Sniper: Simulation-Based Instruction-Level Statistics for Optimizing Software on Future Architectures

Wim Heirman Alexander Isaev Ibrahim Hur

Intel Corporation

ABSTRACT

In this paper we address the problem of optimizing applications for future hardware platforms. By using simulation—traditionally a tool for hardware architects—applications, libraries and compilers can be optimized before hardware is available, allowing new machines to start doing useful scientific work more quickly. However, traditional processor simulators are not very user-friendly and are, due to their extreme level of detail, too slow to run applications with large input sets or allow for interactive use. In contrast, the Sniper many-core simulator uses higher abstraction level models, trading off some accuracy for a much higher simulation speed. By adding instrumentation into the simulator that can annotate performance information at fine granularity, down to individual instructions, it becomes a valuable tool for software optimization on future architectures.

1. SHIFT-LEFT OF SOFTWARE DEVELOPMENT

Performance projections of future systems are crucial for both software developers and processor architects. Developers of applications, runtime libraries and compilers need predictions for tuning their software before the actual systems are available, and architects need them for architecture exploration and design optimization. Simulation is one of the most commonly used methods for performance prediction, and developing detailed simulators constitutes a major part of processor design. Traditionally, simulators were only used in the exploration and design phases of product development. This means software development and optimization have to wait until (prototype) hardware becomes available, see Figure 1(a). This puts the software development effort on the critical path towards bringing products to market (from the point of view of the vendor), or delays the point at which new machines can start running optimized science codes (for the HPC user).

(a) Traditional flow

(b) Enabling early software development

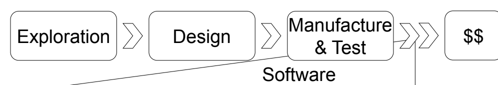

Figure 1: Shift-left of software development enables quicker time-to-market.

Recently, much effort has been put into enabling software developers to start work early; at least before final hardware is available, and ideally to make application optimization part of the hardware exploration process—enabling a true co-design of hardware and software where both can be optimized in combination (Figure 1, b). The lack of available hardware requires early software development and optimization to be done using some form of performance simulation. Creating detailed, usually cycle-accurate simulators is part of the hardware development and validation effort. However, most detailed simulators, while very accurate, are too slow to simulate meaningful parts of applications—especially in the context of many-core systems with large caches. Instead, these simulators typically run short traces of code and require great care and often manual effort to both select these traces and provide adequate warmup of structures with long-living state such as caches and branch predictors.

2. HIGH-LEVEL SIMULATION

By trading off some accuracy for the ability to run larger parts of the application, higher abstraction level simulation can play a valuable role in both software tuning and architecture exploration. Simulation speed can be increased by not modeling some hardware components that are known to be a bottleneck (e.g., instruction caches in many HPC codes), or by moving away from structural models that try to model exactly what each hardware component is doing and instead using analytical models such as interval simulation [4] or instruction-window centric models [2] for the processor core, or queuing theory for on-chip networks.

Sniper [1] is an x86 many-core simulator that combines many of these techniques, in addition to being built on a parallel simulation framework which can make use of modern multi-core hardware. These properties result in an acceptable

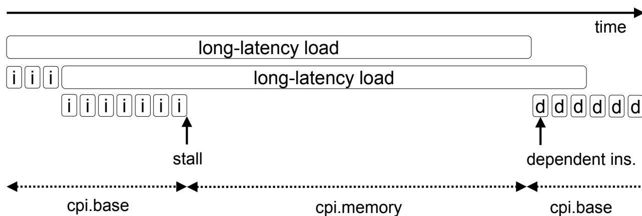

Figure 2: Overlapped execution of cache misses and independent instructions on out-of-order processors.

accuracy (around 20% average absolute error compared to Nehalem hardware) but much improved simulation speed (around 1 MIPS, which is around 1000× faster than typical industrial detailed simulators). This brings interactive (overnight) runs of representative parts of an application within reach, greatly speeding up the optimization cycle.

3. ACCURATE PERFORMANCE METRICS

On real hardware, many performance counters are available that can give valuable insight into how codes are behaving. Cache miss rates are especially valuable, as these often indicate long pauses in the execution of instructions by the processor leading to low performance (typically expressed in instructions per clock cycle, IPC). In the simulator, the behavior of structures such as caches is modeled in detail so extracting statistics such as hit rates is trivial.

However, the use of miss rates as indicators for application performance can be misleading, as indicated in Figure 2 which plots the execution timeline of a typical section of code when running on modern hardware. Load operations that miss in the processor caches usually take many tens or even hundreds of clock cycles, whereas loads that hit in cache or compute instructions take only a handful of cycles. One could therefore assume that the length of time taken to execute a section of code is proportional to the number of instructions, increased by the number of cache misses multiplied by the typical latency of a cache miss. But this does not take into account the fact that out-of-order processors can continue executing independent instructions, including potentially other long-latency loads, while waiting for the original cache miss to be serviced.

To alleviate this problem, hardware architects often employ the concept of the CPI stack [3]. This is a stacked bar graph which breaks up an application's execution time into a number of components, and is normalized to cycles per instruction (for a CPI stack) or to the total number of clock cycles (for a cycle stack). Each component in the stack denotes the *penalty* caused by a different hardware component, taking into account the fact that many miss events may overlap. In the case of the execution shown in Figure 2, all time spent executing compute instructions is accounted for in the *base* component (denoting the execution time assuming the processor would be capable of reaching its maximum performance all the time) while only the stall time, when no instructions other than the cache misses are in progress, is accounted for in the memory penalty component. This way, each clock cycle of execution is assigned to that hardware component that was on the critical path of execution.

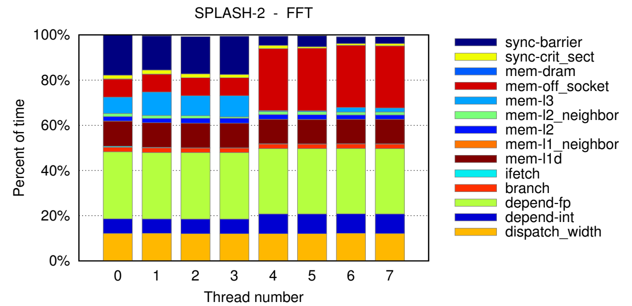

Figure 3: Normalized cycle stacks for each core executing the `fft` benchmark when running the small input set on eight cores.

Solving a given stall that is visible on the cycle stack will therefore be guaranteed to lead to increased performance. In contrast, ignoring the fact that much of the cache miss latency is overlapped would overestimate its effect, potentially leading programmers to spend time to reduce cache misses or other miss events that are not performance critical.

While it is only very recently becoming possible to measure CPI stack components using hardware performance counters [6], they are natively supported in the Sniper simulator [5]. Figure 3 plots an example CPI stack obtained from running an FFT workload on a simulated dual-socket, eight-core Nehalem machine (with one software thread pinned to each core), and illustrates some interesting performance effects. Comparing the behavior of threads 0—3 with that of threads 4—7, one can see that the first four threads spend around 20% of their time in the *sync-barrier* component, denoting they were stalled in a software barrier. This behavior may be surprising as all threads perform the same amount of work. Looking at the other components, it becomes clear that the difference in execution speed can be explained by non-uniform memory access (NUMA) behavior as all cores operate on data that is available in the first socket's level-3 cache to which the first four cores have faster access: cores 0—3 have some amount of *mem-l3* and only little *mem-off_socket* time denoting mostly local L3 accesses, while cores 4—7 have a significant *mem-off_socket* penalty.

4. FINE-GRAINED STATISTICS

To increase insight into the behavior of different parts of the code, we extended an internal version of Sniper to collect hardware events and timing effects at a per-instruction granularity. As in the whole-program case, comparable statistics can in some cases be obtained on existing systems using hardware performance counters, but these suffer from a number of drawbacks: many hardware counters have inaccuracies such as double-counting under certain conditions, skidding (meaning that events are not always associated with the correct instruction), sampling errors (instruction pointers are typically only sampled when a counter overflows), or a lack of insight into how hardware events contribute to execution time. In contrast, our instruction-level statistics are based on the concept of cycle stacks and can assign an execution time cost to each individual instruction. Event counts are added as well to aid in understanding what hardware component causes the time penalty.

```
eip   instruction                                    cycles   ops  mask  dram
--
4050b vpcmpd k2{k1}, zmm5, zmm4, 0x2                  1.03%
40510 vmovupd zmm8, zmmword ptr [r11+r10*1]           3.51%
40517 vaddpd zmm7, zmmword ptr [r11+r14*1]            7.24%   8.0         6.93
4051e vfmadd231pd zmm8{k2}, zmm7, zmm6                1.84%  12.6   21%
40524 vmovupd zmmword ptr [r11+r10*1]{k2}, zmm8       1.03%
```

Figure 4: Per-instruction statistics.

Figure 4 provides an example for a snippet of AVX-512 code. The third instruction (`vaddpd`) goes out to DRAM (it performs 6.93 DRAM accesses per 1,000 executions) and hence has a high performance impact (it is responsible for 7.24% of total execution time). For HPC workloads it is often important to distinguish instructions that contribute to useful work (floating point operations) from those that manage data and control flow (loads and stores, address and loop index calculations, comparisons and branches, etc.). To this end the *ops* column plots the number of FP operations executed by each instruction, taking into account masked elements: the fourth instruction (`vfmadd231pd`) is a fused multiply-add which performs two operations on each of eight vector elements, but on average 21% of the elements are masked off leading to a useful operation count for this instruction of 12.6 double-precision operations on average.

5. DATA-CENTRIC STATISTICS

Large data-parallel workloads are often limited by cache capacity and memory bandwidth. While gaining insight into which instructions cause cache misses can help in tracking down those data structures that are responsible for poor cache use, often it can be more insightful to be able to look at individual data structures directly. To this end, Sniper can collect cache statistics on a per data structure basis. In a simulator, implementing such functionality is relatively straightforward: application calls to *malloc* and other memory allocation library functions are intercepted, and the address ranges for each data type (determined by the call stack leading up to the *malloc* call) are recorded. Each memory access made by the core can then be tagged with its allocation site and cache statistics are accumulated per site.

An example can be seen in Figure 5. Two allocation sites are detected in the `fft` benchmark, corresponding to the *trans* and x variables in the source code. Whereas *trans* has good cache behavior and can be serviced mostly out of the L1 cache, x experiences many cache misses—and could be a candidate for moving into high-bandwidth memory, or algorithmic optimizations such as blocking.

```
Site #1:  Location:      main     fft.c:251  { trans = malloc(...); }
          Hit-where:     Loads           :      1433601  ( 14.3%)
                           L1            :      1384287  ( 96.6%)
                           L2            :        43560  (  3.0%)
                           dram          :         5754  (  0.4%)
          Total allocated: 2.0MB (2.0MB average)
Site #2:  Location:      main     fft.c:250  { x = malloc(...); }
          Hit-where:     Loads           :      1433601  ( 14.3%)
                           L1            :      1026277  ( 71.6%)
                           L2            :       326618  ( 22.8%)
                           dram          :        80706  (  5.6%)
          Total allocated: 2.0MB (2.0MB average)
```

Figure 5: Per-array statistics for `fft`.

6. REFERENCES

[1] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, Nov. 2011.

[2] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28:1–28:25, Aug. 2014.

[3] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A top-down approach to architecting CPI component performance counters. *IEEE Micro*, 27(1):84–93, 2007.

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

[5] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 38–49, Nov. 2011.

[6] A. Yasin. A top-down method for performance analysis and counters architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, Mar. 2014.