Using Fast and Accurate Simulation to Explore Hardware/Software Trade-offs in the Multi-Core Era

W. Heirman
T.E. Carlson
S. Sarkar
P. Ghysels
W. Vanroose
L. Eeckhout

Report 08.2011.2, August 2011

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

Using Fast and Accurate Simulation to Explore Hardware/Software Trade-offs in the Multi-Core Era

Wim HEIRMAN [a,c,1], Trevor E. CARLSON [a,c] Souradip SARKAR [a,c]
Pieter GHYSELS [b,c] Wim VANROOSE [b] Lieven EECKHOUT [a]

[a] *Ghent University, Belgium*
[b] *University of Antwerp, Belgium*
[c] *Intel Exascience Lab, Belgium*

Abstract. Writing well-performing parallel programs is challenging in the multi-core processor era. In addition to achieving good per-thread performance, which in itself is a balancing act between instruction-level parallelism, pipeline effects and good memory performance, multi-threaded programs complicate matters even further. These programs require synchronization, and are affected by the interactions between threads through sharing of both processor resources and the cache hierarchy.

At the Intel Exascience Lab, we are developing an architectural simulator called *Sniper* for simulating future exascale-era multi-core processors. Its goal is twofold: Sniper should assist hardware designers to make design decisions, while simultaneously providing software designers with a tool to gain insight into the behavior of their algorithms and allow for optimization. By taking architectural features into account, our simulator can provide more insight into parallel programs than what can be obtained from existing performance analysis tools. This unique combination of hardware simulator and software performance analysis tool makes Sniper a useful tool for a simultaneous exploration of the hardware and software design space for future high-performance multi-core systems.

Keywords. Architectural simulation, performance analysis, software optimization

1. Introduction

Programmers typically rely on a range of tools that provide insight into the behavior of their programs, ranging from lightweight profiling to heavy-weight detailed processor simulation. During application development on existing hardware, there are a number of established tools that enable software developers to view their application's characteristics. A few examples are GNU gprof, or Linux's perf tool and VTune from Intel which use hardware performance counters. For exploring software trade-offs with next-generation hardware designs, one needs to rely on either simulation tools or profiling. Profiling and simulation tools all come with their own trade-offs in accuracy, speed and coverage; in addition, profiling tools may be quite intrusive, changing the application's

[1]Corresponding Author: Wim Heirman, ELIS Department, Ghent University, Belgium; E-mail: wim.heirman@ugent.be.

original behavior. Coverage determines what aspects affecting the application's performance can actually be analyzed. For example, a cache simulator such as CacheGrind (part of the Valgrind suite [1]) reports cache hit rates, but can only guess at their effect on timing. When evaluating future hardware, or when co-designing the hardware to match a given algorithm, a hardware simulator is necessary. Traditionally, hardware simulators tailor results for hardware developers who are evaluating potential hardware improvements. They provide insight into why the *hardware* is performing as it is and assumes that the software will not be not modified. It therefore will not help in optimizing *software* performance — either for the given architecture or for an architecture that evolves together with the application. Yet, for meeting both the performance and energy requirements of future exascale HPC systems, most experts agree that application-driven co-design of hardware and software will become a necessity [2].

To this end, we, at the Intel Exascience Lab, are developing Sniper [3], a hardware-validated, fast and accurate parallel architectural simulator. Sniper builds upon the infrastructure provided by the Graphite multicore simulator [4] and uses the interval processor core simulation model [5]. This combination results in a fast, yet accurate hardware simulator for modeling current and future x86 multicore processor systems. We also integrated the McPAT power simulator [6] to obtain power and energy estimates.

For hardware developers, a fast and accurate architectural simulator is indispensible for evaluating the performance of a wide range of benchmarks over a huge design space of next-generation architectures. For a software developer, the advantage of Sniper is that it is much faster than most other detailed architectural simulators, and that it adds a number of simulation postprocessing tools that are aimed explicitly at software performance analysis. Compared to most profiling frameworks the advantage is that Sniper is not intrusive — the application's notion of both time and hardware are completely simulated in software, so simulation overhead does not affect scheduling or synchronization decisions, nor does it use up cache lines and other processor resources that it shares with the application. Since Sniper is modeling all hardware components, e.g., caches, branch prediction, instruction pipeline, etc., the effect on execution time for each of these components can be determined reliably. Moreover, since instruction dependencies and their out-of-order execution are also taken into account, it is possible to determine which latency contributions can be overlapped with the execution of other instructions — and thus hidden from the total application run time.

The goal of this paper is to illustrate the power of the Sniper simulation infrastructure for software performance analysis. To this end, we describe a number of analysis techniques that Sniper provides to help evaluate and understand application performance. We then demonstrate their usage in a case study with one of the applications used at the Exascience Lab.

2. Understanding parallel performance

Understanding parallel performance is challenging for the various reasons mentioned in the introduction. As a result, it is impossible, or at least very hard, to visualize parallel performance in an intuitive and insightful way using a single representation. In this paper, we present the two performance representations, namely CPI stacks and thread-state timelines, which we have integrated in Sniper and which we found invaluable for analyzing parallel workload performance. We first introduce these two concepts here in this section; the next section then uses these representations in a case study.

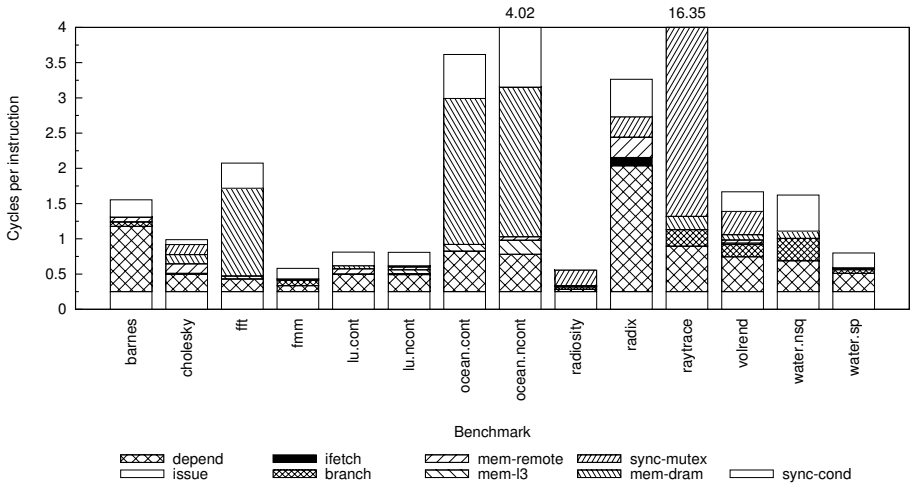

Figure 1. Breakdown of the average cycles per instruction (CPI) taken to execute each of the SPLASH-2 applications on a 16-core machine.

2.1. CPI stacks

Computer system performance can be expressed by counting the number of instructions that can be executed per clock cycle (IPC), or the inverse, the average number of cycles one instruction takes (CPI). A CPI number can be broken down into several components, each contributing a given amount of latency that makes up a program's total execution time. Figure 1 shows such a *CPI stack* for each of the applications in the SPLASH-2 benchmark suite [7]. The CPI breakdown is shown for a single thread out of the 16 threads running on our simulated reference machine. (We observed similar behavior for the other threads because of the homogeneous workload.) CPI stacks are extremely valuable for both hardware architects and software developers; a CPI stack provides insight into the performance bottlenecks and 'lost' cycles.

2.2. Synchronization

Synchronization is a complex matter when optimizing parallel workloads. A thread-state timeline is an insightful representation of a parallel workload's dynamic behavior, as illustrated in Figure 2. Time runs from left to right, threads are shown above one another. Blue (the lowest position for each thread) represents normal computation. Brown (middle position) occurs when a thread is blocked on a mutex or condition variable, while red (topmost position) means that the thread is computing while holding a mutex. Using this type of graph, an algorithm designer can clearly see when threads are computing, when they are synchronizing, and when load imbalance occurs. The application in this case in the heat transfer simulation introduced in Section 3. This benchmark simulates the transfer of heat over a 2-D grid over a number of time steps, where each thread is responsible for a spatial partition of the grid. On the thread-state timeline, different phases can be distinguished which relate to the algorithm's alternation between local computation (blue) and barrier synchronization between time steps (brown). Some amount of load imbalance becomes apparent — even though all threads execute the same amount

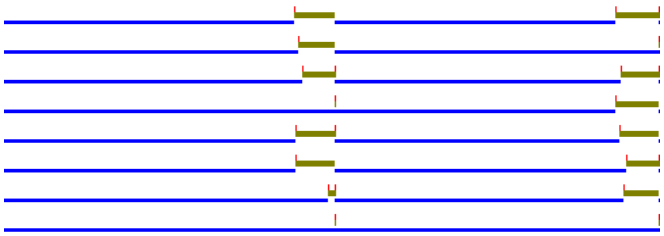

Figure 2. Time trace (left to right) for the synchronization between eight threads running a heat transfer simulation.

of work, their memory behavior is different: some threads experience more cache misses than other threads, which results in differences in relative progress amongst the threads.

Traces such as these can also be obtained by libraries that intercept calls to the pthread functions, such as *mutrace*. These are however not without overhead, which can affect timing and thus synchronization behavior. Also, the inclusion of synchronization primitives in our CPI stack gives a global understanding of where time is spent, and what the impact of synchronization delay is in light of the total execution time.

3. Case study: Heat transfer simulation

In this case study we look at an evaluation benchmark which models heat transfer across a 2-D grid during a number of time steps. The heat transfer equation involves a stencil computation in which the temperature at a given moment in time at each grid location is a linear combination of the temperatures of that location and its neighbors at the previous time step. A naive implementation of the heat transfer application computes a single time step at a time, and iterates over the complete grid to apply the stencil operation to each data point. This implementation has very bad data reuse, however: the data for each grid location is used only once per time step. The next time this data is used again – in the subsequent time step – the processor has touched all other data points so (if one is simulating a large grid) the data values corresponding to the first grid location are long since flushed from the caches. Improving locality in this algorithm would allow for a better usage of the cache, which can have tremendous benefits in both execution time and energy consumption.

To increase the locality of this application, the computation is restructured to compute multiple time steps on a small tile of the complete grid. This allows the tile's data points to be reused multiple times, before moving on to the next tile. A disadvantage is that at the edges of a tile, data from a neighboring tile about future time steps is needed – which are not yet computed. This can be solved by overlapping the edges of the tiles, and let the overlap decrease (by the width of the stencil) at each iteration. This way the computation evolves – when time is represented as a dimension perpendicular to the grid plane – in a pyramid-shaped fashion [8].

Of course, the overlap means that some computation is duplicated. Trading off the amount of this redundant computation against the increase in data locality is the main challenge in optimizing the heat transfer algorithm. This problem is difficult to solve using any of the existing tools, as a full understanding entails several aspects. The redundant computation wastes energy, but can allow for faster execution. The exact execution

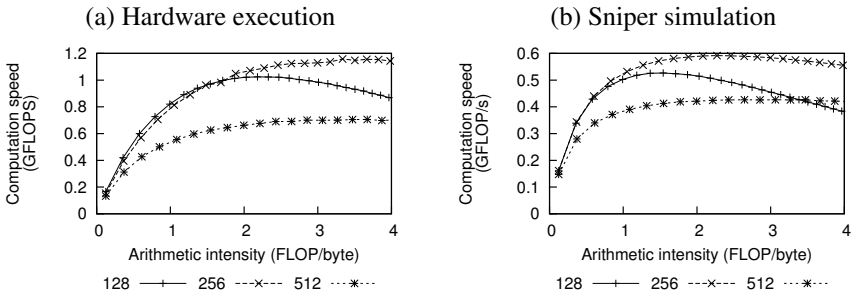

Figure 3. Useful computational rate versus arithmetic intensity for a heat transfer simulation, as run on real hardware (a), or modeled using Sniper (b).

time is also very dependent on the extent to which main memory access latencies can be overlapped effectively.

3.1. Performance analysis

Figure 3(a) is one representation used by the software designers in our group. In this experiment, the heat transfer application is run on a single core of our reference machine.[2] The grid domain is 4096×4096 elements. This domain is split up into square tiles measuring either 128, 256 or 512 data points on each side. The number of time steps performed on each tile before moving on to the next one varies, from 1 up to 65 steps. The graph plots the rate of useful computation — this is the number of non-redundant floating-point operations that are performed — in billions of floating-point operations per second. The horizontal axis denotes the *arithmetic intensity* which is defined as the number of floating-point operations the program performs per byte that is loaded from memory (including caches). This measure is important as it expresses the algorithms' ability to reuse data, which improves when more time steps are computed per tile. One can clearly see how performance improves when increasing the number of time steps, and later falls back once the amount of redundant computation becomes too high.

To better understand this trade-off, and to visualize the root causes of it, we repeated this experiment by running the same application in our Sniper simulator, configured to model the architecture of our reference machine, and extracted several simulation statistics. Firstly, Figure 3(b) repeats the computation rate versus arithmetic intensity graph. Although due to various approximations Sniper's prediction is not 100% accurate, what is important is that the relative performance of the different versions of the algorithm, as predicted by the simulator, tracks actual performance relatively accurately. This allows for correct conclusions to be drawn as to which parameter combination is optimal in light of several hardware and software optimizations, which leads to correct design decisions.

In Figure 4, we plot two run-time performance characteristics, obtained from our Sniper simulations, which point to the root causes of the observed performance. Figure 4(a) plots the fraction of redundant computation. A larger arithmetic intensity, caused

[2] Our reference machine is a quad-socket Intel Xeon X7460 Dunnington server. Each processor chip contains six 45-nm Core (Penryn) based cores running at 2.66 GHz, and includes 3 MB of L2 cache per two cores, and a 24 MB L3 cache shared by all six cores. We also model relevant parts of the Intel 7300 Clarksboro chipset, as well as main memory. In this configuration, we can predict application runtimes for our reference machine with an average absolute error of 23.8%, while simulating up to 2.0 MIPS on an 8-core host machine [3].

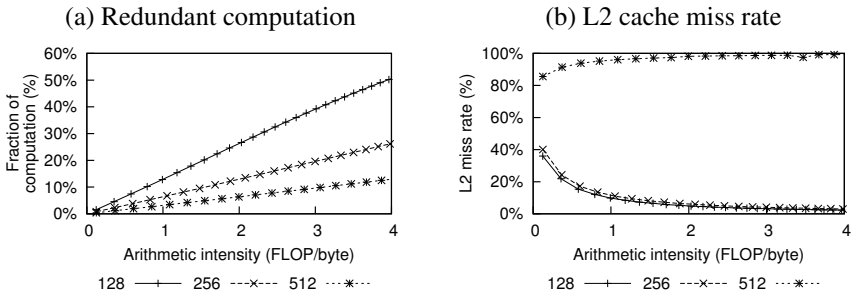

Figure 4. Runtime characteristics of the heat transfer simulation

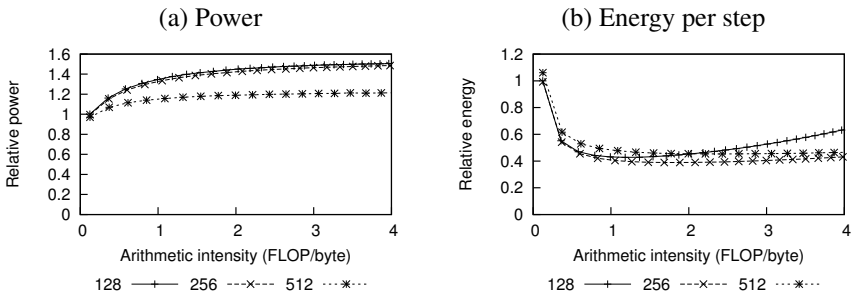

Figure 5. Power and energy usage of the heat transfer simulation, as estimated by McPAT.

by a larger number of time steps being run independently on each single tile, causes a linear increase in duplicated work. Moving to larger tile sizes mitigates this effect, as the size of the edge of the tile, which determines the amount of overlap, becomes less important compared to the total area of the tile.

In Figure 4(b) one can clearly see why the version with tiles of size 512 performs so much worse, even though the amount of redundant work is smaller. A tile of 512×512 double-precision floating-point elements takes up 2 MB, as two copies are used in a double-buffering approach, to save both the current and the previous time steps, this makes the working set of the application a total of 4 MB. Since the second-level cache of the Xeon processor model we used (the X7460) is only 3 MB, the 512-sized version of the algorithm can therefore not use the faster second-level cache effectively and thus has to work out of the slower third-level cache. Also, it is evident from the graph that versions of the algorithm with lower arithmetic intensity have lower data-reuse, which again leads to a higher cache miss rate.

3.2. Power and energy estimation

Sniper extracts several key architectural characteristics, such as the number of instructions executed of each class, the number of accesses to hardware structures such as caches, etc. These run-time characteristics are combined with architectural parameters and sent to the McPAT power simulation tool [6] which estimates power usage while running the application that was simulated.

Figure 5 shows the results for the heat transfer simulation. In Figure 5(a), power usage is plotted relative to a non-tiled approach. It can be seen to go up slightly when

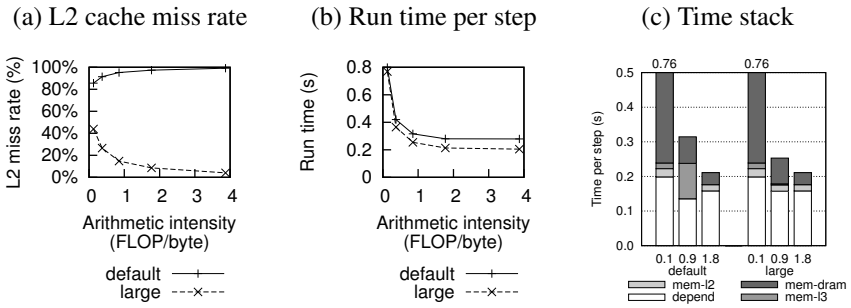

Figure 6. Runtime characteristics of the heat transfer simulation using a tile size of 512 points after doubling the L2 and L3 cache sizes.

increasing the arithmetic intensity — this is because the processor core has a higher activity factor since it is waiting less for memory operations — but quickly levels off. The energy expended per time step can be calculated as well, and is shown in Figure 5(b). These graphs clearly show that, although power usage is higher, the versions of the algorithm that add redundant computation are still able to save energy. When the energy cost of computation versus that of off-chip communication changes, as is expected to happen in future processor generations, the curves on these graphs would shift allowing software designers to fine tune their algorithms accordingly.

3.3. Architectural exploration

Architectural simulation is an interesting tool in obtaining more insight into the behavior of a running application. A graph such as the L2 miss rate (Figure 4, b) can be obtained by using hardware performance counters. On the other hand, the effect of cache misses on run time, which depend on whether the misses can be overlapped by out-of-order execution, cannot be determined using existing performance counters — yet it can be readily determined by analyzing the CPI stacks generated by Sniper.

The real strength of architectural simulation lies in the simulation of future architectures, and in co-designing hardware and software. On the real machine, a 512×512 point tile does not fit in the L2 cache. Figure 6 shows what would happen on a machine with larger caches. We configured Sniper to double the L2 and L3 cache sizes compared to our reference machine and simulated the heat application again using a tile size of 512 points. Figure 6(a) shows that the L2 miss rate goes down significantly, as could be expected. From the new per-step run times in Figure 6(b) though it looks like the large caches do not help much to decrease the overall run time. The breakdown of Figure 6(c) shows why. The time spent waiting for data coming from the L3 cache (mem-l3), which is significant in the cases with the default cache configuration, is replaced by a much smaller component of L2-related latency. But while the runs with low arithmetic intensity (the 0.1 FLOP/byte cases) are dominated by memory latency, the runs with high arithmetic intensity (0.9 and 1.8 FLOP/byte cases) are dominated by instruction dependencies. This clearly shows that a study purely based on cache miss rates would lead to an incorrect result, as the extent to which cache latencies can be overlapped is just as important in the determination of run time as the miss rates themselves.

4. Conclusions

Using Sniper, a hardware-validated architectural simulator for multicore processors, we were able to gain more insight into the behavior of parallel programs. Sniper's combination of a fast yet sufficiently detailed processor simulation model and its collection of software analysis tools presents a unique environment for performing simultaneous optimization of both hardware architecture and software algorithms. In a case study, we analyzed the behavior of a heat transfer simulation program, and root-caused the performance characteristics that lead to the trade-off between various parameters for this algorithm. Moreover, since Sniper is able to feed the timing of memory accesses back into its model of the processor core, we were able to obtain an accurate representation of how cache characteristics affect program performance and energy consumption through various overlap effects, something which is not possible when using a pure cache simulator.

Acknowledgements

We thank the reviewers for their constructive and insightful feedback. Wim Heirman, Trevor Carlson, Souradip Sarkar and Pieter Ghysels are supported by the ExaScience Lab, which is supported by Intel and the Flemish agency for Innovation by Science and Technology (IWT). Additional support is provided by the FWO projects G.0255.08, and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

References

[1] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2007, pp. 89–100.

[2] J. Dongarra *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 2022, 2011.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[4] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2010, pp. 1–12.

[5] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2010, pp. 307–318.

[6] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[7] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *International Symposium on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[8] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, pp. 129–159, 2009.