoutsos and A. Floratos. Combinatorial Pattern Discovery in Biological Sequences: the TEIRESIAS Algorithm. *Bioinformatics*, 14(1):55–67, January 1998.
8. G. Rivera and C. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of Supercomputing'2000*, 2000.
9. X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight Reference Affinity Analysis. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 131–140, New York, NY, USA, 2005.
10. J. Tao, S. Schloissnig, and W. Karl. Analysis of the Spatial and Temporal Locality in Data Accesses. In *Proceedings of ICCS 2006*, number 3992 in Lecture Notes in Computer Science, pages 502–509, May 2006.
11. Y. Yu, K. Beyls, and E.H. D'Hollander. Visualizing the Impact of the Cache on Program Execution. In *Proceedings of the 5th International Conference on Information Visualization (IV'01)*, pages 336–341, July 2001.