On the Cache Access Behavior of OpenMP Applications

Jie Tao and Wolfgang Karl
Institut für Rechnerentwurf und Fehlertoleranz,
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
E-mail: {tao,karl}@ira.uka.de

Abstract:

The widening gap between memory and processor speed results in increasing requirements to improve the cache utility. This issue is especially critical for OpenMP execution which usually explores fine-grained parallelism. The work presented in this paper studies the cache behavior of OpenMP applications in order to detect potential optimizations with respect to cache locality. This study is based on a simulation environment that models the parallel execution of OpenMP programs and provides comprehensive information about the runtime data accesses. This performance data enables a detailed analysis and an easy understanding of the cache operations performed on-line during the execution.

1 Introduction

Shared memory models have been widely used for programming parallel machines due to their ability to provide the programmers with a simple model for specifying parallelism in applications. As a result, various shared memory models have been developed over the last years. Among these models, OpenMP is becoming especially important, since it allows incremental parallelization, and is portable, scalable, and flexible. In addition, OpenMP supports the parallelization of a wide set of applications ranging from scientific codes to any existing sequential programs.

However, like with any other programming models, many OpenMP programs face the efficiency problem with respect to parallel execution that usually exhibits bad performance on multiprocessor systems. This is caused by various issues, among which poor cache locality is rather critical.

Modern architectures traditionally deploy several caches in deep memory hierarchies in order to bridge the widening gap between memory and processor speed, which has a drastic impact on the overall performance of current systems. Due to the complicated access pattern of applications, however, the advantage of caches can usually not be fully exploited resulting in the fact that still a large number of memory accesses has to be performed in the main memory at runtime during the execution. This renders locality optimization with respect to cache utility as necessary. A prerequisite for this is the information about the application's memory access characteristics and the runtime cache behavior.

Existing approaches for acquiring performance data can roughly be divided into three categories:

- Hardware counters built into the processor. On modern microprocessors there exists a small set of registers that counts events, the so-called hardware performance counters [In98]. These counters are capable of monitoring the occurrences of specific signals and provides valuable information about the performance of critical regions in programs.

- Hardware monitors connected to a specific memory location. Several multiprocessors deploy hardware performance monitors to count bus and network events [Gu92].

- Simulation systems covering the complete memory hierarchy. Within this approach, a simulation environment is established that models the target architecture and the execution of application codes.

The first option, the hardware counters, allows precise and low–intrusive measurements during the execution of applications, but is restricted to very specific events like the total number of cache misses or the number of memory accesses. This information is therefore often not sufficient for a detailed optimization. The hardware monitors, on the other hand, are capable of offering fine-grained performance data, but with high hardware cost and intrusion delay thereby losing the ability to capture all accesses when deployed within the processor core.

Hence, we use the simulation approach that allows the acquisition of complete performance information and easier study of access behavior on various program regions and phases. For this, an OpenMP simulation infrastructure has been developed, which is based on an existing multiprocessor simulator SIMT [TSK03] and the Omni OpenMP compiler [KSS00]. While Omni is used to transform the OpenMP directives, SIMT models the OpenMP execution on target architectures and offers detailed information about the cache access behavior. This allows to analyze the runtime data layout and detect optimization locations, forming a general tool for tuning OpenMP applications.

The rest of the paper is structured as follows. Section 2 briefly describes the simulation environment with a focus on the combination of SIMT with Omni. This is followed by an overview of the benchmark applications used for this study in Section 3. In Section 4, experimental results concerning cache behavior and some initial optimizations are discussed. The paper concludes with a short summary and some future directions in Section 5.

2 The OpenMP Simulation Environment

Simulation is regarded as a feasible approach for understanding applications' execution behavior and evaluating novel architectures. Over the last years, many simulation tools targeting multiprocessor machines were developed and have been used for improving hardware designs and system performance. In order to study the memory access behavior of parallel applications on NUMA (Non Uniform Memory Access) machines, such a simulator, called SIMT [TSK03], has been developed and used to optimize program codes with the goal of minimal inter-node communications.

SIMT Overview SIMT is an event-driven multiprocessor simulator modeling architectures with global memory abstractions. It directly runs the executable on host machines and simulates the multithreaded execution of parallel applications. As it aims at supporting the research work on memory system, SIMT contains a detailed memory hierarchy simulator, which models multi-level caches with various cache coherence protocols and distributed shared memory with a spectrum of data allocation schemes. In addition, SIMT uses a specific monitoring component to collect performance data. This monitoring component can be selectively combined with any location in the memory hierarchy, allowing the acquisition of complete and accurate performance data about runtime memory references.

SIMT, on the top level, can roughly be divided into a front-end and a backend. The former simulates the execution of multiple threads running on multiple processors in parallel and generates events of interest, while the latter is a target system simulator invoked every time a significant event occurs. Events are memory references and synchronization primitives, and are generated by instrumenting the assembly code with an augmenter.

For native execution of parallel threads, SIMT provides a thread package that supports one thread on the host machine for each simulated application thread, in addition to a simulator thread for executing the simulation code and the architecture simulator. SIMT switches context between these threads when a specific event occurs. Besides this, SIMT schedules the threads in the same manner like on actual multiprocessor systems and computes the execution time in a fashion as if the code would be executed in parallel.

At the end of the simulation SIMT provides the elapsed execution time and simulated processor cycles, number of total memory references, and number of hits and misses to each cache on the system. In addition, its monitor simulator provides memory access histograms with respect to caches, main memories, and the complete memory system. The cache histogram contains information about loading, hit/miss, and replacement on each cache line, while the memory histogram records the accesses to all pages on the whole virtual space and the complete histogram offers numbers of access hits on each location of the memory hierarchy at word granularity. These histograms are modified periodically and can be accessed during the simulation of an application, enabling hence operations of any on-line purpose.

Overall, SIMT has been developed to be a general tool for system design and performance prediction. The main contribution of SIMT is the memory modeling and the comprehensive performance data about the runtime memory accesses. This allows SIMT to aid the users in the task of analyzing the cache access behavior, memory access behavior, and applications' execution behavior, and of studying the impact of locality optimizations on the memory system.

Simulation of OpenMP As described above, SIMT is a feasible multiprocessor simulator capable of exhibiting the access behavior of memory systems and evaluating the parallel execution of applications. However, it is initially designed for running C/C++ codes, like those within the SPLASH-2 benchmark suite [WOT$^+$95], which use ANL like m4 macros to express parallelism. In order to simulate the OpenMP execution, modifications and extensions are necessary with both SIMT and OpenMP compilers.

For SIMT the most tedious work has been done to allow nested parallelism, which can be found within an OpenMP program. For this the structure of SIMT's thread package has been changed in order to reuse the thread structures for a new parallel construct. On the compiler side, a new OpenMP library is needed for transforming the semantics from real multithreads to simulated multithreads. For this the Omni OpenMP compiler [KSS00] has been deployed.

Omni is a source-to-source OpenMP compiler translating C and Fortran77 programs with OpenMP directives into C code suitable for compiling with a native compiler linked with the Omni OpenMP runtime library. In order to enable SIMT contexts, several functions within the Omni OpenMP library have been rewritten: (1) the ompc_do_parallel function is replaced with a subroutine that creates SIMT user-level threads. In this way OpenMP threads can be simulated. (2) functions for implementing synchronization primitives, such as locks and barriers, are rewritten using SIMT semantics. This is necessary because the traditional OpenMP implementation of these primitives results in deadlocks on the simulation platform due to SIMT's thread scheduling mechanism. Within SIMT, the execution control switches from thread to thread in case that a read or write event occurs. This indicates that a lock operation, for example, which issues a memory reference via setting the lock variable and switches the execution to another thread before the unlock is performed, can cause deadlocks. (3) functions with respect to scheduling are specially handled in order to grant correct computation distribution. Currently, SIMT only supports static scheduling, which statically assigns parallel work to threads in a round-robin fashion. (4) functions concerning sequential regions and ordered execution are replaced. An example is ORDERED that forces threads to run in a specific order. On an actual execution, this order is maintained with a global identifier that specifies the next thread to run. Threads, whose *id* does not match the global identifier, have to wait until the active thread leaves the ordered region and modifies the global identifier. For simulation, however, this scheme can not be used because threads are actually sequentially executed. This means that the execution control, when owned by a thread waiting for the permission to enter the ordered region, can not be transfered to the active thread for modifying the global identifier. For tackling this problem, we use explicit events and appropriate handling mechanisms that are capable of forcing context transformation between simulated threads. For other OpenMP pragma and directives similar work has also been done.

In summary, we have implemented the simulation of OpenMP based on SIMT and Omni. Actually, this approach can be applied to other OpenMP compilers. For instance, we have made a new OMP library for the ADAPTOR compiler [AD02] in the same way, allowing the simulation of Fortran90 applications.

3 Benchmark Applications

In order to fully understand the various aspects of cache access behavior, we have simulated a variety of benchmark applications. This includes the *jacobi* code from the OpenMP Organization website [WW], several numerical codes from the Benchmark suite developed by the High Performance Support Unit at the University of New South Wales [St01], and a few kernels from the NAS parallel benchmarks [Bea94, JFY99].

Jacobi *jacobi* is an easy-to-follow kernel program written in Fortran. It solves the Helmholtz equation with a finite difference method on a regular 2D-mesh, using an iterative Jacobi method with over-relaxation. This code is well suitable for studying loop-level parallelization and the associated cache performance.

The main working set is a large 2D matrix used to store the results of current iteration and an additional array used to store the results of the previous iteration. Within each iteration, two loop nests are deployed, where the first loop performs array initialization by copying the additional array to the main matrix, while the second loop executes the sweep operation. The calculation terminates either after a certain number of iterations or the results are within the expected error range. For the OpenMP version both loops are parallelized.

Matmul and Sparse *matmul* and *sparse* are Fortran programs chosen from the benchmark suite described in [St01]. These programs were developed from a set of small pedagogical examples written for an SMP programming course.

matmul and *sparse* both perform matrix multiplication with the former for dense matrix-vector and the latter for sparse matrix-vector in which most elements are zero. The dense matrix is stored and computed in the normal row-column manner. For the sparse matrix, however, a compressed storage scheme is used in order to save space and explore execution efficiency. Using this scheme, the nonzero elements of the matrix are stored as two arrays with the first array for the values and the second for the position, i.e. the column number. To multiply this sparse matrix with a dense vector, the *sparse* code performs the dot-product for each sparse row, where each value in the i*th* sparse row is multiplied with the i value of the dense vector and the individual results are accumulated.

NAS OpenMP Benchmarks NAS Parallel Benchmarks [Bea94] were designed for comparing the performance of parallel computers and are widely recognized as a standard indicator of computer performance. The OpenMP version used in this work is acquired from the Omni project website. Selected applications include FT, LU, and MG. All of them are written in C.

FT contains the computational kernel of a three-dimensional FFT-based spectral method. It performs three one-dimensional fast Fourier transformations, one for each dimension. LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular and block upper triangular system of equations resulting from an unfactored finite-difference discretization of the Navier-Stokes equations in three dimensions. MG uses a multigrid algorithm to compute the solution of the three-dimensional scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine.

To parallelize these applications using the OpenMP programming model, the serial versions are optimized in order to more efficiently use several working arrays. Based on the optimized sequential codes, OpenMP directives are inserted for the outer-most parallel loops to ensure large granularity and small parallelization overhead. To further reduce the parallelization overhead, several end-of-loop synchronizations are removed.

4 Experimental Results

In order to understand the cache access behavior, and to find the access bottlenecks and the cache miss sources, we have simulated these applications using the OpenMP simulation platform. The simulated target architecture is a 16-node Symmetric Multiprocessor (SMP) system, with each processor node deploying a 16KB, 2-way L1 cache and a 512KB, 4-way L2 cache. Caches are maintained coherent using the MESI protocol that invalidates cache copies in all processors at each write operation. Since the monitor simulator of this simulation platform is capable of providing detailed information about the overall references performed during the complete execution of an application, we could study the global access distribution, the cache miss characteristics, and the temporal locality of caches.

4.1 Global Overview

First, the simulation environment provides cache access histograms showing references to the complete working set at granularity of cache lines. This enables to aquire a global overview of the runtime memory access distribution and to analyze specific memory regions in detail.

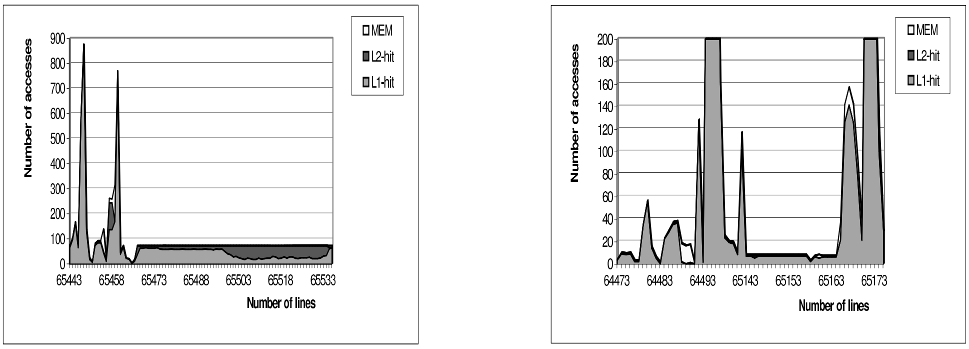

Figure 1: Memory access distribution of *matmul* (left) and *sparse* (right).

Figure 1 visualizes two sample cache access histograms obtained by simulating *matmul* and *sparse* using a 256×256 and a 256×128 matrix individually. The x-axis of the diagrams in Figure 1 shows the individual regions of the complete working set in size of cache lines, while the y-axis presents the number of accesses performed at runtime on each memory location including L1, L2, and the main memory. Due to the large working set size, only the last few memory lines of both applications are illustrated. For the concrete examples, it can be seen that the most memory references are performed on the caches. This can be explained by the fact that both applications are simulated on a system with caches capable of holding the complete matrix. However, even in this case, accesses to the main memory can be clearly seen: for example, memory line 64488–64491 in *sparse*. This indicates that a better cache locality could be achieved.

4.2 Cache Miss Reason

Besides the cache access histogram shown above, the simulation environment provides additional information in order to directly support the work in locality optimization. This includes numerical statistics on causes of cache misses and high-level performance data with respect to data structures within the source code.

A prerequisite for cache locality optimization is to know where and why cache misses occur. Common reasons of cache misses on multiprocessor systems include first reference, replacement, and invalidation. First reference misses occur when data is not properly stored in the memory so that the full line of data fetched on a cache miss can not be efficiently used. Replacement misses occur when a memory line has to be removed from the cache due to mapping overlaps. Both kinds of misses can be reduced by restructuring data in the cache. The same approach can also be used to decrease invalidation misses, where multiple processors usually access different variables on the same cache line.

The monitor facility integrated in the OpenMP simulation environment provides information about the cache miss sources for individual cache regions. This information can be further projected back to the data structure within the source code, enabling hence a detection of possible optimizations.

Application	Variable	Miss rate	First reference	Replacement	Invalidation
matmul	A	62%	8193	110588	0
	x	26%	33	50962	0
	y	10%	33	19701	0
sparse	A	44%	8193	9410	0
	x	2%	33	798	0
	y	5%	33	1775	0
	col	44%	8191	9540	0
jacobi	uold	21%	16436	59656	0
	u	36%	16384	113668	0
	f	41%	16424	130595	0
MG	u	32%	13814	748586	8608
	v	8%	14772	188406	47
	r	23%	10039	541581	5

Table 1: Statistics on L1 cache miss sources.

Table 1 shows the experimental results for small kernels *matmul*, *sparse*, *jacobi*, and the MG code from the NAS Benchmark suite. The first three applications are simulated using a 256×256 matrix, while MG performs its multigrid algorithm on a 64×64×64 grid. The table gives percentage (column 3) of L1 misses on individual data structures (column 2) to the total L1 misses on the complete working set, and the number of specific operations that cause these individual misses.

From Table 1 it can be seen that for all tested codes cache line replacement is the primary reason of cache misses. An example is the *matmul* program. The main working arrays of this code are the dense matrix A and vector x to be multiplied, and the output matrix y. As shown in Table 1, more than 60% L1 misses are caused by accessing matrix A, and

93% of the misses on A is due to replacement. Similar behavior can be seen with vector x, where almost all misses are caused by cache line replacement.

This detection led us to examine the memory distribution of both data structures. It is found that matrix A and vector x map on top of one another in the cache. This results in cache interference that further results in frequent cache line replacement and misses. Hence, we add inter-variable padding between A and x in order to stagger them in the cache. The simulation results with the optimized code show that this optimization significantly reduces the number of replacement and the total cache misses, allowing the code to run 20% faster than the version without padding.

In comparison with cache line replacement, first reference and invalidation do not introduce significant misses for the chosen applications. Besides the *sparse* code that shows a higher miss number due to first reference, the other programs all present slight first reference misses: less than 10% of the total L1 misses. For invalidation, only MG shows a small number, while invalidations with other codes do not cause cache misses.

Overall, the statistical information provided by the simulation environment enables to find cache access bottlenecks and to understand the underlying causes of cache misses. This knowledge can direct the programmers to use appropriate cache optimization techniques to tune applications towards better runtime performance.

4.3 Temporal Locality

The cache access pattern is often different within distinct program phases and functions. Hence, a further insight into these individual code regions can help to handle them separately and respectively. This kind of analysis can be based on the temporal information achieved by the so–called monitor barriers, which trigger a complete swap out of all partial monitoring results followed by a full reset.

Figure 2 gives two such partial results with respect to phases and functions separately. Both diagrams in this figure show the number of L1 and L2 misses within a specific program phase or a function. The information in the left diagram is acquired by simulating the FT code using a data set size of 64×64×32, while the right diagram presents the behavior of different LU functions for performing the equation factorization on a 12×12×12 grid.

For the FT code in Figure 2, phases are identified by the synchronization primitive *barrier*. This includes specific phases for startup, processing, and postprocessing. This also indicates single iterations within the computation itself in the case of iterative methods. For the concrete example, the first phase corresponds to the initialization process, where the 3-D data elements are filled with pseudo numbers. The following three phases represent the process to perform the 3-D fast Fourier transformation (FFT) with each phase for a 1-D FFT in one dimension. The last three phases are within the inverse FFT that starts with the third dimension.

As shown in Figure 2, the number of L2 misses varies slightly between phases. However, the L1 misses differ significantly, especially between the pair for FFT and inverse FFT (i.e. phase2–phase7, phase3–phase6, and phase4–phase5). This result means that the methods for performing inverse FFT are specially important for an efficient execution of the FT

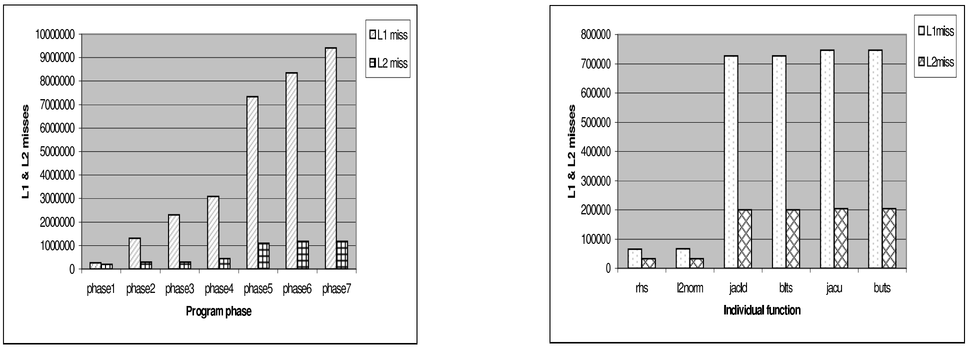

Figure 2: Cache misses within different phases of FT (left) and functions of LU (right).

code.

Besides distinct program phases, like those within the FT program, individual functions within a program can also significantly influence the performance. LU is such an example.

The LU code factorizes the equation into lower and upper triangular systems. The systems are solved using the SSOR algorithm in the following step: (1) the steady-state residuals are computed using routine *rhs*. (2) the L2 norms of newton iteration residuals are calculated using *l2norm*. (3) the lower triangular and diagonal systems are formed with *jacld* and solved with *blts*. (4) finally, the upper triangular systems are formed (*jacu*) and the upper triangular solution is performed (*buts*).

From the right diagram of Figure 2 it can be seen that the four main routines in the SSOR solver are more critical for the cache misses. This indicates that optimizations within these routines can probably significantly improve the cache performance. Actually, the SSOR algorithm can be implemented in parallel using different approaches. These approaches result in different data access pattern and some of them can even keep a processor working on the same data during the whole computation. The *pipeline* approach that transfers the processing on the same block from one processor to another, for example, clearly shows better cache utilization and better performance.

In summary, the temporal information provided by the simulator allows to analyze the cache access pattern of individual program phases and routines. This directs the user to perform fine grain tuning on critical regions of the application program.

5 Conclusions

The OpenMP shared memory programming model is being increasingly addressed due to its portability, scalability, and flexibility. However, as it is a fact for any computer system with uniprocessor or multiprocessors, poor cache locality usually causes inefficient execution of applications. This paper presents our research work on cache locality analysis with the goal of detecting possible optimizations with respect to data allocation and code transformation.

This work is based on an OpenMP simulation environment established on top of an existing multiprocessor simulator for NUMA machines. This OpenMP simulation platform models the parallel execution of OpenMP applications on SMP systems. Its main contribution is the cache simulator and the detailed information about the runtime cache accesses. The latter allows us to analyze the cache access pattern of various OpenMP applications, and to detect the cache miss reasons and the access bottlenecks. Initial optimizations based on this analysis has shown a significant performance gain. In the next step of this research work, more applications, especially those realistic ones, will be simulated and optimized. In addition, the simulation in most cases still shows a slowdown of 1000 factors in comparison with the real running. The established simulator will be parallelized in the near future.

References

[AD02] ADAPTOR. High Performance Fortran Compilation System. 2002. available at http://www.gmd.de/SCAI/lab/adaptor.

[Bea94] Bailey, D. and et. al: The NAS Parallel Benchmarks. Technical Report RNR-94-007. Department of Mathematics and Computer Science, Emory University. March 1994.

[Gu92] Gupta, S.: Stanford DASH Multiprocessor: the Hardware and Software. In: *Proceedings of Parallel Architectures and Languages Europe (PARLE'92).* S. 802–805. June 1992.

[In98] Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual.* volume 1–3. Published on Intel's developer website. 1998. available at http://www.intel.com/design/PentiumII/manuals/.

[JFY99] Jin, H., Frumkin, M., and Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011. NASA Ames Research Center. October 1999.

[KSS00] Kusano, K., Satoh, S., and Sato, M.: Performance Evaluation of the Omni OpenMP Compiler. In: *Proceedings of International Workshop on OpenMP: Experiences and Implementations (WOMPEI).* volume 1940 of LNCS. S. 403–414. 2000.

[St01] Standish, R. K.: SMP vs Vector: A Head-to-head Comparison. In: *Proceedings of the HPCAsia 2001.* September 2001.

[TSK03] Tao, J., Schulz, M., and Karl, W.: A Simulation Tool for Evaluating Shared Memory Systems. In: *Proceedings of the 36th ACM Annual Simulation Symposium.* Orlando, Florida. April 2003.

[WOT$^+$95] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* S. 24–36. June 1995.

[WW] WWW. OpenMP Architecture Review Board. available at http://www.openmp.org/index.cgi?samples+samples/jacobi.html.