MCCCSim – A Highly Configurable Multi Core Cache Contention Simulator

Michael Zwick, Marko Durkovic, Florian Obermeier, Walter Bamberger
and Klaus Diepold
Lehrstuhl für Datenverarbeitung
Technische Universität München
Arcisstr. 21, 80333 München, Germany
{zwick, durkovic, f.obermeier, walter.bamberger, kldi}@tum.de

ABSTRACT

Chip multi processors (CMP) are currently taking over the desktop processor market, supplying more computational power than former single core processors could ever achieve. However, a big increase in computational performance automatically widens the processor-memory gap, having the potential to slow down the whole system. Additionally, in multicore systems, the requirements of the memory hierarchy system get multiplied with the number of cores: While, on a system supporting only one thread at the same time, cache contention particularly introduces cache conflicts when switching from one thread to another at the end of a timeslice, in a multicore scenario, the caches are permanently accessed by several cores. This results in an increase of inter-thread conflict misses. Therefore, several techniques have been proposed to overcome inter-thread cache conflict misses and keep memory hierarchy accesstime low. One way to minimize cache interference in CMP systems is to allocate applications to cores in an optimal way, using either run time performance counter information or other prediction methods. In order to verify and refine such methods, a multicore CPU simulator reproducing parallel instruction execution has to be employed. In this paper, we present MCCCSim, an easy to use and highly configurable multicore cache contention simulator.

1. INTRODUCTION

Chip multiprocessors (CMPs) are supplying more computational power than single core processors could ever achieve. However, in multicore systems, also the processor-memory performance gap increases faster than ever before. Performance of multithreaded systems is particularly limited by the cache hierarchy and bus contention introduced by additional memory requests [2], [12]. Co-scheduled threads often compete with each other for the limited resources they share, the most important of which is cache memory. For SMT systems, Kihm and Connors observed that on average more than 30% of all data misses and 25% of all instruction misses on a randomly selected SPEC 2000 job mix are evoked by inter-thread conflict misses [3]. On CMP systems with many cores, things will be worse. Although cache interference is caused by the limited number of cache sets available in todays and tomorrows processors, increasing the number of sets is not considered to be an acceptable solution: Inter-thread conflict misses might be limited by an adapted cache architecture, but increasing the number of cache sets implies higher hardware cost and – due to additional lane and gate capacitance – also an increase in cache access time. Therefore, minimizing the amount of inter-thread conflicts and its impacts proves to be an effective and necessary technique [3], for which several methods have been proposed recently:

Dynamic cache partitioning exploits the fact that some processes suffer more from misses than others. G. Suh et al. proposed to use counters for each process to monitor L2 cache behavior at runtime. The counter values are used by the operating system scheduler to dynamically partition the cache amongst the executing processes, assigning more cache ways to applications that will benefit most by having additional ways. [13]

H. Dybdahl et al. proposed a cache-partitioning aware replacement policy based on an extra 'shadow tag' register and two counters for every processor cache line to partition the cache with cache line granularity [1], [9].

Liu et al. proposed to split the L2 cache into multiple small units that can be assigned to processor cores at runtime [4], [14].

Our research focuses on minimizing cache interference in CMP systems by merging those threads on the same memory hierarchy entity, that minimize cache contention. We developed a predictor that is based on the number of different cache sets, an application accesses during the execution of a number of instructions. To verify and refine our predictors, we developed *MCCCSim*, a highly configurable multi core cache simulator. MCCCSim consists of a framework that can be used to simulate custom CPUs with an arbitrary number of cores and a customized memory hierarchy, three examples of which are shown in figure 1.

The main difference of our simulator to most others lies in the easiness a broad range of CPU architectures can be simulated by simply supplying applicable command line argu-

ments, while the parametrization of other simulators is generally limited to more "static" parameters such as cache size, associativity, replacement policy, etc.

The *SESC (SuperESCalar Simulator)* [10] is an event driven simulator related to the MIPS processor architecture. Although this simulator is much more powerful than MCCCSim, providing a full out-of-order pipeline with branch prediction, buses, etc., these additional features do not provide any additional information since our traces are based on SPEC2006 benchmarks that were running on a real machine and the inaccuracy of memory traces introduced by the Pin toolchain may be neglected. Since in the SESC simulator, for performance reasons, many configuration options are chosen at compile-time rather than run-time, it would be necessary to compile multiple versions of SESC to simulate different architectures [10]. This does not represent a feasible choice for automated simulation of many different architectures.

Monchiero et al. [7] "use thread-level parallelsism in the software system and translate it into core-level parallelsism in the simulated world" [7]. Their work is based on the approach proposed by [6]: A full-system functional simulator dynamically generates an event trace that is fed into a timing simulator afterwards.

Tao et al. [14] developed *CASTOR*, a cache simulator for a comprehensive study of the multicore cache hierarchy and used *Valgrind* [8] to provide memory references. As with others, the parametrization of their simulator focuses on cache size, cache associativity, cache level and write policies [14]. For Valgrind, *Callgrind/Cachegrind* have been developed to record several cache performance metrics.

The remaining of this paper is organized as follows: Chapter 2 describes the simulator. Chapter 3 shows some simulation results that have been gathered with the simulator. Chapter 4 outlines performance constraints and chapter 5 concludes the paper.

2. MCCSIM

When designing the MCCSim tool, we decided to build a trace-driven simulator to be able to reproduce comprehensible results and make them comparable to one another, since memory address traces, once recoreded, do not change over time, as it might be the case in execution driven simulation. The traces are supplied by a pintool that comes along with the simulator and is a plugin for the *Pin binary instrumentation tool* [5] [11]. For every 1024 instructions, the supplied pintool generates an address *chunk*, i.e. a section in a binary file that holds the memory addresses that have been referenced. Chunks of several applications are fed into the MCCCSim simulator to obtain memory hierarchy related information such as memory system access time and hit rates of the caches in a multicore scenario.

2.1 Configuring the Simulator

Our Simulator is a command line utility written in C++ that evaluates its argument string to generate an arbitrary CPU architecture with respect to the number of cores, the number and configuration of caches and the cache-interconnection. For example, the invokation of our simulator in order to analyze co-scheduling behavior of an architecture as shown in figure 1 a) with SPEC 2006 benchmarks *milc* and *gcc* running timesliced on core 0 and *bzip2* running on core 1 would look as follows:

```
MCCCSim -l1 18-6-8_2_1.0 -l2 16-8-8_32_3.0 -m 100.0
-c0 l1-0_l2-0_m milc.trace gcc.trace
-c1 l1-1_l2-0_m bzip2.trace
```

This call has the following effect:

```
-l1 18-6-8_2_1.0
```

defines a cache object named "l1" with the number of bits to code (key, setnumber, byteoffset) = (18, 6, 8), an associativity of 2 and a hittime of 1.0 ns, which would result in a 32 kByte cache with a waysize of 16 kByte.

The most important difference to other simulators is that this cache object does not present the initialzation of "the L1 cache", but the initialization of an arbitrary cache object. If it gets an l1, an l2, an l3, ... object just depends on where it will be bound in the memory hierarchy.

```
-c0 l1-0_l2-0_m milc.trace gcc.trace
```

then creates a core named "c0" and adds a cache with the now defined configuration of l1 as first memory hierarchy element to that core. The postfix `-0` in `l1-0` enables the differentiation of several caches of the same type. Therefore, the l1 cache that gets connected to core 0 is not the same cache as the l1 cache that gets connected to core 1. However, since `l1-0` and `l1-1` both link to cache `l2-0`, this cache is shared by both `l1-0` and `l1-1`, as it is shown in figure 1 a).

Being able to build up completely different CPU architectures by simply supplying the corresponding command line arguments makes it easy to analyze a specific behavior over a broad range of architectures. Combined with an automated evaluation of the simulation results as we exemplarly developed for cache contention, MCCCSim turns out to be a powerful yet easy to use simulation tool.

2.2 Building up the Simulation Framework

As mentioned before, the simulator has to be called with an argument string that describes the cores, the CPU memory hierarchy, and the applications to run on each core. The argument string is processed by the simulator's front end, which generates and configures the CPU. As an example, we discuss the internal configuration setup for the argument string discussed in section 2.1, referencing the architecture depicted in figure 1 a).

First of all, the CPU and its cores get allocated:

```
CPU *cpu;
Core *core0 = new Core("c0");
Core *core1 = new Core("c1");
```

Then, the elements of the memory hierarchy are created:

```
Cache *l1_0 = new Cache("l1-0", 18-6-8_2_1.0);
Cache *l1_1 = new Cache("l1-1", 18-6-8_2_1.0);
Cache *l2 = new Cache("l2", "16-8-8_32_3.0");
Memory *m = new Memory(100.0);
```

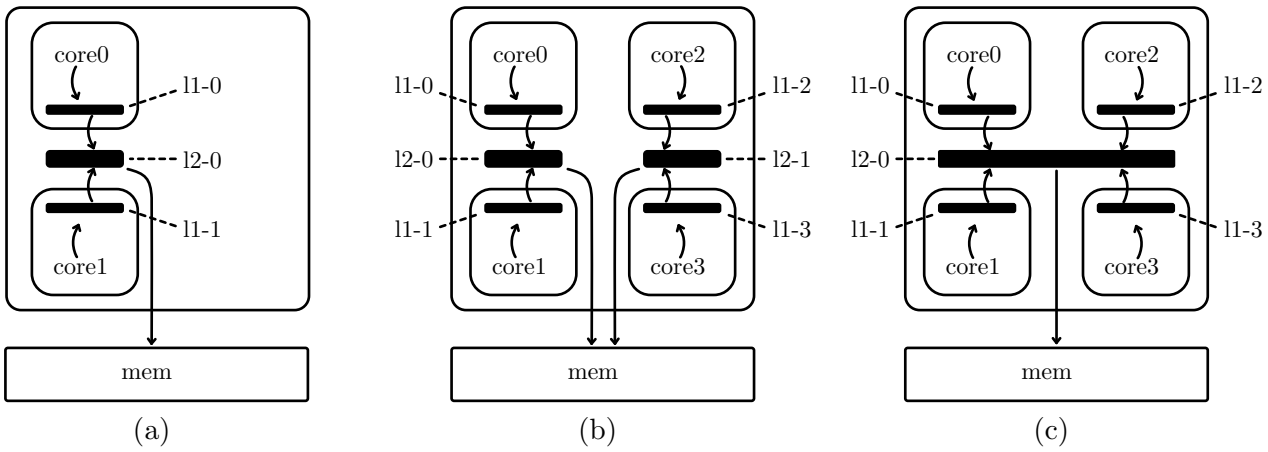

Figure 1: Examples of CPU-Architectures that can be simulated with MCCCSim. MCCCSim does not have any limitations regarding number of cores, hierarchies, size, etc.

To build up the hierarchy, the memory hierarchy elements have to be linked together. To achieve the behavior depicted in figure 1 a), `core0` has to be bound to the L1 cache `l1_0` and `core1` to `l1_1`. Since both L1 caches share the same L2 cache, both have to be bound to `l2`. To connect `l2` to the main memory, it has to be linked accordingly:

```
core0->mem = l1_0;
l1_0->mem = l2;

core1->mem = l1_1;
l1_1->mem = l2;

l2->mem = m;
```

To allocate an application to a specified core, the front-end calls the `addTracefile`-function for that core:

```
core0->addTracefile("milc.trace");
core0->addTracefile("gcc.trace");
core1->addTracefile("lbm.trace");
```

After adding the cores to the CPU, the frontend starts the simulation by calling `run`:

```
cpu->addCore(core0);
cpu->addCore(core1);
cpu->run();
```

Other architectures as the ones shown in figure 1 can be build up accordingly by allocating appropriate cores and memory hierarchy elements. The binding of the elements determines the behavior of the memory hierarchy.

A memory hierarchy representing the Intel smart cache system can be seen in figure 1 b). Applications on *core0* and *core1* cannot access memory in the *l2-1* cache, since there is no link to *l2-1*. In the architecture shown in figure 1 c) however, every application can displace frames of other applications in the l2 cache.

Dynamically generating the CPU configuration based on a parameter string enables the integration of the C++ simulator in script languages such as ruby or python, making it easy to provide automated simulation on a broad range of different configurations.

The integrated automated Analyser sorts the results for significance and, for convenience, automatically summarizes them in latex-tables as depicted in table 1.

2.3 Running the Simulator and Scheduling Applications

Applications that reside on different cores, are simulated to execute in parallel. Since the tracefiles supplied to the simulator do not contain any instructions, but only memory addresses that are separated by a separation mark representing 1024 instructions (one *chunk*), a synchronization mechanism has been provided. This way, applications with lesser memory references per chunk access the memory hierarchy less frequently than applications with a higher memory usage, which can be seen in figure 2. The execution of addresses in one chunk is linearly interpolated in time, synchronizing with the applications on the other cores to simulate parallelism. Figure 4 shows the corresponding algorithm used to determine the next core to be granted memory access according to our synchronization scheme.

If more than one application has been added to a core, then the applications are scheduled with an adjustable time slice value, as can be seen from figure 3.

3. RESULTS

In this section, we present some results gathered by our simulator. Figure 5 depicts the L2 miss rate degradation due to conflict misses with applications sharing the same L2 cache in the architecture depicted in figure 1 a). One can easily see that co-scheduling *milc* with *astar* or *gcc* achieves much lower cache contention than co-scheduling *milc* with *gobmk* or *lbm*.

Table 1 shows the effect of co-scheduling applications on the l2 cache memory for some more SPEC2006 benchmarks,

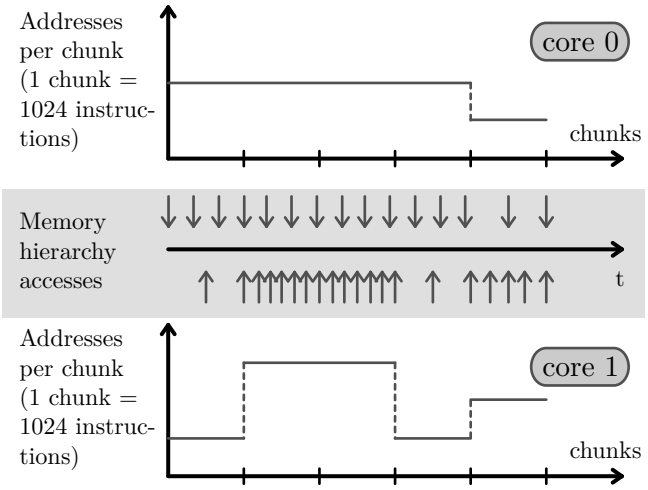

Figure 2: Accessing the memory hierarchy according to the number of memory references per instruction.

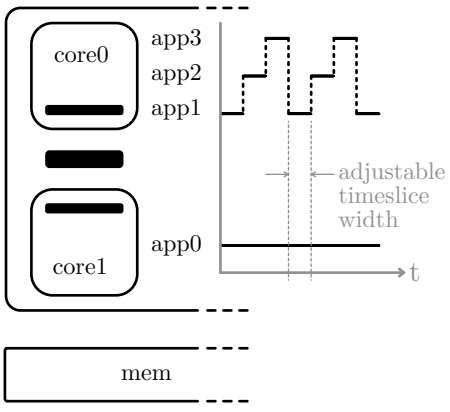

Figure 3: Scheduling.

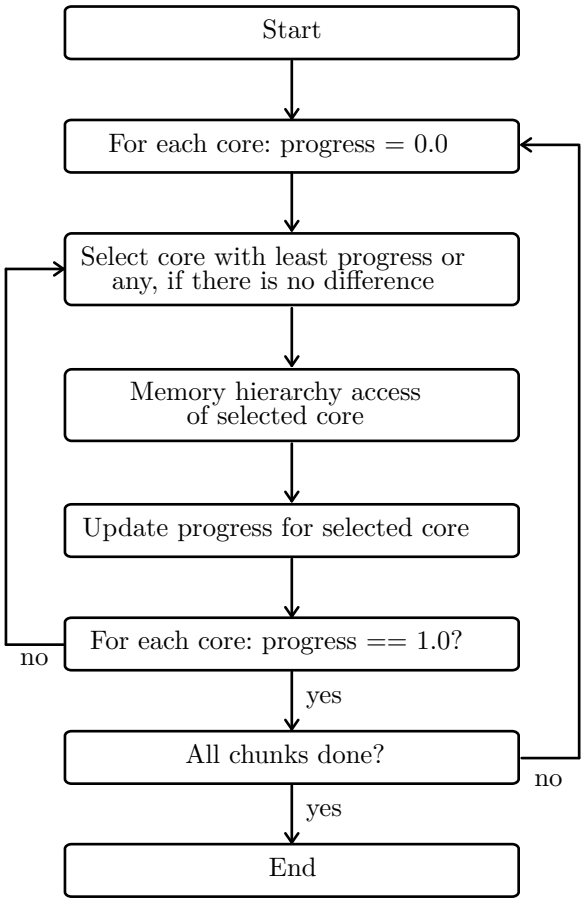

Figure 4: Core Synchronisation Method to simulate parallel execution of applications.

sorted by significance regarding the value in the column p_{add}. This value represents the additional penalty due to L2 cache contention and is measured by summing up the differences of the standalone hitrate ($0 \leq h \leq 1$) and the co-scheduled hitrate ($0 \leq h \leq 1$) of each chunkset (1 chunkset = 1024 chunks = 2^{20} instructions). The first column shows the application affected by the performance degradation, the second and third column the core on which this and the co-scheduled application resides on.

Row one of the table shows, that co-scheduling the SPEC2006 benchmarks *milc* and *lbm* results in the highest cache contention among all other combinations. The introduced access time penalty for this combination is depicted in figure 6.

4. PERFORMANCE

Figure 7 shows the performance of our simulator executed on a 2.4 GHz Intel Core 2 Duo MacBook Pro with 4 GB of 1067 MHz DDR3 RAM, running the OSX 10.5.6 Leopard operating system. The Performance is depicted with subject to the number of cores simulated in a scenario, where each core has its own L1 cache and every L1 cache accesses the same L2 cache. Averaged on the depicted SPEC 2006 benchmarks, a per core performance of about $121 + 4.8 \times cores$ ms to process an address chunk (2^{20} instructions) can be observed. Figure 8 shows, that for a senario with a separate L2 cache

application	app on core 1	app on core 0	p_{add}
milc	lbm	milc	126.77
hmmer	hmmer	lbm	94.39
mcf	lbm	mcf	85.02
milc	gobmk	milc	63.86
mcf	gobmk	mcf	55.35
mcf	mcf	milc	47.79
mcf	bzip2	mcf	37.29
astar	astar	lbm	36.54
milc	bzip2	milc	35.81
gcc	gcc	lbm	34.83
astar	astar	milc	32.03
milc	gcc	milc	30.12
milc	astar	milc	28.47
bzip2	bzip2	lbm	27.53
mcf	gcc	mcf	23.77
astar	astar	gobmk	21.26
hmmer	gobmk	hmmer	18.25
h264ref	h264ref	lbm	16.71
mcf	astar	mcf	16.44
povray	lbm	povray	15.56
astar	astar	bzip2	15.07
astar	astar	gcc	14.07
gcc	gcc	milc	13.99
gcc	bzip2	gcc	13.49
gcc	gcc	gobmk	11.30
bzip2	bzip2	gobmk	10.76
...	...	...	...
mcf	h264ref	mcf	3.25
povray	milc	povray	3.24
bzip2	astar	bzip2	3.08
astar	astar	h264ref	2.87
hmmer	bzip2	hmmer	2.55
bzip2	bzip2	povray	2.13
astar	astar	hmmer	1.88
gobmk	gobmk	lbm	1.66
bzip2	bzip2	mcf	1.52
astar	astar	povray	1.44
...	...	...	...
hmmer	hmmer	mcf	0.19
gobmk	gobmk	mcf	0.17
gcc	gcc	hmmer	0.15
hmmer	h264ref	hmmer	0.14
hmmer	hmmer	povray	0.14
h264ref	h264ref	hmmer	0.13
astar	astar	libquantum	0.06
gobmk	gobmk	hmmer	0.06
povray	hmmer	povray	0.06
gobmk	gobmk	h264ref	0.05
bzip2	bzip2	libquantum	0.02
h264ref	h264ref	libquantum	0.01
lbm	astar	lbm	0.00
libquantum	astar	libquantum	0.00
lbm	bzip2	lbm	0.00
lbm	gcc	lbm	0.0
gcc	gcc	libquantum	0.00
...	...	...	...

Table 1: Additional L2 penalty due to conflict misses.

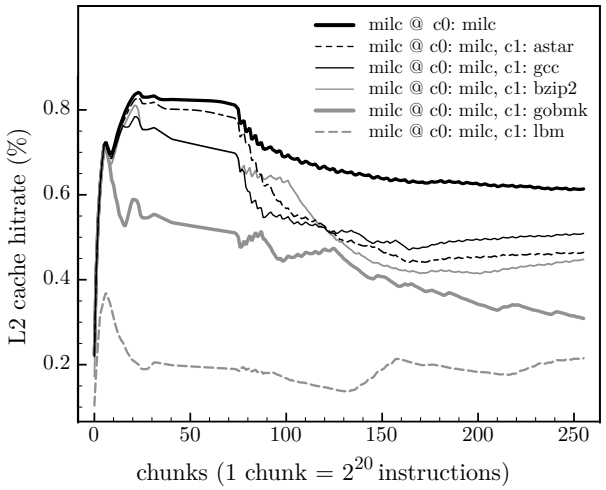

Figure 5: L2 cache misses when co-scheduling *milc* with the SPEC2006 benchmarks *lbm*, *gobmk*, *bzip2*, *gcc* and *astar*.

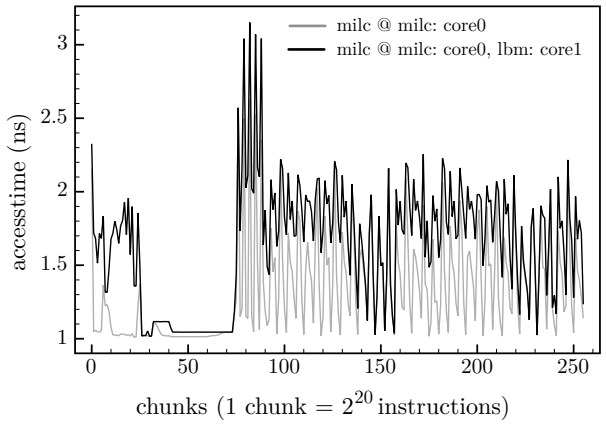

Figure 6: Memory access time running *milc* standalone and with *lbm*, running on another core.

for every 4 cores, about the same average performance per core is achieved. This means that the performance is nearly independent of the memory hierarchy design: Supposed a L1 hitrate of about 97 %, only three out of one hundred memory accesses cause an L2 tag RAM search and L3 tags are searched about 100 times more rarely. Therefore, simulation time is about $121 + 4.8 \times cores$ ms per address chunk and core, i.e. about $114 + 4.8 \times cores$ ns per instruction.

5. CONCLUSIONS

In this paper we presented MCCCSim, a simple and easy to use simulator to analyse cache contention. In contrast to other tools, the cpu architecture MCCCSim simulates is defined by the argument string supplied to the binary. There is no need to recompile the source or manipulate ini-files to change the simulated CPU architecture, making it easy to simulate over a broad range of different architectures. Simulation time scales linearly with the number of addresses to simulate, nearly independent of the number of cores introduced in the system.

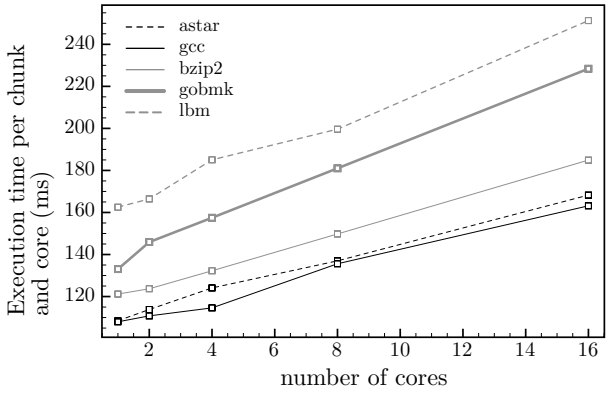

Figure 7: Performance in *ms* to process a chunk (2^{20} instructions), divided by the numbers of cores. In this scenario, every core has its own L1 cache and all L1 caches are bound to a single L2 cache.

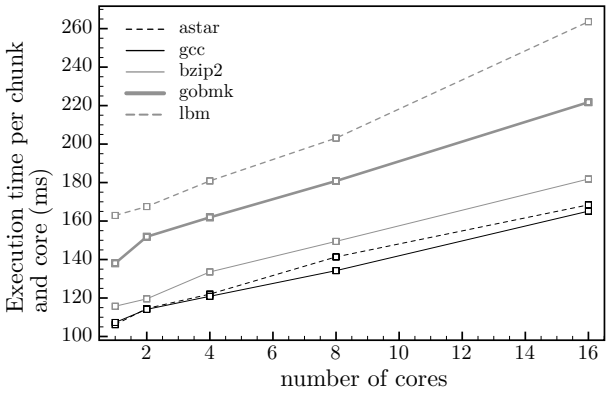

Figure 8: Performance in *ms* to process a chunk (2^{20} instructions), divided by the numbers of cores. In this scenario, there's a different L2 cache for every 4 cores.

The simulator is free software, licensed under GPL and can be downloaded from http://www.ldv.ei.tum.de/research/areas/downloads/MCCCSim.zip.

6. REFERENCES

[1] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Lecture Notes in Computer Science 4297, Springer-Verlag, Berlin*, 2006.

[2] S. Hily and A. Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, Institutional De Recherche En Informatique Et En Automatique, 1997.

[3] J. L. Kihm and D. A. Connors. Implementation of fine-grained cache monitoring for improved smt scheduling. In *Proceedings of the 22nd IEEE International Conference on Computer Design*, October 2004.

[4] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'04)*, 2004.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klausner, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation, ACM*, 2005.

[6] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 2002.

[7] M. Monchiero, J. H. Ahn, A. Falcon, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. In *Workshop on Design, Architecture and Simulation of Chip Multiprocessors (dasCMP'08)*, 2008.

[8] N. Nethercote and J. Seward. *Valgrind: a program supervision framework.* Elsevier Science B. V., http://www.elsevier.nl/locate/entcs/volume89.html, 2003.

[9] K. Nikas. *An Analysis of Cache Partitioning Techniques for Chip Multiprocessor Systems.* PhD thesis, University of Manchester, School of Computer Science, 2008.

[10] P. M. Ortego and P. Sack. Sesc: Superescalar simulator, 2004.

[11] V. J. Reddi, A. Settle, D. A. Connors, and R. Cohn. Pin: A binary instrumentation tool for computer architecture research and education. 2004.

[12] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced smt job scheduling. *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques*, September 2004.

[13] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. In *The Journal of Supercomputing.* Kluwer Acedemic Publishers, 2004.

[14] J. Tao, M. Kunze, and W. Karl. Evaluating the cache architecture of multicore processors. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2008.