

A Cache Simulation Environment for OpenMP

Jie Tao¹, Thomas Brandes², and Michael Gerndt¹

¹Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik, Technische Universität München
Boltzmannstr.3, 85748 Garching, Germany
{tao, gerndt}@in.tum.de

² Fraunhofer-Institute for Algorithms
and Scientific Computing (SCAI)
Schloss Birlinghoven, 3754 Sankt Augustin
thomas.brandes@scai.fraunhofer.de

Abstract

Modern architectures usually employ several caches to bridge the gap between memory and processor speed. Due to the relatively small size of caches and the suboptimal locality of the application's access patterns, however, a large amount of memory accesses is still performed in main memory. Cache locality is hence highly addressed in the research area of high performance computing.

This paper presents a simulation environment that can be used as a comprehensive platform for understanding the cache behavior of OpenMP applications on modern SMP architectures. The simulator consists of comprehensive mechanisms for modeling the memory system and the OpenMP execution. Especially, it can provide detailed information about cache references, forming therefore a feasible tool for studying and improving both spatial and temporal locality of access patterns.

1 Introduction

The rapid increment in CPU speed has resulted in a considerable gap between memory and processor performance. Cache, as a fast storage, bridges this gap with its ability to prevent a large percentage of memory references from directly accessing the main memory and hence helps to reduce memory access latency. However, due to the relatively small cache size compared to the size of data structures in scientific applications as well as due to complex and unpredictable access patterns of applications, cache is usually not best exploited and many references have still to be performed in main memory. Therefore, lots of projects focus on exploring cache locality.

The work described in this article is part of the EP-Cache project ¹, which explores techniques for efficient programming on cache architectures. The goal is to enhance the performance of realistic OpenMP Fortran applications on Symmetric Multiprocessor (SMP) systems through a better usage of caches by enhancing the program's locality. For this, a new hardware monitor is being developed that allows to gather more detailed information on the memory access patterns without significant intrusion. In addition to standard cache miss counters, the hardware provides data structure-related cache miss counts and access patterns. The information provided by the hardware monitor will be utilized by VAMPIR and an Automatic Monitor Control (AMC). VAMPIR is used to visualize the resulting data in a user-understandable way, while AMC

¹More information at <http://www.gmd.de/SCAI/EP-CACHE/>.

automatically detects program regions where an improvement of data locality might be possible. Based on this information, applications can be optimized with respect to cache locality.

While the hardware monitor is still not available, a simulation environment is currently being implemented in order to enable earlier development of tools and optimization methodology. The environment is not only a standard cache simulator, but allows also to simulate arbitrary OpenMP programs and to collect data structure specific cache information via the simulated hardware monitor. In this paper, we describe this simulation environment with a focus on its contributions, implementation details, and sample use.

This simulation environment is based on an existing multiprocessor simulator SIMT, which models the parallel execution of C/C++ programs on shared memory machines. SIMT is a general toolkit capable of being used for performance prediction, system design, and application optimizations. It can model a large range of multiprocessor systems with varying numbers of processors, and different memory hierarchies, access latencies, and data distributions. Within SIMT, applications have to use specific macros for specifying parallelism and synchronizations. In order to simulate OpenMP, this simulator was extended. In addition, a Fortran OpenMP compiler ADAPTOR has been modified with a new OpenMP library that uses SIMT contexts. The result is a comprehensive simulation environment, which is capable of providing detail information about the runtime cache behavior of OpenMP programs.

In the following, we first give an overview of this approach in Section 2, followed by an introduction of SIMT and ADAPTOR in Section 3. In Section 4, the implementation details of simulating OpenMP Fortran programs will be discussed. Section 5 shows the initial experimental results based on a few kernel benchmarks. The paper concludes in Section 6 with a short summary and some future directions.

2 The Approach

As mentioned above, we use SIMT [8], a multiprocessor simulator, to model the parallel execution of OpenMP programs. SIMT is based on an SPMD model and all applications have to be written using C/C++ in combination with ANL like m4 macros. In addition, SIMT models user-level threads meaning that parallel threads are executed sequentially on a single processor system with specific scheduling methodology for thread switching. In order to simulate OpenMP, the actual multithread structure has to be adapted to SIMT sequential-thread structure.

Our initial idea was to implement a general OpenMP simulation environment, which is independent of compilers. This means we intended to simulate OpenMP by only exchanging the traditional PThread library for OpenMP with a SIMT thread library switching the semantics between real threads and simulated threads, without any change of the OpenMP compilers. In this way, OpenMP applications can be generally simulated, no matter what compilers are used for generating the executable.

However, we faced a problem with this approach due to the thread handling of compilers. The runtime system of an OpenMP compiler traditionally creates all participating threads during initialization in order to reduce the overhead for thread creation. These threads first register themselves by incrementing a specific counter and then sleep until they are needed. At the same time, the master thread, after creating the helper threads, continues with the execution of the program code. When a parallel construct arises, the master wakes up the helper threads and assigns each with a parallel task. Within this scenario, the master is allowed to go further with

its task only when all threads have been correctly created, while a helper thread has to wait until a task is scheduled.

This scenario results in a deadlock on the simulation platform. Within SIMT, multithreads are simulated in a manner that only a single thread is active at a time. In case of active master, the execution control is fixed to the master while waiting for the registration of helper threads, and can therefore not be transferred to the helpers. On the other hand, in case that a helper thread is active, it holds the control while waiting for a parallel task, so that the master can not go further to the point, where parallel tasks can be generated and helper threads can advance.

A more powerful simulator that allows concurrent execution of multiple threads could be a solution. To our knowledge, however, no multiprocessor simulator supports real multithreads and additionally we have no experience with this issue. As the first step, an easier way has been chosen, i.e. to exchange the OpenMP library of a compiler. We use ADAPTOR [1, 2], a High Performance Fortran (HPF) Compilation System with OpenMP extensions.

3 Basis Platforms

SIMT and ADAPTOR are both basis platforms deployed for simulating OpenMP. While ADAPTOR implements a source-to-source translation of OpenMP Fortran applications to sequential codes in combination with OMP library calls, SIMT is used to establish the fundamental simulation structure and provide comprehensive performance data.

3.1 SIMT: An Event-driven Multiprocessor Simulator

SIMT [8] is a general simulation tool specially developed for evaluating the memory system and shared memory applications on NUMA (Non Uniform Memory Access) machines. It is built on top of Augmint [5], a fast execution driven multiprocessor simulation toolkit for Intel x86 architectures. It uses Augmint's memory reference generator to drive the simulation and implements a backend for modeling the target architectures: NUMA machines and SMPs. The backend comprises mainly mechanisms for simulating the entire memory hierarchy and the interconnection traffic. SIMT runs applications from the SPLASH-2 suites [9] and any application written in C/C++ together with m4 macros in a fashion similar to SPLASH and SPLASH-2 applications.

Building an application SIMT is an event-driven simulator using native execution on the host machine. It inserts calls to the simulator directly into the program to be simulated so that the execution of the program explicitly generates events such as memory references and synchronization events. This is done by compiling the source code with the simulator linked in.

Figure 1 shows the procedure to build an executable. As shown in this figure, the application source code, in which parallel constructs are expressed using the m4 macros, is first processed by an m4 macro processor, along with the Augmint macro library. A GNU C compiler then compiles the resulting C code into x86 assembly code, which is augmented by the *augminter Doctor* with instrumentation code that updates the simulated clock, calculates data addresses, and generates events. Finally, the augmented code is assembled and linked with the threads package and the architectural model (backend) to produce a single executable.

Thread structure In order to support native execution of application threads, SIMT provides a threads package that supports one thread on the host machine for each simulated application thread, in addition to a simulator thread for executing the code of SIMT. The augmented

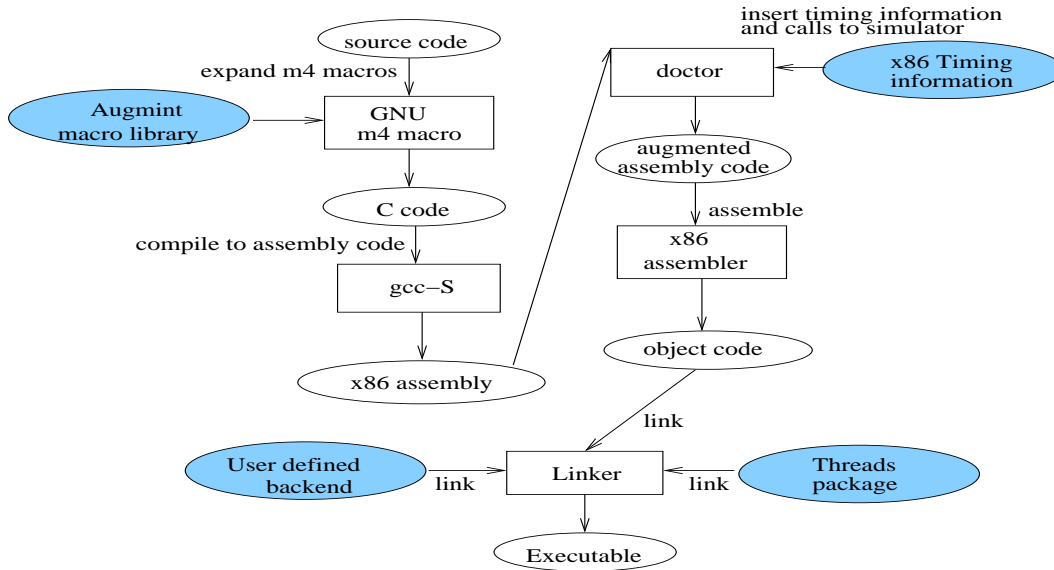


Figure 1: Building an SIMT application.

application consists of the code that switches context to the simulator thread at specific events. SIMT handles these events and typically creates and schedules tasks for these events to be simulated by the backend.

The work starts with the simulator thread that first creates a task queue and schedules a task that switches context to the main application thread as the first task in the queue. When this task is extracted from the queue and executed, control is passed to the main application thread and the augmented application code is executed until the first event is generated. At this point, the main application thread switches context to the simulator thread which then handles the generated event. Based on this event, a task is created and inserted into the task queue to be executed at a specific simulation cycle. When this task becomes the first one on the task queue, SIMT extracts it and executes a function associated with the task. Such functions can generate and schedule more tasks which wait for execution on the task queue. After the completion, a task can signal SIMT to allow the next task on the task queue to be executed or the corresponding application thread to advance.

Backend: the architecture model SIMT, on the top level, is composed of a front-end and a backend. The former is a memory reference generator inherited from Augmint, while the latter is a self-developed package that models the target architectures. As SIMT aims at research work on the memory system, the back-end contains mainly mechanisms for modeling the complete memory hierarchy in detail. This includes a flexible cache simulator which models caches of arbitrary levels and various cache coherence protocols, a memory control simulator which models the management of shared and distributed shared memories as well as a set of data allocation schemes, and a network mechanism simulating the interconnection traffic. More details about the backend can be found in [8].

In addition, SIMT models a hardware monitor [3] developed for tracing the inter-node communications. The monitor simulator [7] is designed and implemented as an independent component and can be selectively inserted into any level of the memory hierarchy. This allows to gather

information about each component of the memory hierarchy and hence a concrete optimization of individual locations.

SIMT provides extensive statistics about the execution of an application at the simulation process. This includes elapsed simulation 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.

3.2 ADAPTOR: Source-to-source OpenMP Compiler

ADAPTOR is a Fortran source-to-source translation system that has been developed at the SCAI Institute of FhG during the last years [1].

The first versions of ADAPTOR has been used as a High Performance Fortran (HPF) compilation system that compiles data parallel HPF programs for distributed memory machines. The HPF mapping directives define a mapping of data objects (arrays) to abstract processors. Data objects that have been mapped to a certain abstract processor are regarded as being owned by that processor. Based on the ownership of data (owner-computes rule), the distribution of computations to the abstract processors and the necessary communication between processors is derived automatically. The parallel program generated by ADAPTOR is executed by a set of processes that communicate via MPI.

In later versions ADAPTOR has been extended in such a way that data parallel programs can also be compiled for shared memory machines by using threads and the global address space in order to generate more efficient code. This execution model yields better performance since the global address space reduces the overhead for non-local data accesses and avoids memory overhead implied by replication of data and by non-local copies of data. In a similar way ADAPTOR has been extended to support HPF compilation, where a multi-threaded node program is generated for a hierarchical execution model on clusters of SMPs.

For the EP-CACHE project, ADAPTOR has been extended to also support the OpenMP directives. It translates the OpenMP parallel regions into own subroutines that will be executed by the threads. Corresponding runtime support realizes the synchronization and the scheduling of parallel loops.

4 OpenMP Simulation

In order to model the multithreaded execution of OpenMP applications, SIMT is extended with respect to thread operation and task creation. A significant step towards this is to enable nested parallelism on the simulator. However, the most tedious work has been done on the side of the compiler.

Combination of SIMT with ADAPTOR Our first endeavor was to integrate the simulation infrastructure into the compiler environment. For this, an OpenMP Fortran program has to be first translated to a sequential code with OMP library calls within it. This work is

done by ADAPTOR. In the second step, the sequential Fortran code is compiled to an assembly code using a chosen Fortran compiler such as PGI, Intel, or any others, depending on the user specification. The work of SIMT starts at this point by first instrumenting the assembly code and then generating an executable, in the similar manner that it processes a C/C++ application as shown in Figure 1. For this step an issue has to be noted that is specific for OpenMP. The problem lies in the instrumentation process of the assembly code. Since the augminter *Doctor* is combined with *gcc*, the assembly code generated by some Fortran compiler, which use different semantics, potentially can not go through the instrumentation procedure. In this case, preprocessing has to be done before the assembly code is augmented by *Doctor*.

Adaptive thread structure In order to avoid deadlocks, the OpenMP application creates no threads during initialization. Instead, the master thread, the so-called application thread in terms of SIMT, directly reaches the parallel construct, where ADAPTOR generates a subroutine for the parallel region, like the most other OMP compilers do. This subroutine would be, in the real execution, assigned to several threads according to the user specified number. Correspondingly, SIMT does the same work. Figure 2 illustrates a small example.

ADAPTOR	SIMT
<pre> dalib_pthread (parallel_routine, parameter) { parallel_do (thread_number-1); parallel_routine; } </pre>	<pre> for i=1; i<thread_num; i++ CREATE (parallel_routine, parameter); parallel_routine; </pre>

Figure 2: Adapting OpenMP thread operation to simulation.

On real execution, as shown in the left side of Figure 2, each chosen thread, including the master, is assigned with the `parallel_routine`. Similarly, SIMT creates corresponding user-level threads using macro `CREATE`. These threads, together with the application thread, run alternately on the processor, however, they are scheduled in the same manner like on actual multiprocessor systems and, in addition, SIMT counts the execution time in a fashion as if the code would be executed in parallel.

Memory distribution SIMT uses its own address space for simulating multiple threads. This single address space is divided into a shared area, a private region, and a stack. Shared data is allocated (using macro `G_MALLOC`) in the shared region and is visible to all processors. Data and stack for each thread is allocated in the private region and is only accessible to the specific thread. This kind of distribution allows to implement the cache coherence by only handling the data within the shared area. For OpenMP Fortran, however, the location of shared data can not be explicitly specified within the source code. Rather shared data is placed in the private region, together with private data. Hence, cache coherence has to be performed at each memory access, causing high overhead. This problem will be overcome in the near future. Currently, we are extending ADAPTOR to allow explicit allocation of data.

Synchronization Locks and barriers are the common used synchronization primitives for achieving correct access to shared variables. The traditional OpenMP implementation of these primitives, however, can not be directly deployed by SIMT for simulation due to its 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 result in deadlock. Therefore, we have replaced all synchronization related functions in the ADAPTOR OMP library with functions using SIMT semantics that guarantee no deadlocks occur.

Scheduling Currently, SIMT only supports static scheduling, which statically assigns parallel work to threads in a round-robin fashion. This kind of scheduling is done within OpenMP by counting the lower and upper boundaries of the loops according to the iteration space and the thread *id*, which is chronologically assigned within the participating threads. For simulation, however, each parallel construct creates new threads and each thread acquires an *id* number greater than the thread previously created. This causes incorrect computing of boundaries. Hence, specific handling has been done in order to grant correct computation distribution.

Sequential regions OpenMP uses specific directives for forcing sequential and ordered execution of specific regions. For simulation these directives have to be implemented using mechanisms varying from those employed by OpenMP compilers. An example is ORDERED, which forces threads to run in a specific order. On an actual execution, this order is maintained by deploying a global identifier that shows the next running thread. 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 transferred 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.

Summary Overall, we have implemented the simulation of OpenMP based on SIMT and ADAPTOR. Actually, this approach can be applied to other OpenMP compilers. For instance, we have made a new OMP library for the Omni compiler [4] in the same way, allowing the simulation of both Fortran and C applications.

5 First Experimental Results

Using the OpenMP simulator described above, we have studied several applications from the Benchmark suite developed by the High Performance Support Unit at the University of New South Wales [6]. This includes *matmul* for dense matrix-vector multiply, *sparse* for sparse matrix-vector multiply, *mandelbrot* for a Mandelbrot set calculation, and *heat* for solving a heat equation on a 2D grid. In addition, the *jacobi* code from the OpenMP Organization website [10] has been chosen as well. This program solves the Helmholtz equation on a regular mesh, using an iterative Jacobi method with over-relaxation.

First we measured the total misses of L1 and L2 cache on both uni- and multiprocessor systems. For parallel we simulated all applications on a 8-node system. The used working set size is 128×128 for *matmul* and *mandelbrot*, 40×128 for *sparse*, 256×256 for *heat*, and 128×128 for *jacobi*. We simulated 2-way caches using a configuration of 32 KB L1 and 512 KB L2.

Table 1 depicts the experimental results. Within this table, both L1 and L2 miss behavior is presented with three columns of data. The first two columns are the absolute number of misses,

	L1 miss			L2 miss		
	sequential	parallel	seq/par	sequential	parallel	seq/par
matmul	4298	4616	93%	2110	4202	50%
sparse	2753	2964	93%	1349	2611	52%
mandelbrot	16482	32981	50%	2067	32815	6.3%
heat	25000	26274	93%	16508	23861	69%
jacobi	29473	30882	95%	12359	20288	61%

Table 1: Cache misses on both uni- and multiprocessors.

while the third column shows the percentage of the sequential case to the parallel execution.

It can be observed that multiprocessors perform more cache misses than uniprocessors. For the level one cache, this situation is not critical according to the data in Table 1. The fourth column shows a slight difference for most applications, with an exception of the *mandelbrot* code for which the sequential execution performs only a half of the cache misses. For the L2 cache, however, specially worse behavior can be seen with the multiprocessors. In most cases, significant more cache misses are performed.

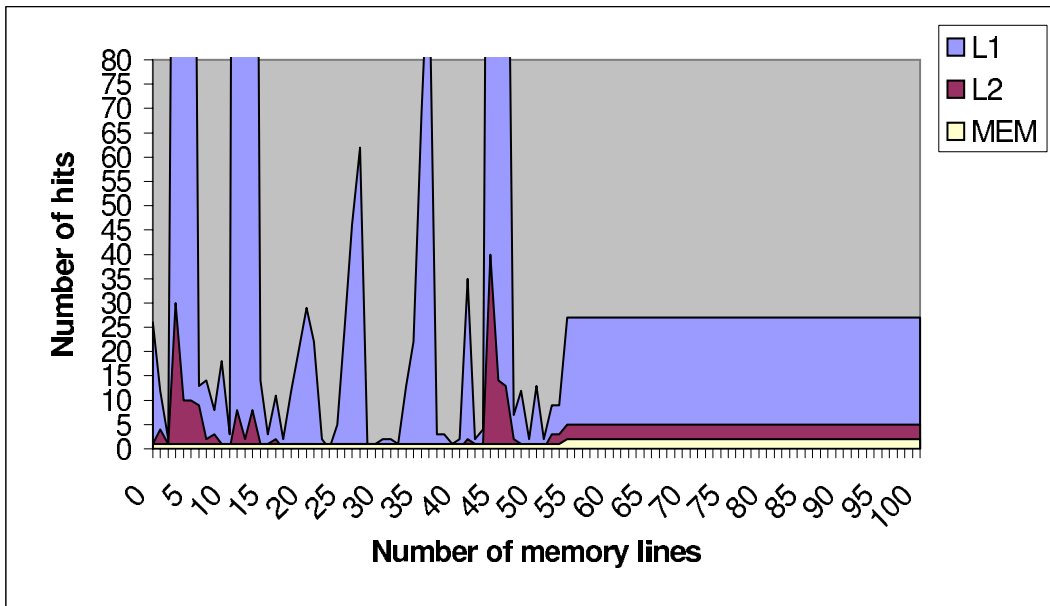


Figure 3: Access histogram of the Jacobi code (first 100 memory lines).

The second measurement was done for complete access histograms. Figure 3 shows such a histogram obtained by simulating the *jacobi* code on a 8 node system. As illustrated in this figure, the access histogram depicts the accesses to the whole working set and the entire memory system at granularity of cache line size. The x-axis shows the accessed locations, while the corresponding numbers of accesses to L1, L2, and the local memory are presented on the y-axis. Since it clearly exhibits the different behavior of each location in the memory hierarchy, this histogram is capable of directing the user to optimize the data placement towards a better

cache locality.

Our last experiment was performed for examining the cache behavior at different program phases distinguished by iterations and synchronization primitive. We use the *heat* program for this study and simulated it on a 4-node system. Within *heat*, each iteration is identified as a program phase, and we measured the cache misses within every single phase.

Figure 4 illustrates the cache miss number in the first few phases of the tested code. In this figure, the first phase represents the initialization process, hence, a high L1 and L2 miss can be seen. The rest phases are iterations and it can be observed that both caches behave similarly over these phases. A specific case is the L2 cache at the second phase, where a high miss number has been shown. This indicates that a potential optimization with respect to L2 cache could be performed and a performance gain could be achieved for this application.

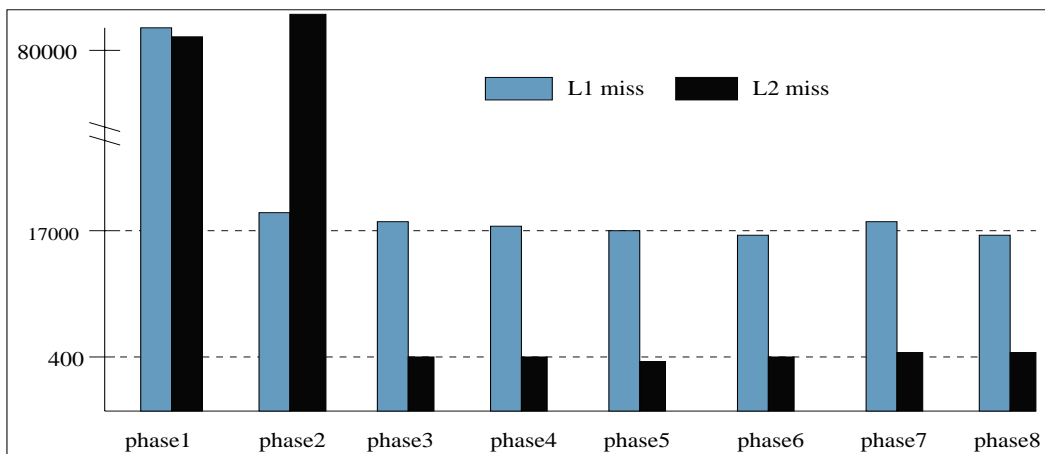


Figure 4: Cache misses within program phases.

6 Conclusions

In this paper, we present a simulation environment for understanding the memory access behavior of OpenMP applications. The simulator is based on an existing simulation tool and an OpenMP compiler, and models the OpenMP execution on target architectures. This simulation tool provides detailed information about the runtime cache accesses and can therefore direct programmers to optimize the cache locality at various granularity.

However, this work is still in its initial phase. The established cache simulator is restricted to specific OpenMP compilers due to the simulation structure of Augmint. Currently, we are working on Valgrind [11], a memory debugger containing a cache profiler modeling caches on uniprocessors. This tool is being used as the memory access generator of our cache simulator. As it models PThread and uses runtime instrumentation that is directly performed on executables, Valgrind does not relying on compilers. The result is a general simulation environment fully supporting OpenMP programs and enabling the use of commercial OpenMP compilers. Another limitation of the current simulator lies on the simulation slowdown that is even as high as factor 1000 in some cases. In the next step of this research work, the simulation process will be par-

allelized. We are intending to provide a fast, general simulation environment for understanding and optimizing the cache access behavior of OpenMP applications.

References

- [1] ADAPTOR. High Performance Fortran Compilation System, 2002. available at <http://www.gmd.de/SCAI/lab/adaptor>.
- [2] T. Brandes and S. Benkner. Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs. In *Proceedings of Euro-Par 2000 Parallel Processing*, volume 1900 of LNCS, pages 647–657, Munich, Germany, September 2000.
- [3] R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, and M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-based Monitoring Approach for DSM Systems. In *Proceedings of Euro-Par'98 Parallel Processing / 4th International Euro-Par Conference Southampton*, volume 1470 of Lecture Notes in Computer Science, pages 206–215, UK, September 1998.
- [4] K. Kusano, S. Satoh, and M. Sato. Performance Evaluation of the Omni OpenMP Compiler. In *Proceedings of International Workshop on OpenMP: Experiences and Implementations (WOMPEI)*, volume 1940 of LNCS, pages 403–414, 2000.
- [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] R. K. Standish. SMP vs Vector: A Head-to-head Comparison. In *Proceedings of the HPCAsia 2001*, September 2001.
- [7] J. Tao. Supporting the Memory System Evaluation with a Monitor Simulator. In *Proceedings of 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 31–38, Genova, Italy, February 2003.
- [8] J. Tao, M. Schulz, and W. Karl. A Simulation Tool for Evaluating Shared Memory Systems. In *Proceedings of the 36th ACM Annual Simulation Symposium*, pages 335–342, Orlando, Florida, April 2003.
- [9] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop 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.
- [10] WWW. OpenMP Architecture Review Board. available at <http://www.openmp.org/index.cgi?samples+samples/jacobi.html>.
- [11] WWW. Valgrind, version 2.0.0. available at <http://developer.kde.org/sewardj/docs-1.9.5/manual.html>.