CacheIn: A Toolset for Comprehensive Cache Inspection

Jie Tao and Wolfgang Karl

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

Abstract. Programmers usually rely on cache performance data to optimize their applications towards high runtime cache hit ratio. In this paper, we introduce a software toolset CacheIn, which uses simulation and monitoring to collect comprehensive cache performance data. CacheIn consists of a cache simulator for modeling the cache activities, a cache monitor for gathering different kind of information, and a multilayer software infrastructure for processing the raw monitoring data towards statistical, high-level representations, like histograms and summarized numbers. CacheIn exhibits both the details of traditional software mechanisms and the feasibility of performance counters. Based on a code instrumentor, we have verified CacheIn using standard benchmarks. Initial experimental results show its full functionality in terms of providing accurate, comprehensive, and corse-grained performance data.

1 Introduction

Cache locality optimization is regarded as a critical issue for achieving high performance. A prerequisite for such optimization is performance data that shows the cache access behavior of applications. Currently, computer systems rely on either software profiling or hardware counters to acquire this information . However, both approaches can provide only limited performance data; and information about important performance metrics, like false sharing, cache line invalidation, cache line replacement, and access pattern, is missing.

We developed CacheIn for acquiring sufficient information that allows comprehensive analysis and optimization. For feasibility, we use a cache simulator to model various cache organizations and different caching policies. This allows to apply CacheIn to study caches on different target machines. For comprehensive data, we implemented a software model of a cache monitor capable of observing the memory traffic on all levels of the memory hierarchy and collecting detailed information about the cache access behavior. This monitoring facility can be configured to a variety of working modes at the runtime and provide different information needed for understanding the various access pattern of applications and for the selection of appropriate optimization techniques. Additionally, in order to avoid delivering fine-grained, low-level performance data, a multilayer software infrastructure has been developed for transforming the original monitoring information into a high level abstraction with respect to data structures. This includes both APIs for address transformation and interfaces for convenient

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 174–181, 2005.

access of the performance information. Based on the software infrastructure, CacheIn provides, for example:

- Access histograms on individual location or the whole memory hierarchy. They record the access distribution to the complete working set at granularity of cache lines. This gives the user a global overview of the memory accesses allowing an easy detection of access hotspots.
- Statistics on single events, like cache misses and total references to a specific memory region or performed inside an individual iteration, loop, or function; number of cache line replacements within an array; and number of first references with respect to memory regions, arrays, or code regions. This allows to find the sources causing cache miss and inefficiency.
- Profile of access addresses. Based on this information, accesses at close intervals can be grouped into the same cache line, thus reducing cache misses caused by first references and also increasing spatial reuse of the cached data.
- Sequence of cache events which records the frequency of replacements and numbers of hits between a first reference and a replacement performed on the same memory line. This helps to hold frequently reused data in the cache prohibiting frequently replacement and reloading.
- Additional information for multiprocessor systems, e.g. information about false sharing. This helps to reduce the number of cache line invalidations and thereby improve the cache efficiency.

CacheIn requires memory references as input for its cache simulator. This can be e.g. extracted from debugging information, or acquired from profiling tools and compilers. For this, CacheIn provides a simple interface to theses systems. For example, to verify CacheIn's functionality, we have applied a code instrumentor to provide memory references.

The remainder of this paper is organized as following. Section 2 introduces several related works in acquiring cache performance data. This is followed by a detailed description of this toolset in Section 3, including the cache simulator, the cache monitor, and the software infrastructure. In Section 4 some initial experimental results are illustrated. The paper concludes in Section 5 with a short summary.

2 Related Work

As locality tuning requires information about memory accesses, various approaches have been developed for collecting performance data with respect to the memory system. These approaches can be roughly divided into two categories: hardware supported and simulation/profiling based.

On the area of hardware, modern processors provide performance counters for recording important information about the runtime execution. The UltraSPARC IIi Architecture [6] provides two registers to count up to 20 events like cache misses, cache stall cycles, floating-point operations, branch mispredictions, and CPU cycles. The IBM POWER2 [7] has 5 performance counters enabling the concurrent monitoring of five events, such as the number of executed instructions, elapsed cycles, and utilization

of the various execution elements. Intel's Pentium4 [2] supplies even 18 performance counters to allow the collection of more information, like that about specific instructions and pipeline conflict.

For simulation/profiling based schemes, well-known examples are SIP [1], MemSpy [4], and Cachegrind [9]. SIP (Source Interdependence Profiler) is a profiling tool that provides information about cache usage and cache miss ratios. It uses SimICS [3], a full-system simulator, to run the applications for collecting cache behavior data. MemSpy is a performance monitoring tool designed for helping programmers to discern memory bottlenecks. It uses cache simulation to gather detailed memory statistics and then shows frequency and reason of cache misses. Cachegrind is a cache-miss profiler that performs cache simulations and records cache performance metrics including L1 instruction cache reads and misses, L1 data cache accesses and misses, and L2 unified cache accesses and misses. The input for Cachegrind is extracted from the debugging information.

Overall, the hardware counters provide only limited performance data and details are missing; programmers often have to do hard work in analyzing the applications in order to understand the code and the data structure. For simulation/profiling based approaches, performance data provided by existing implementations is also restricted to numbers of specific events and this information does not suffice for a full understanding of the cache access behavior. We therefore design a toolset in order to provide the programmers with detailed, understandable, and easy-to-use performance data which is necessary for efficient code and data optimization.

3 The Approach

The goal of this work is to deliver comprehensive cache performance data that not only shows the various aspects of cache behavior but also is easy to use, e.g. in the form of statistics and at high-level in terms of data structures. We achieve these features with a software framework consisting of a flexible cache simulator, a generic cache monitor, and a multilayer software infrastructure. A high-level overview of this framework is depicted in Figure 1.

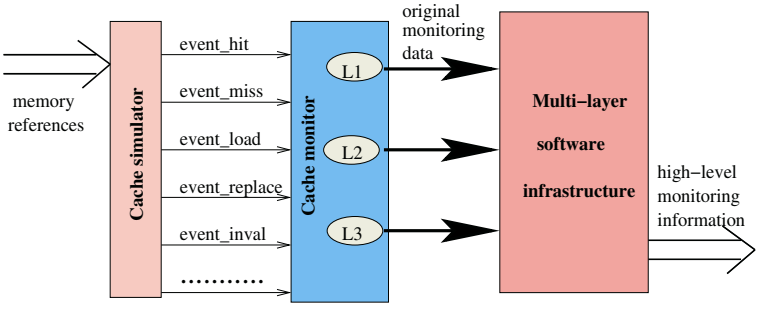

Fig. 1. CacheIn software infrastructure on the top level

3.1 Cache Simulation

We implemented a flexible cache simulator to model the operations within the caches. Its input are memory references which can be provided by debuggers, simulators, or compilers. Based on the access address and transaction type, it simulates the process of cache searching. In comparison with the similar work in this area, e.g. the cache simulator in Cachegrind [9], our simulator distinguishes in its flexibility, feasibility, and comprehensiveness.

First, it models a multilevel cache hierarchy and allows each cache to be organized as either write-through or write-back depending on the target architectures. All relevant parameters including cache size, cache line size, and associativity (from directly mapped to fully associative) can be specified by the user.

In addition to simulating the caches themselves, the cache simulator models a set of cache coherence protocols. This includes the hardware-based MESI protocol, several relaxed consistency schemes, and an optimal, false-sharing free model using a perfect oracle to predict required sharing. This property of the cache simulator can be used to understand the invalidation behavior and further to develop adaptive mechanisms for optimization.

The output of the cache simulator are various events that can be directly delivered to the cache monitor. As shown in Figure 1, the cache simulator generates a set of different type of events, including hit, miss, load, replacement, and invalidation. Besides the type, an event also holds parameters, such as access address, cache level, and transaction type, which are needed for the cache monitor to achieve its functionality.

3.2 Cache Monitoring

To acquire performance data capable of reflecting the various aspects of cache accesses, a cache monitor is needed. For flexibility, modularity, and unified interfaces and construct, several specific requirements must be considered in the monitor design: 1) each cache level needs an individual monitor and all monitors have the same structure; 2) monitors can be configured to a variety of working modes depending on the user demand; 3) a monitor must have the capability of event filtering, aggregating, and preprocessing for avoiding to deliver information of unessential details.

Following these requirements, we designed a novel monitoring concept and implemented a generic, multi-functional monitoring component which can be combined to any location of the cache hierarchy. This cache monitor is comprised of an input interface, an output interface, and a flexible analysis module. The former is an interface to the cache simulator, while the output interface allows the data to be delivered to the software infrastructure for further processing.

As the primary component of the cache monitor and also one of the most distinguished part of this work, the analysis module is responsible for event handling. It has mechanisms for bypassing events of no user interest. It also provides a dynamic granularity control that allows to monitor single words for e.g. detecting false sharing and that enables the aggregation of neighboring events in case of access histograms where fine granularity is not required. For this granularity control it deploys a counter array to transiently store the processed events, and a ring buffer to hold the later generated monitoring data.

Another feature of the analysis module is its various functions. It can be specified to work in a set of modes, either triggering user-defined events and counting their occurrence or working in a histogram-driven mode where it captures all related events for generating a certain type of histograms, like memory access histogram recording all accesses to the complete working set, replacement histogram that stores each cache line replacement in chronological order, or histogram of access addresses providing a series of access targets.

3.3 Software Infrastructure

The monitoring data delivered by the cache monitor is still event-based, fine-grained, and hence not suitable for directly providing to users or performance tools that usually need high-level representations in the form of access histograms and summarized access numbers. For this, a software infrastructure has been developed.

As illustrated in Figure 2, the software infrastructure contains several layers, each processing a step further of the monitoring information. As mentioned, the cache monitor stores the monitoring data in a ring buffer. From there, data is sorted by the Control Component and stored into a histogram chain that is ordered by the addresses of corresponding memory blocks with cache line size, so called memory lines.

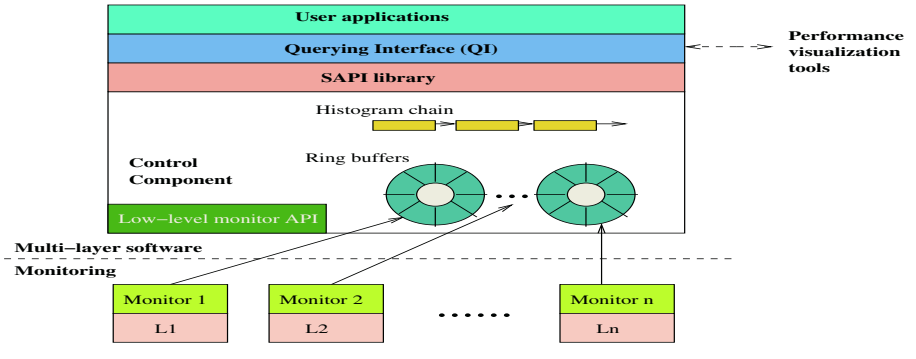

Fig. 2. Software infrastructure for data processing

In addition, the Control Component also combines the monitoring data from different monitors and probably different processors (on a multiprocessor system). On top of this component, the SAPI (Standardized API) library further processes the monitoring data into statistical forms such as total number of single events and access histograms. Finally, the Querying Interface (QI) maps the virtual addresses into data structures using the context table provided by some compilers or using the debugging information. Also from this component, the final high-level data abstraction is delivered to performance tools and applications for visualization or performance analysis.

As the primary data processing component, SAPI provides a set of functions for generating both statistical numbers on individual events and access histograms recording the occurrence of single events over the complete working set. In addition, SAPI provides functions for analyzing the access addresses and invalidation operations. These

functions can be used to provide address groups and to create invalidation sequences, which are needed in address grouping and false sharing detection.

4 Verification Based on a Code Instrumentor

In order to verify the functionality of CacheIn, we have deployed Doctor [5], a code augmenter, to deliver memory references. Doctor is originally developed as a part of Augmint [5], a multiprocessor simulation toolkit for Intel x86 architectures. Doctor is specially developed for augmenting the assembly code with instrumentation instructions that generate memory access events. For every memory reference, Doctor inserts code to pass the accessed address and its size to the simulation subsystem of Augmint. For this work, we have slightly modified Doctor in order to build an interface to the cache simulator of CacheIn.

Actually, CacheIn is capable of supplying a variety of performance data, like statistics on single events, individual access histogram on a single cache level, combined histogram on the whole memory hierarchy, and operation histogram showing ordered events. We choose two examples to show its basic function: complete access histogram and statistics on false sharing.

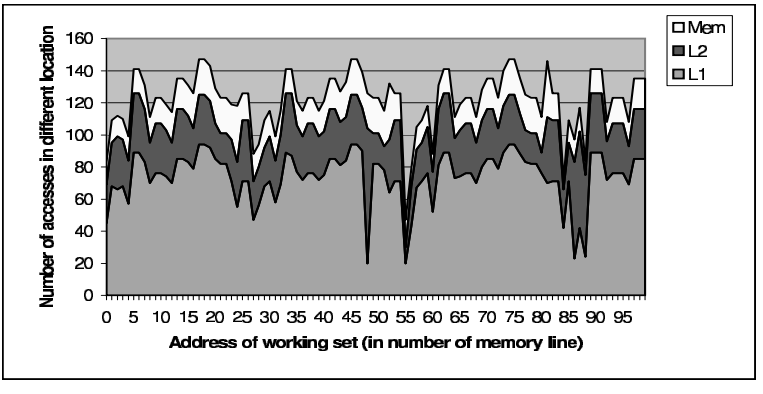

Fig. 3. Combined memory access histogram of WATER

A complete access histogram consists combined data for all monitors on a cache hierarchy. Figure 3 illustrates a sample histogram with WATER, an application for evaluating water molecule systems. For this, we simulated the cache behavior of WATER on a system with two caches. For the concrete diagram in Figure 3, the x-axis presents the first 100 memory lines of the complete working set and the y-axis presents the number of accesses performed on each memory line. These accesses can be either L1 hits, L2 hits, or have to be performed in the main memory. As can be seen, most memory references can be found in the caches, however, for this concrete example there also exists memory regions with a high access rate of the main memory. Hence, the access histogram enables a direct observation of distinct memory access behavior within a

single code. This allows to find access hotspots, forming the first step towards cache optimization.

Our second example addresses multiprocessor systems with shared memory. On such a system, cache line invalidation is an additional critical issue causing cache misses. Such invalidations, however, could be unnecessary. A specific case is false sharing, where a cache line on a processor is invalidated because another processor has modified a word of the same data copy; but the processor needs a different word within the cache block. In this case, it is not necessary to invalidate the cache line.

For supporting the optimization with respect to false sharing, the cache monitor of CacheIn provides event profile that records the histogram of memory operations in a serial order. This allows its API to compare the target of a shared write with all following shared reads thus to detect false sharing. Table 1 shows the statistics reported by the cache monitor API. This result was acquired by simulating several applications on a 32 processor system, including a Fast Fourier Transformation (FFT) code, an LU-decomposition code for dense matrices (LU), an integer radix sort (RADIX), the OCEAN code for simulation of large scale ocean movements, and the WATER code. All these applications are chosen from the SPLASH-II Benchmarks suite [8].

We have measured the number of false sharing for four different cache coherence protocols. MESI is a common used scheme for hardware-based shared memory machines. This scheme performs cache line invalidation by each write operation to shared data. FULL is a kind of release consistency model usually deployed on systems with distributed shared memory. This scheme performs whole cache invalidation at each synchronization event like lock and barrier. OPT is an optimal scheme that invalidates a cache line by a read operation and only when the accessed cache line has been modified. SCOPE is an optimized version of FULL, where only the cache lines holding remote data are invalidated rather than the complete cache. In principle, OPT should performs better than MESI, MESI better than FULL, and SCOPE better than FULL.

Table 1 depicts the absolute number of total invalidations and false sharing of them. It can be observed that applications vary in this behavior. For FFT, 40% of the invalidations with MESI are false sharing, 50% with OPT, and even 78% with SCOPE. For the FULL scheme, it is senseless to compute this proportion because most invalidations are performed on invalid cache lines. However, more false sharing can be observed with this protocol. LU performs better than FFT with a slight percentage of false sharing, e.g. 3% with MESI. RADIX reports the highest false sharing (65% with MESI) and the other two applications perform better than RADIX and FFT.

Table 1. Number of false sharing with different cache coherence schemes

	MESI		FULL		OPT		SCOPE	
	invalid.	false share.	invalid.	false share.	invalid.	false share.	invalid.	false share.
FFT	2670	1057	18517	2626	992	496	2110	1647
LU	16185	487	304137	4728	3392	24	106380	3886
RADIX	7274	4716	63857	5488	654	160	15937	4774
OCEAN	17990	2875	253808	4405	8411	2538	133490	2748
WATER	9302	2704	70662	4486	2920	2557	20590	2378

Overall, the experimental results indicate that for most applications optimization with false sharing is necessary in order to alleviate cache misses. For this, CacheIn provides required data for analysis.

5 Conclusions

This paper introduces a software framework developed for acquiring comprehensive, accurate, and detailed performance data about the cache access behavior. CacheIn is comprised of a cache simulator for modeling the cache activity, a cache monitor for gathering original performance data, and a multilayer software infrastructure for creating statistical monitoring information in the form of histograms and summarized number. The specific feature of CacheIn lies in its feasibility, flexibility, and the capability of providing comprehensive performance data with various details. This has been proven by the verification based on a code augmenter.

References

1. E. Berg and E. Hagersten. SIP: Performance Tuning through Source Code Interdependence. In *Proceedings of the 8th International Euro-Par Conference*, pages 177–186, August 2002.
2. Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 1–3. 2004. available at Intel's developer website.
3. P. S. Magnusson and B. Werner. Efficient Memory Simulation in SimICS. In *Proceedings of the 8th Annual Simulation Symposium*, Phoenix, Arizona, USA, April 1995.
4. M. Martonosi, A. Gupta, and T. Anderson. Tuning Memory Performance of Sequential and Parallel Programs. *Computer*, 28(4):32–40, April 1995.
5. A-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, October 1996.
6. Sun Microsystems. *UltraSPARC IIi User's Manual*. October 1997. available at http://www.sun.com/processors/documentation.html.
7. E. Welbon and et al. The POWER2 Performance Monitor. *IBM Journal of Research and Development*, 38(5), 1994.
8. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.
9. WWW. Cachegrind: a cache-miss profiler. available at http://developer.kde.org/šewardj/ docs-2.2.0/cg_main.html#cg-top.