A Profiling Tool for Detecting Cache-critical Data Structures

Jie Tao[1,2], Tobias Gaugler[3], and Wolfgang Karl[3]

[1] Department of Computer Science and Technology
Jilin University, P.R.China
[2] Institut für wissenschaftliches Rechnen
Forschungszentrum Karlsruhe GmbH, Germany
[3] Institut für Technische Informatik
Universität Karlsruhe (TH), Germany
E-mail: jie.tao@iwr.fzk.de

Abstract. A poor cache behavior can significantly prohibit achieving high speedup and scalability of parallel applications. This means optimizing a program with respect to cache locality can potentially introduce considerable performance gain. As a consequence, programmers usually perform cache locality optimization for acquiring the expected performance of their applications.
Within this work, we developed a data profiling tool *dprof* with the goal of supporting the users in this task by allowing them to detect the optimization targets in their programs. In contrast to similar tools which mostly focus on code regions, we address data structures because they are the direct objects that programmers have to work with. Based on the Performance Monitoring Unit (PMU) provided by modern processors, *dprof* is capable of finding cache-critical variables, arrays, or even a segment of an array. It can also locate theses access hotspots to the most concrete position such as individual functions and code lines. This feature allows the user to apply *dprof* for efficient cache optimization.

1 Motivation

Amdahl's law shows the speedup and scalability of a parallel program is influenced by the parallelism: if no sufficient parallel tasks exist, an additional processor can not introduce a performance gain. Besides the parallelism, however, overhead of thread/process management and the memory behavior are also critical factors that decide the parallel performance [4]. The latter indicates both the cache behavior and the performance of the main memory in case of shared memory programs on multiprocessor machines with distributed memories. As the cache performance represents a common case, increasing programmers try to optimize their applications with respect to the cache efficiency.

With the growth of processor speed at exponential rate, the gap between memory and processor speed is continuously widening. Actually, cache is introduced into the computer system for bridging this gap. However, general architectures are not tailed to applications; hence most programs, both sequential and

parallel, show a significant number of cache misses when initially executed on a specific architecture. This scenario could be more critical for chip-multiprocessor (CMP) systems because on this kind of machines per-processor cache is smaller in comparison to the caches on uniprocessor or SMP machines. According to an existing research work [3], up to a 4-fold of cache miss rate has been measured on a 4-processor CMP as on a uniprocessor system.

The first challenge for cache locality optimization is to know the access bottlenecks, i.e. code regions or data structures which cause the most cache misses. To acquire this knowledge, cache miss events must be traced at the runtime during the execution of a program.

Actually, modern microprocessor architectures provide a hardware Performance Monitoring Unit (PMU) for collecting information about specific runtime events. However, theses PMUs only give global statistics on individual events, for example, the number of total misses at each cache location. The information in this form does not suffice for detecting bottlenecks.

In this case, we developed the profiling tool *dprof* in order to achieve high-level, user-understandable information capable of showing the optimization objects. *Dprof* is based on specific PMUs that are capable of delivering addresses of both the instruction and the data associated with a cache miss, like the PMU of the Intel Itanium 2 processor [8]. Using this special ability of such PMUs, it first generates a cache miss trace that records the captured runtime cache misses with address information included. In the next step, the virtual addresses in the trace file are mapped to code lines and data structures in the source program. For this, we rely on both the debugging information in the binary code for static data structures and a self-made library for dynamic ones. Based on this mapping information as well as the source code, miss events in the trace file are assigned to individual data structures and miss statistics on them are generated. This statistical information can be given at different granularity, covering e.g the whole problem, a single function, or a code line.

Additionally, *dprof* produces a histogram file, which records the number of cache misses with each individual data block of the whole working set of a program. This histogram is specifically formed in the way that it can be delivered to an existing visualization tool [12] to highlight the hot variables. These access hotspots in the form of graphical representations give the user a straightforward overview of the data structures with cache problems.

The first version of this profiler is implemented on Itanium 2, but can be ported to other processors, such as the Pentium 4, which also provide address information.

The rest of the paper is organized as follows. Section 2 describes the profiling tool in detail, including the data acquisition and the address mapping. This is followed by initial experimental results with several OpenMP applications in Section 3. Section 4 introduces several related work. The paper concludes in Section 5 with a short summary and some future directions.

2 Data Structure Profiling

Dprof aims at giving programmers possible support in their task of cache locality optimization, in the base of restricted information provided by available hardware resources. A large part of the development work is done with acquiring the needed knowledge for detecting hot data structures without involving the programmers. In this section, we first give a global overview of this profiling tool and then describe the implementation details.

***Dprof* Infrastructure**. Figure 1 illustrates our global concept of data profiling with performance counters. As shown on the right side of this figure, the performance registers are accessed during the execution of a program and the captured cache misses are recorded in a trace file. The access to performance counters is done through *pfmon* [6], an interface, similar to the PAPI [1] library, for establishing a connection between user-level software and the kernel-level hardware counters. Also at the running time of the program, as shown on the left side of the figure, dynamic data structures are instrumented with *libmalloc_hook*, a self-developed library. Static data structures, as shown in the middle of the figure, are achieved from the debugging information in the binary code of the program.

In the next step, the captured misses are processed, and associated to the data structures and code lines, according to the mapping information. Program source code is also required in this step, because it contains information for excluding those cache misses caused by other processes, e.g. a system process. The achieved result can be either in a text form for directly delivering to the users or in the form required by our visualization tool for graphical presentation.

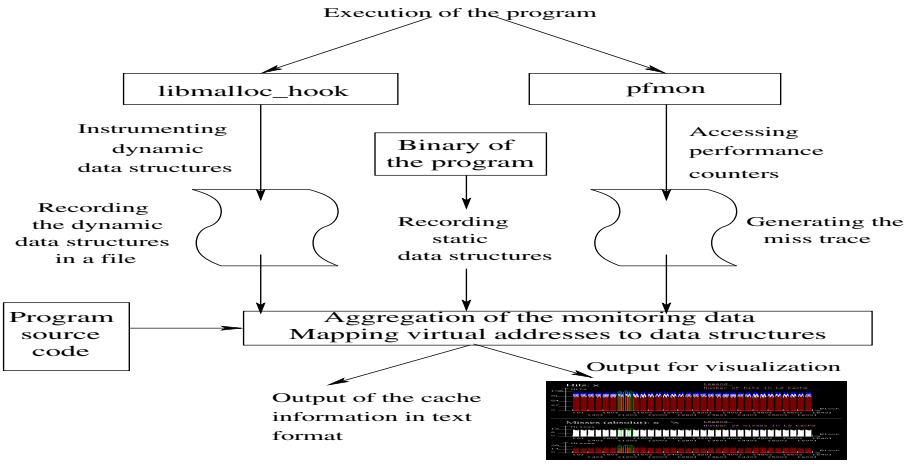

Fig. 1. *Dprof* overall infrastructure

Accessing the Performance Counters of Itanium 2. The Itanium 2 PMU has four 48-bits counters which enable the concurrent monitoring of up to four micro-architectural events. Among more than one hundred possible events, several can be used to measure cache misses and total references at each cache level. More specifically, the Itanium architectures provide specific registers, i.e. Event Address Register (EAR), which can be programmed to capture both the event and the instruction which caused the event.

When the data EAR is programmed to trace cache misses, the processor samples load misses and delivers for each sampled miss the data address, the instruction address, and the latency for accessing the data. The sampling follows an approach of interrupt, where a register counts cache misses and an interrupt is caused when the counter overflows. By giving the counter an initial value the sampling rate can be controlled.

The sampling rate is a tradeoff between the accuracy of gathered cache performance data and the intrusion influence on the program execution. A sampling rate of 1 enables to capture all cache misses, however, introduces high overhead. For this work a high accuracy is not necessary because we aim at finding access hotspots rather than measuring the exact number of cache misses. Hence, in order to reduce both the overhead and the trace size, we use an appropriate lower sampling frequency that can still present the correct hot data structures even only resulting in a lossy trace. This frequency is chosen based on the experiments with several applications. In the following section we demonstrate this result.

Overhead. *Dprof* introduces overhead in two scenarios: instrumenting *mallocs* and reading performance counters. According to our experimental results, the former is in most cases lower than 1% of the overall execution time. The overhead for accessing performance counters depends on the sampling rate. In our case of a low frequency, this penalty is under 4% of the total running time of the program.

Mapping Cache Misses to the Program Source. The addresses in the miss trace have to be mapped to the source code because they are not visible to programmers. As mentioned, the mapping information of static data structures can be extracted from the debugging information in the binary code. For dynamic data structures, however, the allocation is first performed after the execution of the program. Their mapping information can hence only be obtained at the runtime.

For this, we implemented a library that captures the *malloc()* calls and acquires thereby the start address and size of the allocated memory space. In addition, the instruction address of a *malloc()* call is also gathered. This information enables to order each memory allocation to the cache code line where a dynamic data structure is declared.

Based on the mapping information between virtual addresses and data structures, we could associate each captured cache miss to the responsible variable in the program. The instruction address in the miss record allows us further to locate the code line as well as the function, where the miss occurs.

Dprof is capable of delivering access hotspots at different granularities:

1. Data structure overview: statistics on the whole data structures within a program. Information includes number of misses at each cache level and the access latency for the misses.

2. Data structure per function. Miss statistics on data structures are individually calculated for each function in the program. In addition, time for both executing the function and accessing the data is provided and separately computed. This information shows how critical the penalty of memory operations is and this knowledge allows the user to determine whether the cache optimization is essential.

3. Data structure per code line. Number of misses of a data structure is individually calculated for each code line. This allows the programmers to locate cache problems to the most concrete position in the program source.

4. Data block: the finest granularity of miss statistics. In this case, a data structure is divided into blocks of cache line size and for each block the number of misses is computed. This information enables a separate processing of data segments, which is helpful when the optimization on the whole data structure is not efficient.

3 Initial Experimental Results

We applied *dprof* to examine several applications from the NAS OpenMP benchmark suite [2, 10]. The measurement was conducted on a 4-way Itanium 2 machine with an L1 data cache of 16KB, an L2 cache of 256KB, and an L3 cache of 3MB. All applications were executed using four threads.

Accuracy and Overhead of Sampling. The first experiment aims at studying the impact of sampling rates. For this, we tested all applications using different sampling rates. As similar results have been acquired, we only show the MG code as an example.

As depicted in Table 1, we measured the sampling overhead and the number of cache misses with three main data structures in the code. In addition, we computed the miss distribution, which presents the percentage of the misses with a single data structure to the total miss. This metric highlights the variable with poor cache performance.

In Table 1, a sampling rate of 1/10 means that every 10th miss is recorded and correspondingly a rate of 1/100 enables to capture every 100th miss event. First, it can be seen that the absolute number of cache misses varies significantly over different sampling frequency, and a higher sampling rate results in the collection of more cache misses but also introduces higher overhead. Nevertheless, the miss distribution does not alter a lot over the sampling rates. Even with the lowest rate of 1/10000 we can clearly see that variable u and r are access bottlenecks. For the following experiments we use the middle rate of 1/1000.

Access Hotspots. *Dprof* gives the user a clear view of the access bottlenecks, from global to concrete. It first shows which data structure in the whole program has cache problems (a global overview), then the function where most cache

Table 1. Number of cache misses with different sampling frequency

		Sampling rate				
		1/10	1/100	1/1000	1/10000	1/100000
Overhead (%)		29.17	9.83	3.29	0.83	0.43
Number of cache misses	$u[l][k][j]$	116540	101629	39045	3950	390
	$v[k][j]$	21682	15686	5529	574	49
	$r[l][k][j]$	85841	72824	22647	2347	233
Miss distribution	$u[l][k][j]$	0.52	0.53	0.58	0.57	0.58
	$v[k][j]$	0.09	0.08	0.08	0.08	0.07
	$r[l][k][j]$	0.38	0.38	0.34	0.34	0.35

misses with this data structure occur, and finally the concrete code line. Again we use the MG program to demonstrate the experimental results.

Table 2 demonstrates the global overview of cache misses with the MG application. In this table, the first two columns give the source files and line number where the data structures are accessed and declared. This is followed by the name of the observed data structure. Misses at different cache level as well as the number of total miss are depicted in the last four columns. Note that the concrete values in this table could be different with the results in the previous table because with the sampling technique, each run gives a different result even using the same sampling rate.

Table 2. Miss overview of main data structures

srcFile	decla.	data structure	L1Miss	L2Miss	L3Miss	total miss
mg.c	206	$u[l][k]$	6	0	3	9
mg.c	208	$u[l][k][j]$	32247	442	586	33275
mg.c	214	$v[k]$	1	0	1	2
mg.c	216	$v[k][j]$	3039	0	191	3230
mg.c	223	$r[l][k]$	2	1	1	4
mg.c	225	$r[l][k][j]$	17028	218	209	17455

The MG program clearly shows that variable $u[l][k][j]$ and $r[l][k][j]$ are the access bottlenecks. The former introduces more than the half of cache misses, while the later causes about 37% of the total cache miss.

Now we further examine the miss behavior of variable $u[l][k][j]$. Table 3 presents all functions containing this variable. For each cache level we measured both the number of misses and the latency for accessing the missing data. Here, the miss statistics does not cover the total misses at a cache level, rather it is the misses found in the next level cache. The values in the last two columns are the sum of misses and latency of all caches. Hence, the total misses in the table

Table 3. Functions containing the hot data structure

Function	L1 cache		L2 cache		L3 cache		total	
	#miss	latency	#miss	latency	#miss	latency	#miss	latency
interp	2703	13515	16	224	108	28291	2827	42030
comm3	19	95	0	0	8	2035	27	2130
resid	21804	109020	246	3445	253	65940	22303	178405
psinv	1176	5880	0	0	81	21280	1257	27160

are actually total L1 misses and the total latency is the time (in CPU cycles) for accessing the data missing in L1.

It can be clearly seen that *resid* is the hot function where optimization has to be performed. This function results in totally 22303 L1 misses, whereby 21804 ones can be tackled with L2 and another 246 with L3. However, still 65940 CPU cycles are lost for the data in the main memory which, according to our experimental results, is more than 20% of the time used to run this function.

Table 4. Hot data structures in a single function

Code line	L1 cache		L2 cache		L3 cache		total	
	#miss	latency	#miss	latency	#miss	latency	#miss	latency
531	2294	11470	113	1582	40	10480	2447	23532
532	2441	12205	0	0	0	0	2441	12205
533	2326	11630	131	1835	47	12009	2504	25474
534	2438	12190	1	14	166	43451	2605	55655
538	2454	12270	1	14	0	0	2455	12284

For optimization, *dprof* further gives the code line where the programmer has to do some transformations. As shown in Table 4, line 534 must be optimized because it shows a high memory access latency. Other lines has similar total misses, but most of the missing data can be found in other caches without accessing the main memory. Hence, the cache problem with them is not as critical as that with line 534.

Visualization of Bottlenecks. The profiling results can also be visualized using YACO [12], a visualization tool for displaying the runtime cache access behavior. Figure 2 and 3 are two sample views with the CG application.

Figure 2 is the Variable Miss Overview which shows the global access hotspots. Within this view, main data structures of a program are horizontally listed and separated with a bidirectional arrow. Corresponding to each data structure, the numbers of cache misses with all parallel threads at a selected cache level are illustrated in bars. For this example, four colored bars represent the four threads running the application. It can be seen that CG has an identical

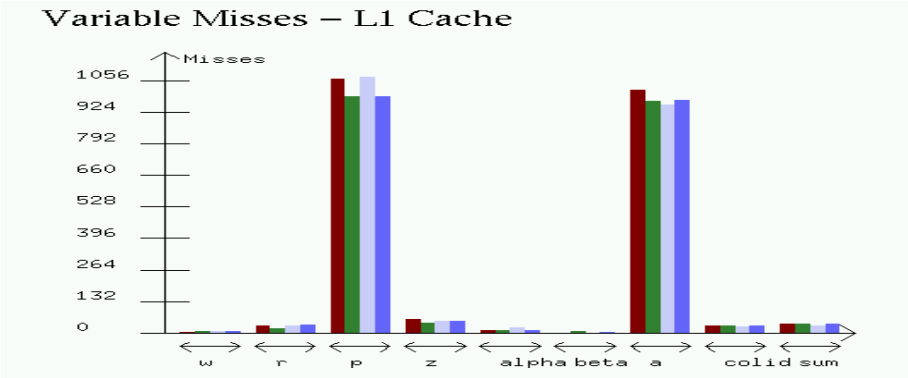

Fig. 2. Sample view: Variable Miss Overview

behavior with all threads, each showing a dominating misses with variable *p* and *a*. The visualization makes these bottlenecks more visible.

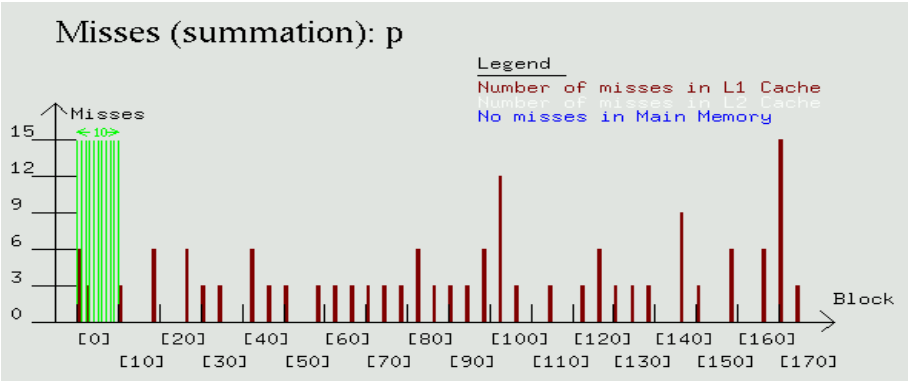

Fig. 3. Sample view: Data Structure Information

Figure 3 is the Data Structure Information view that exhibits the miss distribution within a single array, in this case array *p*. Within this diagram, the array is divided into data blocks in size of a cache line and the number of misses to each block is presented. As shown in Figure 3, this view allows the user to detect individual data blocks with excessive cache misses. For a focus of this hot block, a mask (the vertical lines on the most left side of the diagram) is provided and can be used to enclose the specific block for acquiring information about the corresponding array elements contained in it.

The block-based miss statistics can also be visualized in a per-function way using the 3D Phase Information view of YACO. This view contains several 2D diagrams, similar to that shown in Figure 3, each corresponding to a single function. This enables to highlight both the critical function and the data blocks.

4 Related Work

Currently, there are mainly three approaches for gathering performance data: hardware counter, simulation, and analysis models. The first approach is usually deployed by performance analyzer to acquire source data. A well-known example is the Intel VTune Performance Analyzer [7] that provides a set of graphical views to help identify performance bottlenecks.

In the area of simulation-based data acquisition, a number of simulators have been implemented. MemSpy [11] and Cachegrind [15] are examples. The former 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 the frequency of cache misses. The latter is a cache-miss profiler that performs cache simulation and records cache performance metrics.

The third approach uses analysis models to statically analyze the source code for predicting the runtime performance and system behavior. These models are often integrated in the compilers. An example is the work introduced in [5], where a framework is built in a compiler to automatically diagnose the cause of cache misses. This framework is based on the Cache Miss Equations (CMEs), an analytical representation of cache misses in a loop nest. The diagnosis is then used to select program transformations that improve cache performance.

Overall, the counter approach is straightforward because it directly deploys the hardware provided by modern processors. However, the acquired data is limited. In contrast, the simulation approach can achieve detailed and comprehensive data, but can not exactly exhibit the runtime behavior due to the simulation accuracy. Analysis models can provide only static information and additionally make the complex compiler more complicated. We intend to utilize the hardware resource and at the same time to deliver detailed information that allows the detection of more concrete bottlenecks. For this purpose, the developed profiling tool focuses on data structures and especially has the ability of showing the cache performance within a data block.

5 Conclusions

In this paper we introduce a profiling tool for acquiring cache performance data using hardware counters. Goal of this work is to provide the users possible support in their task of cache locality optimization, in the base of restricted information which can be gathered by available hardware resources. A specific feature of this profiler is the ability of mapping the counter information to data structures in the source program and showing the most concrete access bottlenecks, actually the optimization targets.

As the initial step, we have applied the profiling tool to analyze the cache problems of several applications. In the following, we will start with the optimization process and use traditional optimization strategies, such as data and code transformation, to correct the detected memory penalty.

References

1. J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI For Hardware Performance Monitoring On Linux Systems. In *Linux Clusters: The HPC Revolution*, June 2001.
2. D. Bailey et. al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994.
3. S. Fung. Improving Cache Locality for Thread-Level Speculation. Master's thesis, University of Toronto, 2005.
4. K. Fürlinger and M. Gerndt. Analyzing Overheads and Scalability Characteristics of OpenMP Applications. In *Proceedings of the 7th International Meeting on High Performance Computing for Computational Science*, July 2006.
5. S. Ghosh, M. Martonosi, and S. Malik. Automated Cache Optimizations using CME Driven Diagnosis. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 316–326, 2000.
6. HP. Perfmon Project Web Site. available at http://www.hpl.hp.com/research/linux/perfmon/.
7. Intel Corporation. Intel VTune Performance Analyzer. available at http://www.cts.com.au/vt.html.
8. Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*, volume 1–3. 2002. available at http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm.
9. Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 1–3. 2004. available at Intel's developer website.
10. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
11. M. Martonosi, A. Gupta, and T. Anderson. Tuning Memory Performance of Sequential and Parallel Programs. *Computer*, 28(4):32–40, April 1995.
12. 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 LNCS, pages 694–703, Sorrento, Italy, September 2005.
13. Sun Microsystems. *UltraSPARC IIi User's Manual*. October 1997. available at http://www.sun.com/processors/documentation.html.
14. E. Welbon and et al. The POWER2 Performance Monitor. *IBM Journal of Research and Development*, 38(5), 1994.
15. WWW. Cachegrind: a cache-miss profiler. available at http://developer.kde.org/šewardj/docs-2.2.0/cg_main.html#cg-top.