A Cache Simulator for Shared Memory Systems

Florian Schintke, Jens Simon, and Alexander Reinefeld

Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
{schintke,simon,ar}@zib.de

Abstract. Due to the increasing gap between processor speed and memory access time, a large fraction of a program's execution time is spent in accesses to the various levels in the memory hierarchy. Hence, cache-aware programming is of prime importance. For efficiently utilizing the memory subsystem, many architecture-specific characteristics must be taken into account: cache size, replacement strategy, access latency, number of memory levels, etc.
In this paper, we present a simulator for the accurate performance prediction of sequential and parallel programs on shared memory systems. It assists the programmer in locating the critical parts of the code that have the greatest impact on the overall performance. Our simulator is based on the *Latency-of-Data-Access Model*, that focuses on the modeling of the access times to different memory levels.
We describe the design of our simulator, its configuration and its usage in an example application.

1 Introduction

It is a well-known fact that, over the past few decades, the processor performance has been increased much faster than the performance of the main memory. The resulting gap between processor speed and memory access time still grows at a rate of approximately 40% per year. The common solution to this problem is to introduce a hierarchy of memories with larger cache sizes and more sophisticated replacement strategies. At the top level, a very fast L1 cache is located near the CPU to speed up the memory accesses. Its short physical distance to the processor and its fast, more expensive design reduces the access latency if the data is found in the cache.

With the increasing number of memory levels, it becomes more important to optimize programs for temporal and spatial locality. Applications with a good locality in their memory accesses benefit most from the caches, because the higher hit rate reduces the average waiting time and stalls in arithmetic units.

In practice, it is often difficult to achieve optimal memory access locality, because of complex hardware characteristics and dynamic cache access patterns which are not known *a priori*. Hence, for a realistic cache simulation, two things are needed: a hardware description with an emphasis on the memory subsystem and the memory access pattern of the program code.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 569–578, 2001.

We have designed and implemented a simulator that is based on the *Latency-of-Data-Access Model* described in the next section. The simulator is execution-driven, i.e. it is triggered by function calls to the simulator in the target application. It determines a runtime profile by accumulating the access latencies to specified memory areas.

2 The Latency-of-Data-Access Model

The *Latency-of-Data-Access (LDA)* model [9] can be used to predict the execution characteristics of an application on a sequential or parallel architecture. It is based on the observation that the computation performance is dominated (and also strictly limited) by the speed of the memory system.

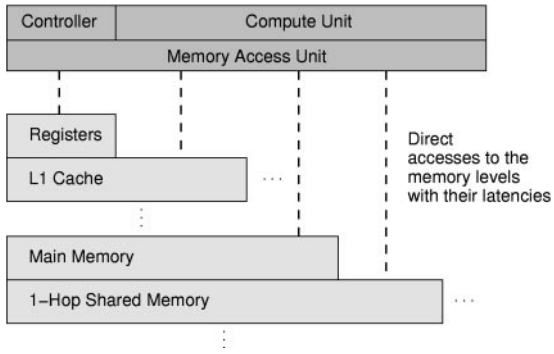


Fig. 1. Concept of the LDA model.

In LDA, data movements are modeled by direct memory accesses to the various levels in the memory hierarchy. The number of data movements combined with the costs of computations provide an accurate estimate on the expected runtime [9].

LDA models the execution of an application by a set of abstract instructions, which are classified into *arithmetic operations*, *flow control*, and *data movements* between the processing unit(s) and the memory system. The data movements are considered to be issued from an explicit load/store unit with direct access to each level of the memory hierarchy (see Fig. 1).

With this abstract model, complex protocols of today's memory hierarchies can be treated uniformly, while still giving very precise results. Cache misses caused by a load operation in a real machine are modeled by a single access to the next lower memory level that holds the data. This is done with the rationale that the time taken by the initial cache miss is much shorter than the time taken by the access to the next lower memory level. Hence, for each load/store instruction, LDA uses the execution cost of that access to the specific memory level.

Note that the precise timing depends on the access pattern: Consecutive memory accesses are much faster than random accesses. The LDA model reflects these effects by adding different costs to the individual accesses.

3 The LDA Simulator

Our simulator [8] gets two sources of input: the configuration of the memory architecture of the target machine, and the memory access pattern of the simulated program. The latter is obtained by manually inserting function calls to the simulator in the program source code.

We have verified the accuracy of our simulator by comparing simulated data to the timing of practical execution runs.

In the following sections we describe the features and architecture of the simulator in more detail and explain its usage with a simple example.

3.1 Features

The LDA simulator can be used to optimize programs, but also to help in the design of the memory hierarchy of a processor. It can be compiled on any system with a standard C++ compiler. The simulated architecture is independent of the computer that the simulator runs on.

The simulator shows the currently accumulated execution time in every time step of the simulation. It is also possible to get the number of memory accesses for a specific memory level.

The simulator allows to split the costs for user-specified memory areas to different cost accounts. With this feature, the execution time on subsets of the address space can be monitored (Sec. 4), which is useful for, e.g., optimizing the data structure for better cache usage.

3.2 Architecture

Figure 2 shows the simulator's architecture described in the *Unified Modeling Language UML* [5]. The actual implementation has been done with the C++ programming language. The simulator consists of two main parts:

- The classes `MachineDef`, `Protocol`, `CacheDef`, `RStrategy`, and `WStrategy` describe the static architecture of a memory hierarchy.
- The classes `Caches`, `CacheLevel`, and `Cacheline` describe the dynamic part. Their behavior depends on the static description.

The classes `Caches`, `Cachelevel`, and `Cacheline` determine which memory levels are accessed for each memory operation and then update their internal counters for calculating the execution time. For this purpose, it is necessary to simulate the behavior of the cache levels and to determine the currently cached addresses.

The user interface is defined in the two classes `MachineDef` and `Caches`.

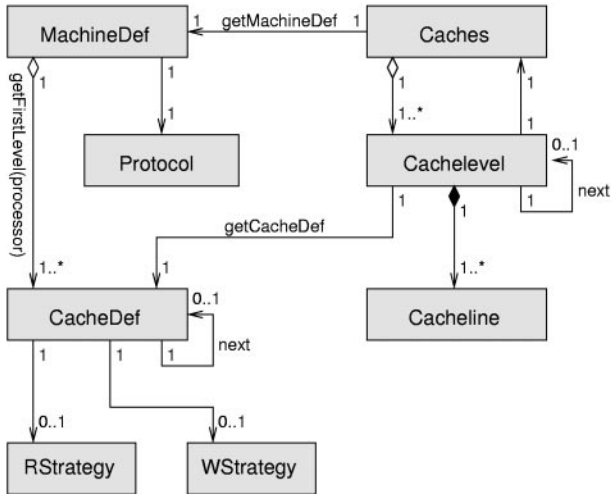


Fig. 2. UML class diagram of the simulator showing the software architecture of the simulator.

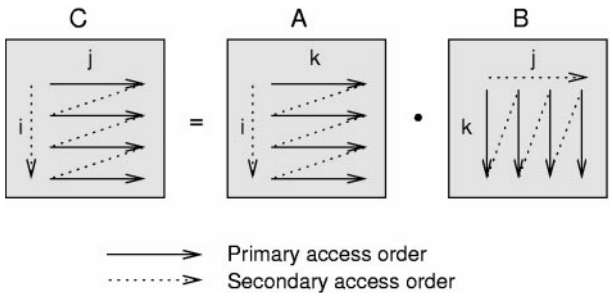
The simulator is execution-driven and the target source code must be instrumented by manually inserting function calls into the program. By the approach not to use an executable editing library like EEL [1], the user has the burden to modify the source code on the one hand, but on the other hand, it allows him to focus on the most relevant parts in the application, e.g. program kernels. Simulation overhead has only to be added where really necessary. An example of an instrumented program code is shown in Fig. 4.

3.3 Configuration

At startup time, a configuration file is read by the simulator that contains the static specification of the memory architecture. Many different architectures can be specified in the same configuration file and can be referenced by a symbolic name. With this feature, several system architectures can be simulated in the same run.

Memory hierarchies of arbitrary depths and sizes may be defined, thereby also allowing to simulate hierarchical shared memory systems. For each memory level, there may be different settings for

- the number of blocks,
- the block size,
- the associativity,
- the replacement strategy,
- the write strategy, and
- the read and write latency.




```
for (i = 0; i < matrixSize; i++)
  for (j = 0; j < matrixSize; j++)
    for (k = 0; k < matrixSize; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Fig. 3. Memory access pattern of an unblocked matrix-matrix multiplication and its implementation (with C memory layout).

For the input configuration file, the hardware characteristics of the target architectures can be obtained either from the technical manuals of the system or by running a suite of microbenchmarks. For configurable systems, only microbenchmarks can be used to determine the actual configuration. In our experiments we used the method of McVoy and Staelin [3]. It measures the memory hierarchy characteristics by enforcing strided accesses to memory blocks of different sizes. Their scheme determines the cache size, the latency and the block size for each memory level.

However, some of the characteristics can not be determined by microbenchmarks. This is true for the cache associativity, the write strategy (write through or write back), and the replacement strategy (LRU, FIFO, random, ...) which must be specified manually.

4 Example

In this section, we illustrate the use of the simulator with an unblocked matrix-matrix multiplication on a single and an SMP multiprocessor system.

4.1 Matrix Multiplication on a Single Processor

As an example we have simulated the matrix-matrix multiplication $A * B = C$. Figure 3 depicts the default memory layout of the matrices. The two-dimensional memory blocks of matrices A and C are mapped to the linear address space of the computer by storing them line by line. Hence, the system accesses the elements of A and C with consecutive memory accesses, resulting in a good spatial locality. The accesses to matrix B, however, occur strided over the memory. The lower spatial locality of matrix B slows down the whole computation.

```
for (i = 0; i < matrixSize; i++)
  for (j = 0; j < matrixSize; j++)
  {
    for (k = 0; k < matrixSize; k++)
    {
      caches.load(&a[i][k] - offset, 4);
      caches.load(&b[k][j] - offset, 4);
      caches.doFloatOps(2);
    }
    caches.store(&c[i][j] - offset, 4);
  }
```

Fig. 4. Instrumented code for the matrix-matrix multiplication.

Temporal locality is given only in the accesses to the result matrix C. The temporal locality increases with growing matrix size (Fig. 6).

In preparation of the simulation, all load and store operations must be searched in the source code and function calls to the simulator must be inserted. Figure 4 shows the instrumented code. Note that the function calls to the simulator `caches.load`, `caches.store` expect the (one-dimensional) array address and the data size as parameters.

In practice, the elements c[i][j] of the result matrix are stored only once to the main memory. Hence in the simulation we can save CPU time by moving the data assignment out of the inner loop as shown in Fig. 4.

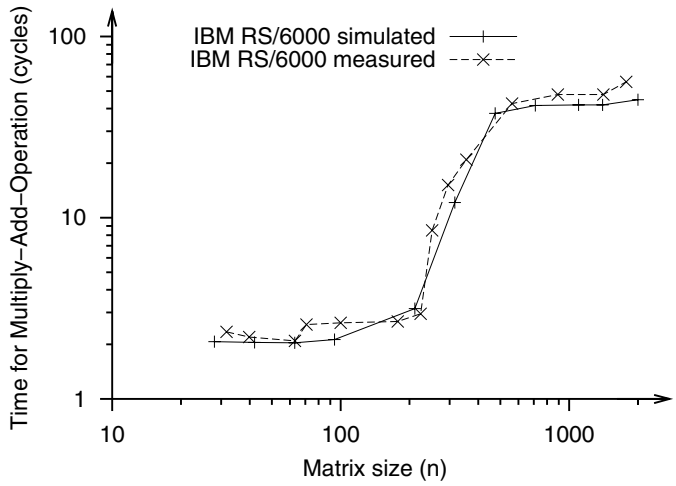


Fig. 5. Measured and simulated execution times of the matrix-matrix multiplication for different matrix sizes.

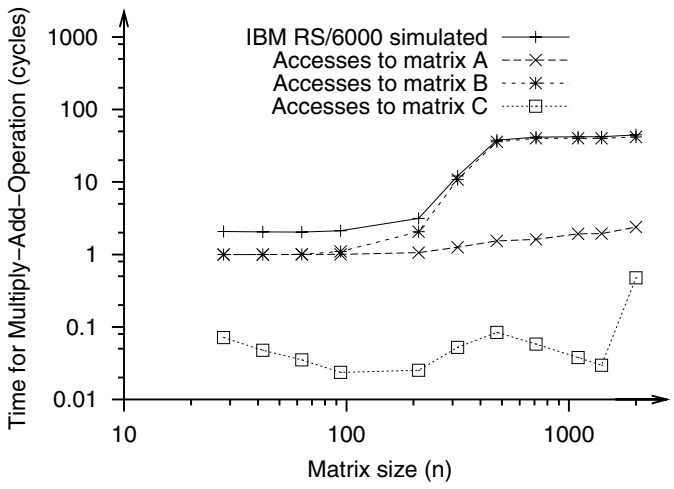


Fig. 6. Split-down of the execution times for each matrix involved in the multiplication.

4.2 Single Processor Results

Figure 5 shows the measured and simulated execution times of the matrix multiplication for different matrix sizes. In theory, we would expect an $O(n^3)$ execution time, corresponding to a horizontal line in Fig. 5. In practice, the actual growth rate is not cubical, because of the memory hierarchy.

As can be seen in Fig. 5, our simulation closely matches the real execution time. The prediction is slightly more optimistic, because the simulator does not include side effects of the operating system like task switching and interrupts. In real life these effects invalidate a part of the cache and thereby produce additional overhead.

Figure 6 gives a more detailed view. It shows the access time for each single matrix. This data has been determined by counting the accesses to matrix A, B, and C separately. As can be seen now, for the small matrix sizes, matrix A and B are both stored in the first level cache. When the matrices get bigger, matrix B falls out of the cache, while A is still in. This is caused by the layout of matrix B in the memory, which does not support spatial access locality.

The locality of matrix B can be improved by storing it in the transposed form, giving a similar access pattern for both source matrices, A and B. Each of them takes about half of the cache capacity. Figure 7 shows the simulation results for this improved memory layout.

Another interesting observation in Fig. 6 is the fact that the accesses to matrix C seem to become faster with increasing matrix size. This is due to the increasing number of computations that are done in the inner loop before storing a result element in matrix C. A sudden jump in the graph occurs only

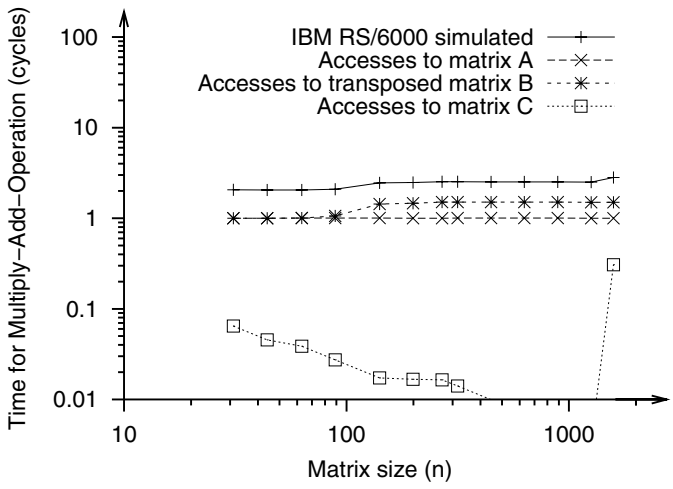


Fig. 7. Simulation of the optimized matrix multiplication with transposed matrix B.

when matrix C grows to such an extend that it cannot be completely contained in the cache.

4.3 Matrix Multiplication on an SMP

We have also simulated the matrix multiplication for a dual Pentium III processor system with 16 kB L1 and 256 kB L2 cache. Figure 8 gives the results of two versions of the algorithm, an alternating distribution where one processor computes all even and the other computes all odd matrix elements, and a blocked distribution where each processor computes a half consecutive block of matrix C. Clearly the blocked version is faster due to lower memory contention.

The memory contention has been simulated by including a simulation of the Pentium's MESI consistency protocol into the simulator. Note that this does not only simulate cache conflicts but also contention on the system bus.

5 Related Work

Several other simulators for caches exist, but they don't use the LDA model for their analysis and have another focus. Some of them are limited in the number of supported memory levels, others aim at simulating different hardware features.

The *Linux Memory Simulator (limes)* [2] has been designed for simulating SMP systems. It has been developed by modifying a compiler backend to generate instrumented code. This approach limits the flexibility of instrumentation and it also depends very much on the specific compiler. This execution-driven simulator is able to analyze parallel programs on a single host processor by creating threads for each target system.

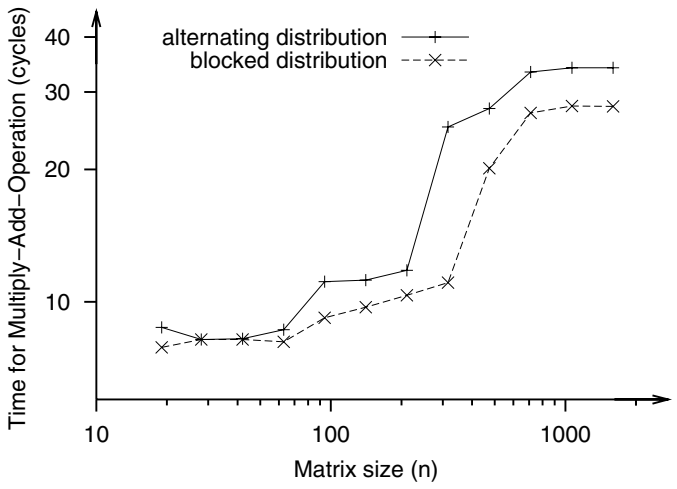


Fig. 8. Simulated execution times of the matrix multiplication on a dual processor SMP system.

The *Rice Simulator for ILP Multiprocessors (RSIM)* [6] has been designed with an emphasis on simulating instruction level parallelism. With this level of details, the internal processor behavior can be simulated more precisely, but at the cost of a higher simulation runtime. The *Wisconsin Wind Tunnel II* [4] in contrast tries to reduce the simulation time by executing instructions of the target machine in parallel on some host processors.

For analyzing new cache architectures the *Multi Lateral Cache Simulator (mlcache)* was developed [10]. It is used, for example to simulate cache that are split into temporal and spatial parts [7]. The depth of the memory hierarchy is limited to only a few levels.

6 Conclusion

We have presented a generic execution-driven simulator for the accurate prediction of memory accesses based on the *Latency-of-Data-Access Model* [9]. This model is motivated by the observation that the computation speed of modern computers is dominated by the access time to the various memory levels.

The simulator can be used to simulate a wide variety of target platforms. This is done with a configuration file that describes the memory characteristics in terms of the number of memory levels, cache sizes, access latency, and replacement strategy.

The simulator is triggered by the execution of the program, which must be instrumented manually before runtime. This allows the user to focus on the important code modules, e.g., the most time-consuming kernels. With the example of a matrix multiplication, we have shown how to use the simulator for deriving

better implementations for a given machine. The simulator indicates which data structure should be changed to optimize the code.

Moreover, the simulator is flexible enough to allow the simulation of complex cache coherency schemes that are used in SMPs. As an example, we have implemented the MESI protocol of the Pentium III. The results of our simulation of a dual processor Pentium lies within 10% of the measured execution time, thereby proving that the LDA model also properly reflects the impact of memory contention in SMP systems. In a similar way, the LDA simulator could be used to analyze different network infrastructure and protocols for distributed shared memory systems.

The simulator is open source under the GNU General Public License and is available at `http://www.zib.de/schintke/ldasim/`.

References

1. James R. Larus and and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 291–300, 1995.
2. Davor Magdic. Limes: A multiprocessor environment for PC Platforms. In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, March 1997.
3. Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference, January 22–26, 1996. San Diego, CA, USA*, pages 279–294, Berkeley, CA, USA, January 1996. USENIX.
4. Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, October/December 2000.
5. OMG. *Unified Modeling Language Specification.* Open Management Group, Version 1.3 edition, June 1999.
6. Vijay S. Pai, Parthasarathy Ranganathan, and Serita V. Adve. RSIM: An Execution-Driven Simulator for IPL-Based Shared-Memory Multiprocessors and Uniprocessors. In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, March 1997.
7. M. Prvulović, D. Marinov, Z. Dimitrijević, and V. Milutinović. Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance. In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, July 1999.
8. Florian Schintke. Ermittlung von Programmlaufzeiten anhand von Speicherzugriffen, Microbenchmarks und Simulation von Speicherhierarchien. Technical Report ZR-00-33, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), 2000.
9. Jens Simon and Jens-Michael Wierum. The Latency-of-Data-Access Model for Analyzing Parallel Computation. *Information Processing Letters*, 66(5):255–261, June 1998.
10. Edward Tam, Jude Rivers, Gary Tyson, and Edward S. Davidson. mlcache: A flexible multi-lateral cache simulator. Technical Report CSE-TR-363-98, Computer Science and Engineering, University of Michigan, May 1998.