Sequential Performance: Raising Awareness of the Gory Details

Erven Rohou, David Guyon

Sequential Performance: Raising Awareness of the Gory Details

Erven Rohou[1] and David Guyon[1]

Inria, Campus de Beaulieu, Rennes, France
first.last@inria.fr

Abstract
The advent of multicore and manycore processors, including GPUs, in the customer market encouraged developers to focus on extraction of parallelism. While it is certainly true that parallelism can deliver performance boosts, parallelization is also a very complex and error-prone task, and any applications are still dominated by sequential sections. Micro-architectures have become extremely complex, and they usually do a very good job at executing fast a given sequence of instructions. When they occasionally fail, however, the penalty is severe. Pathological behaviors often have their roots in very low-level details of the micro-architecture, hardly available to the programmer. We argue that the impact of these low-level features on performance has been overlooked, often relegated to experts. We show that a few metrics can be easily defined to help assess the overall performance of an application, and quickly diagnose a problem. Finally, we illustrate our claim with a simple prototype, along with use cases.

Keywords: sequential performance, microarchitecture

1 Introduction

The complexity of computing system increases at a fast pace, and this trend is likely to continue in the foreseeable future. Roadmaps [3, 9, 10] predict thousands of cores on a chip by the end of this decade, and several vendors have already released chips with a number of cores in the hundreds (Intel, Kalray, Tilera). Complex processors go along with complex memory hierarchies, interconnects, etc. The increasing share of parallel systems in the consumer market (initially dual cores, and quad cores, now 12 cores, Xeon Phi, etc.) makes parallelism appealing to all developers, not only those interested in HPC.

Extracting parallelism is intrinsically complex, but the speedup achievements can be easily assessed: optimized sequential code gives a reference, and the number of cores gives a target, assuming linear scalability. Sequential code (or sequential sections of an application) are more difficult to assess. In particular, there is no easily defined performance target. Nevertheless, the performance of sequential sections is of utmost importance, even for parallel applications. As per Amdahl's law [2], it ultimately limits the overall performance of the system.

Due to the complexity and heterogeneity of platforms, and to the lifetime of software, developers just cannot fully exploit the hardware features of the platforms where their applications run. Intricate interactions between hardware, operating system, compiler, and application may result in major performance issues. In some cases, the performance also depends on input data. New trends such as cloud computing make things even worse by sharing resources between applications in a non deterministic ways, impossible to reproduce and very difficult to comprehend. The actual performance of an application is thus known only at run time. In presence of parallelism, the slowdown incurred by under-performing code can surpass the gain brought by parallel execution. Resolving performance issues is often the business of experts, and it is beyond the scope of this paper. However being aware that something goes wrong is the first step that yields to proper investigation of the reasons behind.

The contributions of this paper are the following: 1) we show that very low-level micro-architectural events can impact the performance in dramatic ways, sometimes more that what parallelism delivers on current machines; 2) we propose as a proof-of-concept a simple tool to illustrate that users can be made aware of such pathological behaviors.

Section 2 further motivates our approach and illustrates it with micro-architectural behaviors that have a severe impact on performance. We present our proof-of-concept in Section 3, including chosen metrics, and implementation, and we apply it to selected use-cases in Section 4. Section 5 overviews related work. We conclude in Section 6.

2 Motivation

To estimate how well their applications run, many UNIX users typically rely first on commands such as *ps* or *top* and look at the column labeled `%CPU` for the corresponding processes. Any number significantly lower than 100 % indicates a problem that must be investigated: resource conflicts (e.g. more processes than hardware threads), slow I/O, virtual memory effects, etc. However, when the CPU usage is close to 100 %, users can only conclude that there is no visible reason to be concerned.

For better performance, programmers then turn to parallelization. Current general purpose machines, however, feature up to a dozen cores, and more parallelism requires clusters of machines, where the cost of communications must be carefully balanced with the increase in computational power. In all cases, only a fraction of an application can be parallelized, and scaling is usually "sub-linear", yielding diminishing returns.

We argue that in some cases, parallelization may be a premature effort. CPU usage only tells the user one part of the story: how often their processes are scheduled for execution by the operating system. It does not say anything about the way execution proceeds. Modern micro-architectures are extremely complex, and the performance of applications can be severely impacted by many factors, intervening at several moments in the lifetime on an application, from compile time to run time.

The roofline model [25] is a very straightforward way to give users a visual understanding of how the bandwidth is exploited. The model plots raw performance (flops/s) versus operational intensity (flops/bytes). The plot visualizes the performance bottlenecks of the processor (CPU and memory bandwidth) and indicates what is limiting the performance of a given application.

Beyond memory bandwidth, the performance of modern processors can be limited by many other microarchitectural features. We develop a few examples below.

- Most compilers have the ability to auto-vectorize loops, an optimization that consists in recognizing when an operation is repeatedly applied to consecutive elements of an array, and exploiting dedicated SIMD instructions to process them in parallel. Unfortunately,

SIMD extensions are very diverse, and even a single vendor can provide several families (e.g. Intel SSE, SSE2, SSE3, SSSE3, SSE4.x, AVX...) When running old legacy code, or when targeting unknown machines (as in the cloud), applications turn out to be under-optimized for recent hardware. Moreover, the SSE and AVX introduce additional constraints. Due to very low-level design details, mixing them results in severe penalty, typically 75 cycles [16]. A compiler is unlikely to produce such a situation, however, it may happen with hand-written code or when linking with legacy libraries.

- Memory is slow compared to CPUs. To bridge the gap, architects designed hierarchies of cache memories. Performance now depends on the spatial and temporal locality of memory accesses. The time to access a data spans up to two orders of magnitude, from a few cycles for the first cache level, up to 100 cycles for local memory, and even several hundreds in case of a remote access on a NUMA machine [4]. For memory bound applications, this translates into an order of magnitude slowdown. Achieving the best memory throughput is a daunting task for programmers and compilers. The roofline model has been recently made cache-aware [14] to help programmers quickly visualize the behavior of the memory hierarchy.

- High clock frequencies can be achieved thanks to deep execution pipelines. Branch instructions create a disruption in the sequence of addresses, that may cause a penalty. To limit this penalty, branch predictors attempt to guess the target of a branch, and let the processors speculatively execute code. In case of misprediction, the processor must stall execution, flush the pipeline, and resume execution at the newly computed address. This accounts for a few dozens cycles. Indirect branch instructions, in particular, have been reported [11] to be extremely difficult to predict when used in the context of interpreters.

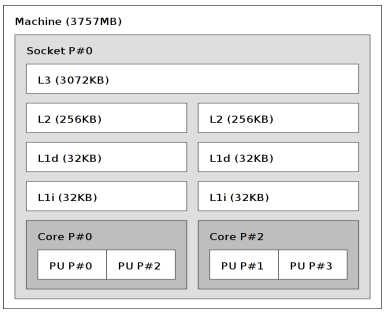
- High-end processors have dedicated hardware for floating point computations. Some Intel processors however handle denormal numbers [13] thanks to a sequence of micro-operations. It is described as "extremely slow compared to regular FP execution" in the Intel manual [16]. We observed that a simple computation on the x86 architecture can perform up to 87× worse than expected for pathological parameters, because of micro-operations [23]. A real world application (illustrated in Section 4) suffers a 17× slowdown.

This list is clearly not exhaustive. However, it shows that performance issues originate from many sources, and they can be dramatic. Despite the fact that these slowdowns are very significant, our experience is that they are usually unnoticed, simply because there is no available easy-to-use tool to report them. We advocate that developers should be made aware of these performance issues on sequential code. Fixing these performance bottlenecks may surpass the speedups obtained by parallelization efforts, and the corresponding effort may be worthwhile.

3 Proof of Concept

As a proof of concept, we leveraged the existing *hwloc* software, and we augmented it to dynamically display information collected from low-level hardware performance counters.

In this section, we first present *hwloc*, followed by the characteristics of the performance monitoring unit available in modern processors. We then present the collected data, and the metrics we derive. Finally, we briefly describe our implementation, and discuss possible remote execution on compute nodes.



Intel Core i3 M350 processor (Westmere micro-architecture). Two cores are present, with Hyper-threading enabled. Each physical core has dedicated L1 instruction and data cache (32 KB each), and a 256 KB L2 cache. A L3 cache of size 3 MB is shared by the cores.

Figure 1: Snapshot of original *lstopo* showing cores and cache hierarchy

3.1 hwloc

The Portable Hardware Locality (*hwloc*) software package [5] provides a portable abstraction (across OS versions, architectures...) of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multithreading. It also gathers various system attributes such as cache and memory information. It primarily aims at helping applications with gathering information about modern computing hardware so as to exploit it accordingly and efficiently. *Hwloc* provides a static graphical interface representing the hardware architecture thanks to its tool *lstopo*. We base our development on the last version of *hwloc* at the time of writing (version 1.9). See Figure 1 for an example.

3.2 Performance Monitoring Unit

Virtually every processor in use today embeds a Performance Monitoring Unit (PMU), consisting of hardware dedicated to counting many architectural and micro-architectural events, such as instructions, cycles, cache accesses and misses, mispredicted branches, etc. The PMU is entirely a hardware mechanism. Counting events has virtually no impact on the behavior and performance of running applications (as opposed to instrumentation, for example).

Recent versions of Linux provide a system call (`perf_event_open`) to facilitate the access to the PMU. The most relevant parameters are the event ID, the process ID to monitor, and the CPU ID to observe. It is also possible to collect data for all processes on a given CPU,or a given process on any CPU. When successful, the system call returns a file descriptor, from which the values of the counter can be directly read.

3.3 Collected Data

The Linux system call can be used to monitor any event provided by the micro-architecture. As a first approach, we focus on the following events: cycles the process or core was in execution; retired instructions; accesses to each cache level, and corresponding misses.

When Hyper-threading is enabled, a physical core is made of several logical cores. We instantiate counters for each of them. To handle shared caches (in our example L1 and L2 across logical cores, and L3 across all physical cores), we consolidate the cache events seen from different logical cores to obtain the total number of events seen by each shared cache. Thanks to the API provided by *hwloc*, we automatically capture the topology of the hardware.

Despite the large number of events, only a limited number of counters is available. We rely on multiplexing when we need more events than the actual number of counters available in the processor. Multiplexing is natively available in the Linux system call.

Beyond hardware events, we also report the CPU load. It is calculated using the information from the `/proc` file system as the ratio of the execution time over the total running time (including the idle time) The detail of this calculation depends on the running mode: either machine-wide or monitoring of a specific process.

Machine-wide In machine-wide mode, our tool scans the whole system for performance data. We gather information from the file `/proc/stat`, in particular the time spent in the *user*, *nice*, *system* and *idle* modes for each logical core. We compute the CPU load for each as follows:

$$load = 100 \times \frac{user + nice + system}{user + nice + system + idle}$$

Per PID The CPU load is calculated only for the selected process, considering all its threads. It uses the values from all `/proc/<pid>/task/<tid>/stat`. We add the *utime*, *stime*, *cutime* and *cstime* values of each thread to obtain the total time the process was in execution during the past time period. The elapsed time is obtained from the Linux real-time clock. We derive the CPU load as follows:

$$load = 100 \times \frac{\sum utime + \sum stime + \sum cutime + \sum cstime}{elapsed}$$

3.4 Metrics

Raw data collected from the counters, such as the number of instructions or cycles, is of limited use. To make them more amenable to quick understanding, we build a few metrics.

The simplest aggregated metrics is probably IPC, i.e. executed instructions per cycle. There are many pitfalls related to IPC, as it may not be a direct proxy for performance. As an example, when generating code, a compiler cannot be evaluated (only) by the IPC of the generated code: what really matters is the number of cycles. Poor code generation might add useless instructions that artificially inflate the IPC without making the program run any faster. However, from the point of view of a user given a program executable, the number of instructions (I) to execute to completion is fixed. Hence the higher the IPC, the lower the number of cycles (C), the better. IPC captures many reasons for low performance, including the events described in Section 2.

Poor data locality is a typical performance bottleneck, as it hinders the mechanisms of the memory hierarchy. Popular metrics include the cache miss rate (fraction of all cache accesses that are misses), and the MPKI (number of misses per kilo-instruction). The former characterizes the cache in isolation, quantifying its ability to handle requests. But it fails to discriminate between situations where both the number of accesses and the number of misses change in the same proportion. A high miss rate is acceptable as long as the absolute number of misses remains low. MPKI addresses this issue by relating the number of misses to the number of instructions, hence directly quantifying the impact on performance. We opted for a two-dimensional representation: the number of cache accesses represents the demand on the cache, and the miss rate its effectiveness.

By definition, the value of a miss rate is between 0 and 100 %. The maximum number of accesses, however, depends of the particular hardware. To obtain an upper bound, we developed micro-benchmarks that stress each level of the hierarchy. This is done by walking through an array such that each cell contains the index of the next cell to visit. The permutation is precomputed in a way to minimize locality (and maximize the number of misses). Other approaches exist, such as Ofenbeck et al. [21] derived from the roofline mode.

Other popular metrics (such a Flops, or Mips) can be easily added.

3.5 Implementation

As a proof of concept, we integrate dynamic performance data in the static graphical interface of *lstopo*. We purposely applied very limited changes to the graphical interface to maximize code reuse. In practice, we restricted to adding color boxes to represent metrics of interest. We used the *hwloc API* to get information about the processor architecture: number of sockets and the number of logical cores per socket.

The first change makes *lstopo* dynamic, thanks to the POSIX *select* method linked to the graphical user interface event listener and a timer. If the listener triggers first, the handler of user interactions is executed. When the timer fires, the update methods are executed. The refresh period is configurable (by default, 1 s). At each window refresh, *lstopo* calls the update methods which read the counters and save the value globally, so the user interface can access them to update the window.

We represent the CPU load, as a new box of variable height inside each PU box. The height is proportional to the load, ranging from a barely visible box at the bottom when the system is unloaded, to a fully filled solid PU box when the load is 100 %.

IPC is represented by the intensity of the background color of the CPU load boxes. It varies from red to yellow. A solid red box means that the value of IPC is close to zero. Yellow represents a high IPC value. A minor problem lies in the fact that a low IPC is rather easy to identify, but the maximum IPC *realistically* achievable on a given machine is much more difficult to determine. We decided to saturate the value at a user defined threshold, derived from the peak performance of the architecture. Anything above this value is reported in the same color.

The graphical representation of caches is also augmented with a variable height box. The height represents the number of accesses, i.e. the demand for data. The intensity of the background color represents the miss rate. A completely dark blue box means indicates 100 % miss rate.

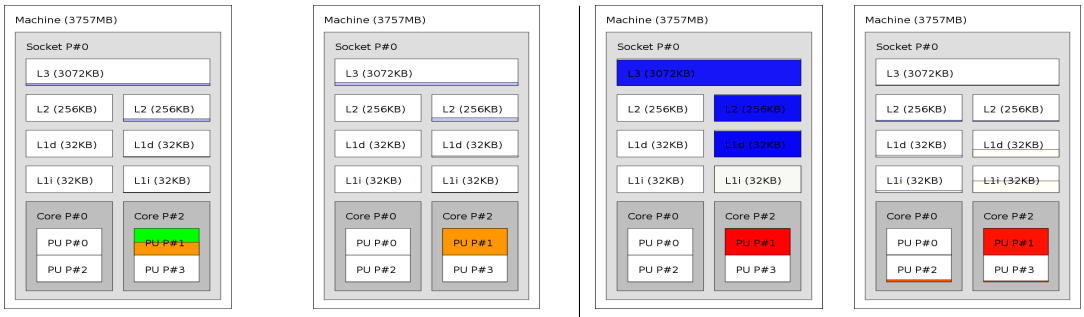
3.6 Remote Execution

Our prototype development is clearly driven by 1) the choice to apply minimal changes to *lstopo*, and 2) the desire to report graphically real-time performance characteristics. However, in case X Window System (X11) is not available on the compute nodes, we consider emitting the information in log files, and processing them on the host (*lstopo* already has the capability to dump information in various formats, including text and XML).

Job schedulers may provide interactive access to users having jobs running on a node. When this is not the case, it would still be possible for the system administrator to run a light-weight daemon on execution nodes, in charge on collecting data and conveying it to a graphical rendering application on the front-end.

4 Use Cases

The goal of our prototype is to help users first identify that *something* is going wrong with their application, and second obtain a rough idea of the kind of pathological behavior they are experiencing. This section illustrates a few use-cases. It is by far not meant to be exhaustive. Rather, the purpose is to show that even a few simple, graphical representations can raise awareness of low level mechanism that are typically hidden from users. In all these cases, with the exception of the Figure 2 (a), tools such as *top* report a CPU usage close to 100 %.



(a) CPU load: $\ll 100\,\%$ (b) CPU load: $\sim 100\,\%$ Graphical representation of CPU load and IPC

(c) cache effect (d) no cache effect Diagnosing Low IPC

Figure 2: Graphical representation of CPU load and IPC

4.1 Symptoms of Low Performance

The first symptom of a pathological behavior is a low CPU load. This may have many causes: inherent to the application or due to external factors. An application may perform heavy I/O operations, or depend on many thread synchronizations, causing idle time. It may also be impacted by other applications, running on the same physical or logical core, and competing for resources, such as functional units, or shared cache. These cases can be observed by standard utilities, such as *top* or *ps*. We also report this phenomenon, as illustrated on Figure 2 (a): logical cores P#0, P#2, and P#3 are idle. Only core P#1 has a workload, but the CPU load (yellow/orange box) is only 49 %. The core P#1 has a green background because the process in pinned to this core (as defined by the original *lstopo*). The reason for low CPU load is another process running on the same core.

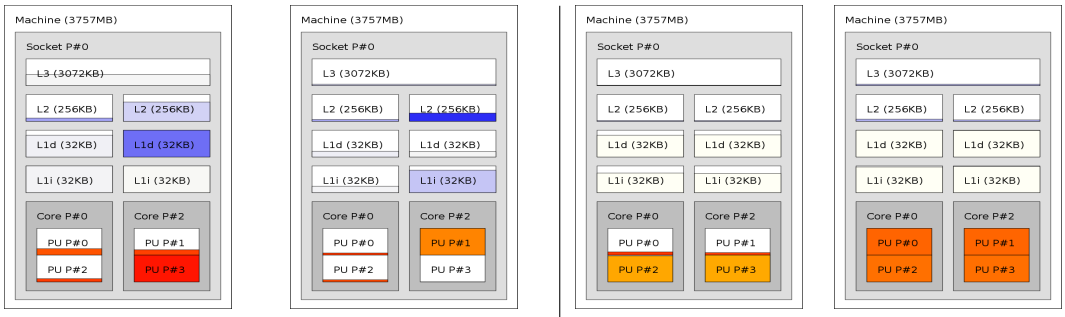
Conversely, in Figure 2 (b), core P#1 is loaded at 100 %. IPC is measured at 1.42 and reported in orange. It may be possible to improve it, but nothing major impacts performance.

Figure 2 (right images) illustrates two cases where applications experience a very low IPC (reported in red). Even though they both eventually result in degraded performance, the reasons are very different. In Figure 2 (c), the application performs highly irregular accesses to an array much larger than the L3 cache. The data caches, drawn in solid dark blue, are heavily solicited, and have extremely poor behavior. Abnormal miss rates are 97 %, 95 % and 91 % respectively for L1D, L2, and L3. Since a memory access (deriving from a L3 miss) costs at least hundred cycles, the running application is clearly memory bound. The IPC is 0.01.

Figure 2 (d) illustrates the behavior of an application solving RAM (Range-dependent Acoustic Model) parabolic equations. All caches behave well, but again the performance is extremely low (on average 0.2 instruction per cycle). It turns out that the reason lies in the floating point computations using denormal numbers (see also Section 2). Intel processors also provide counters to track these pathological events. We have not implemented this yet, and this information could not be derived from the graphical interface (although it helped exclude memory related problems). However, this is a very relevant information for the user, and we plan to add it in future work. The same applies to branch prediction, or mix of SSE and AVX instructions.

4.2 Caches

As seen on Figure 2 (c), some workloads have an extensive memory footprint that severely impacts the entire cache hierarchy. Smaller footprint can impact only the lower levels. This is



(a) L1 data cache (b) L1 instruction cache
Behavior of L1 caches

(c) 2 threads (d) 4 threads
Impact of parallelism on IPC

Figure 3: Behavior of L1 caches and impact of parallelism

the case of the application depicted in Figure 3 (a). Its workload exceeds the capacity of L1, under high demand, and suffering a high 80 % misses. The second level achieves 14 % miss rate. Consequently, L3 receives less demand. Its miss rate is also much better, at 2 %. The resulting IPC is about 0.4.

The instruction cache is less likely to show a poor behavior, due to the regularity of addresses in sequences of instruction. This behavior can anyway be captured. As an illustration, we developed a micro-benchmark for this purpose. It consists in a gigantic switch statement, much larger than the instruction cache size (32 KB), within an infinite loop that enters the `case` entries randomly. We are able to increase the miss rate up to 24 %. Figure 3 (b) illustrates the result. Though unrealistic, the benchmark shows our capability to identify unusual instruction cache behavior. Such phenomena may happen in the presence of very large code bases, with very dynamic and unpredictable control flow.

4.3 OpenMP Parallelism Degree

OpenMP is an API for parallel programming. It targets shared memory, multithreaded systems, and applications written in C, C++ and Fortran. The API consists in compiler directives as well as a runtime library. The directives let a programmer instruct the compiler that sections of code can be run in parallel. In particular, known parallel loops can be easily designated to the compiler.

During execution, the runtime allocates threads and distributes tasks (such as chunks of the loop iteration space). The number of created threads depends on several factors, including the number of cores, the load of the machine, and values of environment variables.

We experimented with a CPU intensive loop and a dual core processor with Hyper-threading enabled (hence four logical core). The loop computes successive values of Riemann ζ function, calling `libm`'s `pow()` function millions of times. Figure 3 (c) and (d) illustrates the behavior of the application with respectively two and four threads. When two threads are used, the operating system adequately allocates one per physical core, and achieves 1.92 IPC on each core. The program completes in 28.5 seconds. With four threads, the IPC of each is reduced to 1.25, as shown by the orange color of the cores. The program completes in 22.2 seconds. Running four threads is clearly more efficient in this example. Still, the speedup is about 1.28×, far from the expected 2×.

5 Related Work

Performance monitoring counters have recently attracted a lot of interest. Most modern processors now provide support to collect data in hardware, and many tools exist to help collect the data [24]. The number of available countable events, and counters vary greatly across architectures [17]. Moore [18] compares efficiency, accuracy and bias of each method in the PAPI library [6]. PAPI requires manual intervention to insert probes in the source code. We advocate a high-level and synthetic view of the behavior of an application or the whole system, that provides a programmer with a immediate understanding of the overall performance.

Other work [12, 19, 20] study how the insertion of probes impacts the execution of the monitored application. Our approach avoids this problem altogether by not modifying the application or its execution environment.

Some tools integrate the access to the performance counters with a graphical interface. Intel offers the commercial VTune Amplifier [15] performance analyzer, which samples the execution based on hardware or operating system events and combines the results with other analyses to provide tuning advice. Similar to our approach, VTune does not require recompilation. But its purpose is to relate performance issues to subroutines and locations in the source code to assist programmers with performance tuning. Our focus is presenting overall metrics at a glance to the user. HPCToolkit [1] provides graphical post-mortem analysis of parallel programs. It focuses on attributing bottlenecks to code locations, and estimating parallel scalability. In contrast, we focus on sequential performance, and restrict ourselves to an overall view of performance. Our tool is hence complementary to HPCToolkit. PerfExpert [7] builds on top of HPCToolkit. It collects performance counter measurements, computes various metrics, and suggests steps to improve performance. ThreadSpotter [22] is an integrated development environment that lets programmers quickly identify and locate performance issues related to data locality, cache utilization, and thread interaction.

Perf [8] and tiptop [23] are command-line utilities that display the values of selected hardware counters. The former reports aggregated values for the entire run. The latter periodically displays the values, much in the way *top* does. Both require that specific metrics be defined and encoded in the tool. In addition, they do not provide any graphical interface, which is the key for instant understanding that a problem occurs.

As mentioned, micro-architectures have become extremely complex. Many compiler optimizations focus on various features (such as caches, branch prediction...) Many manuals and books have covered performance optimization at length, see for example Intel's 600+ pages Optimization Reference Manual [16]. Our goal is to focus on simple, high-level, and rapid identification that performance problems *exist*.

6 Conclusion

Architecture are following several design trends. On the one hand, they have become multi- and many-core. Developers need to address parallelism to deliver performance. On the other hand, cores have become extremely complex, and penalties when the hardware occasionally misbehaves are increasing. We argue that the latter point has not received enough attention. The *right* level of performance may be impossible to assess. But even though the actual level of performance of sequential code is also very difficult to measure, we argue that a number of low-level metrics should be made readily available to the programmer. We propose a proof-of-concept tool to help them understand at a glance the overall behavior of their systems.

Future work will investigate what metrics help diagnose severe performance penalties, derived from low-level micro-architectural features, yet can be made sensible to a programmer.

References

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference*, AFIPS '67 (Spring), 1967.

[3] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, UC Berkeley, 2006.

[4] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. In *PACT*, 2010.

[5] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP*, 2010.

[6] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing*, 2000.

[7] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[8] Arnaldo Carvalho de Melo. Performance counters on Linux. In *Linux Plumbers Conference*, 2009.

[9] Computing Systems Consultation Meeting. *Research Challenges for Computing Systems – ICT Workprogramme 2009-2010*. European Commission – Information Society and Media, 2007.

[10] Koen De Bosschere et al. *High-Performance Embedded Architecture and Compilation Roadmap*, chapter 1. LNCS. 2007.

[11] M Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par Parallel Processing*. 2001.

[12] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. MAO – an extensible micro-architectural optimizer. In *CGO*, 2011.

[13] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[14] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, (2):1, 2013.

[15] Intel. Technologies for measuring software performance. White Paper.

[16] Intel. *Intel64 and IA-32 Architectures Optimization Reference Manual*, June 2011.

[17] Michael E. Maxwell, Patricia J. Teller, Leonardo Salayandia, and Shirley Moore. Accuracy of performance monitoring hardware. In *Los Alamos Computer Science Institute Symposium*, 2002.

[18] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *ICCS*, 2002.

[19] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. We have it easy, but do we have it right? In *IPDPS*, 2008.

[20] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

[21] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Puschel. Applying the roofline model. In *ISPASS*, 2014.

[22] Rogue Wave Software. *Rogue Wave ThreadSpotter*, 2012.

[23] Erven Rohou. Tiptop: Hardware performance counters for the masses. *ICPP Workshops*, 2012.

[24] Brinkley Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4), 2002.

[25] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.