

# Tiptop: Hardware Performance Counters for the Masses

Erven Rohou

▶ **To cite this version:**

Erven Rohou. Tiptop: Hardware Performance Counters for the Masses. [Research Report] RR-7789, INRIA. 2011, pp.23. hal-00639173

**HAL Id: hal-00639173**
**https://inria.hal.science/hal-00639173**

Submitted on 8 Nov 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.




# Tiptop: Hardware Performance Counters for the Masses

**Erven Rohou**

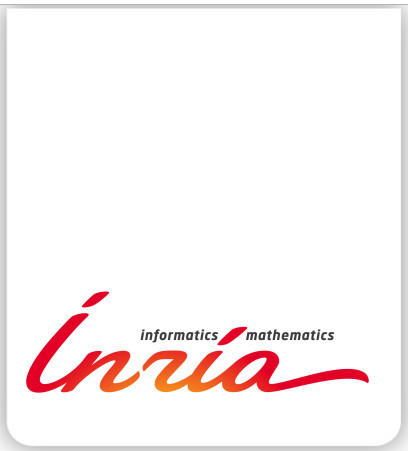
**RESEARCH REPORT**

**N° 7789**

8 November 2011

Project-Team ALF

ISSN 0249-6399 ISRN INRIA/RR--7789--FR+ENG




# Tiptop: Hardware Performance Counters for the Masses

Erven Rohou

Project-Team ALF

Research Report n° 7789 — 8 November 2011 — 23 pages

**Abstract:** Hardware performance monitoring counters have recently received a lot of attention. They have been used by diverse communities to understand and improve the quality of computing systems: for example, architects use them to extract application characteristics and propose new hardware mechanisms; compiler writers study how generated code behaves on particular hardware; software developers identify critical regions of their applications and evaluate design choices to select the best performing implementation.

In this paper, we propose that counters be used by all categories of users, in particular non-experts, and we advocate that a few simple metrics derived from these counters are relevant and useful. For example, a low IPC (number of executed instructions per cycle) indicates that the hardware is not performing at its best; a high cache miss ratio can suggest several causes, such as conflicts between processes in a multicore environment.

We also introduce a new simple and flexible user-level tool that collects these data on Linux platforms, and we illustrate its practical benefits through several use cases.

**Key-words:** performance, hardware counters, analysis tool

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

# Tiptop: compteurs de performance matériels pour les masses

**Résumé :** Les compteurs de performance matériels ont récemment reçu un surcroît d'intérêt. Ils ont été utilisés par diverses communautés pour analyser et améliorer la qualité des systèmes informatiques: par exemple, les architectes les utilisent pour extraire des caractéristiques applicatives et proposer de nouveaux mécanismes matériels; les concepteurs de compilateurs étudient comment le code produit se comporte sur un matériel particulier; les développeurs logiciel identifient les régions critiques de leurs applications et comparent différentes approches pour sélectionner la meilleure implémentation.

Dans ce document, nous proposons de mettre les compteurs de performance à la disposition de tous les utilisateurs, en particuliers les non-experts, et nous préconisons quelques métriques simples, dérivées de ces compteurs, qui sont pertinentes et utiles. Par exemple, un IPC faible (nombre d'instructions exécutées par cycle) indique que le matériel n'est pas utilisé au maximum de ses possibilités; un taux de défauts de cache élevé suggère plusieurs raisons, comme des conflits entre processus dans un environnement multicœurs.

Nous proposons un nouvel outil, simple et flexible, qui collecte ces informations dans une plate-forme Linux, et nous illustrons son utilité par plusieurs études de cas.

**Mots-clés :** performance, compteurs matériels, outil d'analyse

# 1 Introduction

The complexity of computing system increases at a fast pace, and this trend is likely to continue in the foreseeable future. Several roadmaps [4, 11, 12] predict thousands of cores on a chip by the end of this decade, which also implies complex memory hierarchies, interconnects, etc. The increasing variability of the lithographic process will also directly impact the diversity of platforms available in the computing ecosystem in the near future.

Moore's law drives the complexity of processor micro-architectures, which impacts all other layers: hypervisors, operating systems, compilers and applications follow similar trends. While a small category of experts is able to comprehend (parts of) the behavior of the system, the vast majority of users are only exposed to — and interested in — the bottom line: how fast their applications are running.

UNIX users typically rely on commands such as *ps* or *top* and look at the column `%CPU` for their processes. When this number is significantly below 100 %, they can investigate the reasons: resource conflicts (e.g. more processes than hardware threads), slow I/O, virtual memory effects, etc. When the CPU usage is close to 100 %, users can only conclude that there is no visible reason to be concerned.

CPU usage, however, 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. In Section 3.1, we describe an extreme case, derived from real-life experiments: a simple floating point computation on the x86 architecture can perform up to 87× worse than expected for pathological parameters, because of micro-code assistance, and still show 100 % CPU usage.

In this work, we propose to take advantage of hardware performance counters to expose some details of the execution of applications that are currently not easily available to the average user. We propose *tiptop*, a new tool that is as easy to use as the *top* UNIX utility, and requires neither special privilege, nor application source code, nor expert knowledge what so ever. Simple metrics can let users feel how fast their applications are actually running. Advanced users, such as compiler developers or HPC (high-performance computing) experts, can also use our tool to compute more sophisticated ratios and get deeper insights, while still using the same simple tool. The contribution of this paper is three-fold:

- we advocate that performance monitoring counters should be easily available to the rest of us, as easily as one launches the *top* utility, to quickly obtain a simple, and high-level view of *what is going on* in the machine. Users should not require complex setups, `root` access, or even application source code. They should not need to restart an application (that may have been running for hours or days, and is not acceptable in a commercial environment) to monitor its behavior;
- we advocate that simple and easy to understand metrics, such as IPC (instructions per cycle) or cache miss ratio, are meaningful and useful to non-expert users, and we illustrate our claim with several use cases;
- we present a simple utility called *tiptop*, very similar to the *top* utility present in UNIX environment, that practically achieves our goals.

The rest of this paper is structured as follows: Section 2 describes the tool we propose, with its features and characteristics. In Section 3, we present several use cases that show the practical and theoretical usefulness of such a tool and the information it can provide. We review related work in Section 4 and we conclude in Section 5.

```
  PID USER   %CPU Mcycle Minst  IPC DMIS  COMMAND
 2962 user1 100.0  26456 52125 1.97  0.0  process1
22831 user3 100.0  26417 34996 1.32  0.0  process2
 2954 user1  99.9  28180 63941 2.27  0.0  process3
 2969 user1  99.9  28184 66409 2.36  0.0  process4
22833 user3  99.9  26419 30844 1.17  0.0  process5
25242 user2  99.9  28187 18736 0.66  0.9  process6
 2944 user1  99.8  26424 45582 1.73  0.0  process7
 2965 user1  99.8  28091 40386 1.44  0.0  process8
 2972 user1  99.8  26374 36622 1.39  0.0  process9
 3043 user1  99.8  26348 36619 1.39  0.0  process10
 3058 user1  43.7  12281 19840 1.62  0.0  process11
```

Figure 1: Snapshot of processes (anonymized) running in our data center

# 2 Tiptop

*Tiptop* is a command-line tool for the Linux environment, purposely very similar to the popular *top* utility. Figure 1 illustrates its output at a glance. It is a snapshot of the activity of a node in our data center: three users and eleven processes share a node of an Intel bi-Xeon E5640 quad-core with hyper-threading (16 logical cores). Beyond the familiar PID, USER, COMMAND and %CPU, the tool reports the number of execution cycles and executed instructions (both in millions) since last refresh, the current IPC (the ratio of the previous columns), and the number of last-level cache misses per hundred instructions.

## 2.1 Description

Various means to access hardware performance counters have been proposed. We review several of them in Section 4. *Tiptop* is built on top of the `perf_event` system call recently added in Linux 2.6.31 [16], which is available on many architectures, including x86, PowerPC, Sparc or ARM. This system call lets *tiptop* register new counters for processes running on the machine, and subsequently read the values of the counters. The *tiptop* process monitors the behavior of other processes, and periodically displays the values of some ratios of interest (IPC, miss ratio, branch misprediction, etc.)

*Tiptop* has two running modes. The *live mode* periodically refreshes the screen with new values of the monitored events (similar to *top*) and lets users interactively inspect processes. The *batch mode* produces the same information, but as a streaming text output, similar to *top -b*, convenient for further processing. *Tiptop* has no graphics capability, our focus is only the collection of the raw data, in the spirit of UNIX filters such as *sed*, *awk*, etc. We plan to make *tiptop* publicly available (as soon as the anonymous review process is over).

## 2.2 Features

Our goal is to make the collection of performance and bottleneck data as simple as possible. Extremely simple installation and usage are a basic requirement. In particular, we stress the following points.

- Installation is only a matter of compiling the source code. No patching the Linux kernel is needed, and no special-purpose module needs to be loaded.

```
int perf_event_open(struct perf_event_attr* hw,
                    pid_t pid, int cpu, int grp, int flags)
```

Figure 2: Linux system call

- No privilege is required, any user can run *tiptop*[1].
- The usage is similar to *top*. There is no need for the source code of the applications of interest, making it possible to monitor proprietary applications or libraries. And since there is no probe to insert in the application, understanding of the structure and implementation of complex algorithms and code bases is not required.
- Applications do not need to be restarted, and monitoring can start at any time (obviously, only events that occur after the start of *tiptop* are observed).
- Events can be counted per thread, or per process.

The collected events and displayed ratios are fully customizable. The Linux header files provide a few generic events (total cycles, instructions, cache misses, branch mispredictions) that make it easy to compute portable metrics, such as IPC or last-level cache miss ratio, as shown in Figure 1. The default configuration collects these generic and portable events. But the tool is very flexible and lets users monitor any target-specific event supported by the underlying architecture.

Our focus is not the detection of very fine-grain events, that would help analyze the performance of a small function, or study the cost of a particular lock, for example as in the work of Demme and Sethumadhavan [13]. On the contrary, we provide a coarser-grain view of the behavior, and we typically take samples every few seconds.

## 2.3 Implementation

*Tiptop* is basically an infinite loop that displays how many times the requested events have happened for each task (process or thread), and then goes idle until some timeout expires or the user pressed a key.

Performance counters are accessed thanks to the `perf_event` system call, available since Linux 2.6.31 [16]. Its prototype is shown in Figure 2. `pid` is the task ID to monitor. We set `cpu` to -1 to monitor events per task (as opposed to CPU). `grp` and `flags` are unused. The `hw` struct is defined in `/usr/include/linux/perf_event.h`. It specifies which event is to be tracked. Generic target-independent events are defined in the same header file. Target specific events must be looked up in the vendor's architecture manuals (such as [24]). The system call returns a file descriptor from which we read the values of the counters using a regular `read`.

Additional information such as %CPU, processor on which a task is running, etc. is retrieved from the `/proc` filesystem.

For the live-mode, we rely on the widespread *ncurses* library to refresh the screen and pretty-print information. In case the library is not available, *tiptop* can still be built, but only batch-mode is functional.

[1] Non-privileged users can only watch processes they own. Ability to monitor anybody's process opens the door to side-channel attacks [5, 6].

## 2.4 Validation

We validated our tool in two ways. Note that our goal is not validate the hardware counters themselves, but to confirm that our tool correctly reads the values proposed by the counters.

First, we manually crafted micro-kernels for which we can analytically estimate the number of instructions (by inspecting the assembly file of a single basic-block loop), the number of cache misses or the misprediction ratio (random or periodic indirect jumps to well known locations). *Tiptop* reports numbers in line with predictions. We describe another such micro-benchmark in Section 3.1.

Second, we measured the total number of executed instructions of all SPEC 2006 benchmarks, with reference input, and compared with the numbers produced by Pin [26]. We used the unmodified *inscount2* example provided by the Pin distribution version 2.8. The number of instructions we obtain is on average within 0.06 % (i.e. $6 \times 10^{-4}$) of Pin's count.

## 2.5 Perturbation

There are two main methods to collect data from performance monitoring counters: counting and sampling. Counting is often referred to as an *exact count*, while sampling is statistical in nature. We currently use counting (see Section 4 for more discussion on sampling vs. counting).

Measurement introduces perturbation. In performance monitoring, the bias typically comes with the additional code introduced in the application to setup and read the counters. Recent work [13] focused on reducing this overhead to a few instructions by directly reading the Intel x86 registers, instead of invoking a system call, and dealing with occasional overflows when they occur. While this approach clearly reduces the number of instructions and the time needed to read data from a probe, it also (slightly) modifies the code of the application. Unfortunately, it has been shown that the performance of modern architectures is very unstable, and a single *nop* instruction can impact the performance [20]. Mytkowicz et al. [31] also describe the so-called *observer effect*: they show that the mere fact of inserting probes impacts the layout of the code, which interferes with branch predictors, instruction caches, etc. The same authors previously studied how changing the linking order of object file, or even adding a totally unrelated variable to the UNIX environment of the process can change performance in unpredictable ways, simply because the start address of process' stack changes, and cache effects differ [32]. Other prior work [37] report on the impact of instrumentation on the result of floating point computation, and on the interaction with the virtual memory layout.

Our approach avoids these pitfalls altogether by leaving the application code untouched. No change is required in the UNIX environment either. The impact is limited to the cost of saving a few counters at context switches of the monitored tasks, and the system calls of the monitoring task (one per monitored process and per event of interest) every few seconds. The memory footprint is small: the executable for a Nehalem workstation is 40 KB (.text section is 21 KB) with support of the *ncurses* library, and 32 KB (.text is 16 KB) without *ncurses*. The *tiptop* process is idle most of the time between refreshes. We measured that the CPU activity of *tiptop* itself is below 0.06 % when refreshing every five seconds. Note that in a multicore environment, the Linux scheduler will also likely place *tiptop* on the least loaded core in order to further reduce interference.

We ran the entire SPEC 2006 suite, reference input, on an Intel Xeon W3550, 3.07 GHz with Linux 2.6.31 (three runs, median value reported, as per SPEC). Having *tiptop* run concurrently with the benchmarks resulted in a 0.7 % degradation of the score. As a comparison, we ran the entire suite ten times (three runs each, median reported) on the same idle machine and we measured that the variability across runs is 1.4 % on average. The impact of *tiptop* is definitely within the order of magnitude of the noise. The suite run with *inscount2*, as described in the previous subsection, is 1.7× slower.

## 2.6 Metrics and Methodology

We can compute any metrics that can be derived from performance monitoring counters available on the target hardware. The only practical limit is the number of available concurrent counters. Older machines used to have only a few counters, however newer processors have ample room for event counting. Our Intel Xeon W3550, for example, supports up to sixteen simultaneous events.

Complex metrics might be useful for advanced users, such as HPC experts or compiler writers. But we claim that, in most situations, a few simple metrics can characterize the behavior of an application. We also focus on coarse-grain samples, typically every few seconds.

The simplest metrics is probably IPC. Many pitfalls are related to IPC. In many cases, it is not a direct proxy for performance. For example, a compiler cannot be evaluated (only) by the IPC of the generated code[2]. But, from the point of view of a user given a program executable, the number of instructions (I) to execute to completion is fixed[3]. The higher the IPC, the lower the number of cycles (C), the better.

As noted by Diamond et al. [15], advertised peak performance is rarely attainable, and IPC is only useful when compared to a known reasonable value. A *good* IPC does not necessarily mean that nothing can be improved, however a *low* IPC is a symptom that something is wrong. Users may or may not improve it, depending on their skills, the cause for under-performance, and the steps they can take (such as recompile, migrate the process, modify the source...), but we show in Section 3 that the information is useful in many contexts.

Once we identify a performance bottleneck, several other metrics help pinpoint its cause. Our purpose in this paper is not to provide an exhaustive list of metrics, but to illustrate a methodology with a few use cases (see next section). Diamond et al. [14] propose to consider two metrics: FPC (flops per cycle) and LPC (loads per cycle) to characterize respectively the CPU subsystem and the memory subsystem. Intel provides a list of "drill-down techniques for performance analysis" in §B.5 of [23], as well as an extensive list of event ratios of interest (§B.6). Most of them target advanced users, but a few of them are general enough to give a high-level view of how the architecture performs. Coarse grain characteristics of the application (regardless of its behavior) can be obtained with related metrics: FPI (flops per instruction), LPI (loads per instruction), or BPI (branches per instruction). The reported instruction mix is useful in selecting the most appropriate processor in a family of binary compatible chips, for example with the Roofline methodology [38].

[2] What really matters is the total number of *cycles*; poor code generation might add useless independent instructions that artificially inflate the IPC without making the program run any faster.

[3] Multithreaded applications relying on spinlocks require special handling, since more instructions spent waiting mean lower performance.

# 3 Use Cases

In this section, we illustrate with several use cases the usefulness of a tool such as *tiptop*, and the insights it can easily provide. We first present how we quickly diagnosed an algorithm developed by colleague biologists in the R language and running in an interpreter. We then focus on application phase behaviors and show how we can identify them, at the full running speed of the application. The third use case illustrates how we could easily reproduce and extend recently published work. Finally, we demonstrate how CPU usage can be misleading when evaluating the performance of an application, especially in a multicore environment, and we present how we can estimate the cross-process interferences.

We stress here that these use cases are purposely simple. The goal of this section is not to present new theoretical results, but to illustrate our approach on well known phenomena. Each use case also targets a different category of user.

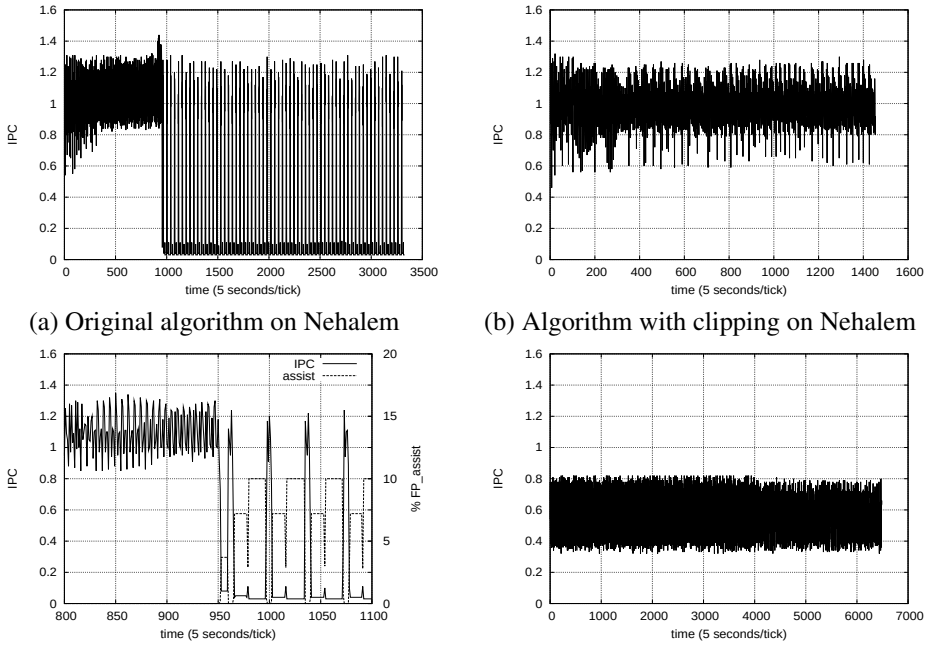
## 3.1 Evolutionary Algorithm

Our first test case is an algorithm developed by biologists to model the evolution of a population subject to external factors such as temperature, growth of plants, winds, etc. The main outer loop of the algorithm represents time steps. The population is described as a large matrix where each cell represents the number of individuals in a small geographic area. Computations consist in matrix multiplications as well as scalar operations on the matrix elements. The algorithm is implemented in the R language, a system for statistical computation [21].

The algorithm was felt to run too slowly. We experimented on an Intel Xeon W3550 clocked at 3.07 GHz, Linux 2.6.31, R 2.10.1 and measured the IPC of the R interpreter when running the algorithm. We took a sample every 5 seconds. Figure 3 (a) illustrates the evolution of the average IPC over the complete run (3327 samples, 4.6 hours). The first iterations show a noisy IPC signal, but close to 1. After 953 time steps, the IPC suddenly drops to 0.03 (with brief pulses). This algorithm is iterative, each iteration performing the same amount of work, and it is not expected to have phases. Analyzing the behavior of the algorithm during the first 1.3 hours of its execution would have shown no abnormal behavior. Knowing the instant when *something* changed let us focus the investigation. We quickly discovered that the algorithm is not numerically stable for particular data sets, and matrices fill with infinite and NaN (not-a-number) floating points values.

On some Intel processors, floating point computations can be assisted in microcode. This is activated by the presence of non regular floating point values. It is described as "extremely slow compared to regular FP execution" in the Intel architecture manuals [23]. The number of executed micro-operations can be tracked by a counter. We added a new column to *tiptop* in order to trace simultaneously IPC and *FP_assist* events. Figure 3 (c) illustrates a zoom on the transition between the two phases: IPC is still reported the left axis, the number of assisting micro-operation per hundred instructions is reported on the right axis (equal to zero hence invisible for the first 953 time steps). The clear correlation confirms our analysis.

We clipped the values of the matrices to force them in a finite interval at each iteration of the main loop. Figure 3 (b) shows the result of the same experiment with the modified algorithm. The slight overhead added by the clipping is negligible in front of the savings in assisted operations. The average IPC remains centered around




(a) Original algorithm on Nehalem

(b) Algorithm with clipping on Nehalem

(c) Original algorithm on Nehalem (zoom)

(d) Original algorithm on PowerPC

Figure 3: IPC of R emulator

Table 1: Measured behavior of the floating point micro benchmark

| | finite | | infinite/NaN | |
|---|---|---|---|---|
| | IPC | %FP_assist | IPC | %FP_assist |
| x87 | 1.33 | 0 | 0.015 | 25% |
| SSE | 1.33 | 0 | 1.33 | 0 |

the value 1. The algorithm also completes in two hours, that is a 2.3× speedup. The speedup obtained on the faulty part alone is 4.8×.

The impact of non-finite values can be easily verified in a controlled environment with a micro-benchmark. Consider the (simplified) source code of Figure 4: a small loop continuously adds two floating point variables x and y, which can be initialized to finite, infinite, or NaN values. We compile this code using GCC 4.4.3 at `-O2` optimization level in two different configurations: `gcc -mfpmath=387` produces x87 code, and `gcc -mfpmath=sse` uses the SSE instruction set extension (in scalar mode, no auto-vectorization is involved). Figure 5 reports the assembly code corresponding to the loop. The IPC reported by *tiptop* are shown in Table 1. We also show the number of instructions which required micro-operation FP_assist per hundred instructions. Infinite and NaN values produce the same result and are reported together. Since the loop consists in four instructions, we confirm that all x87 floating point additions required assist (as expected by construction of the micro-benchmark). The slowdown is as large as 87× (1.33/0.015).

We also experimented on a different kind of machine: a PowerPC PPC970 clocked at 1.8 GHz and running Linux 2.6.32. The resulting IPC for the original algorithm is shown on Figure 3 (d). The PPC970 (an older machine) takes longer to complete the

```
#include <math.h>
double x, y;

void init_fin() { x = -1.0; y = 1.0; }
void init_inf() { x = 0.0; y = INFINITY; }
void init_nan() { x = -INFINITY; y = INFINITY; }

int main(int argc, char* argv[]) {
  double z = 0.0;
  init_XXX();  /* choose init values here */
  for(i=0; i < max; i++)
    z += x + y;
  return 0;
}
```

Figure 4: Micro benchmark to measure impact of non-finite FP values

| .L16:<br>addq $1, %rax<br>fadd %st, %st(1)<br>cmpq %rbx, %rax<br>jne .L16 | .L16:<br>addq $1, %rax<br>addsd %xmm1, %xmm0<br>cmpq %rbx, %rax<br>jne .L16 |
|---|---|
| x87 code | SSE code |

Figure 5: Assembly code of micro benchmark

workload because of a lower clock frequency and a lower IPC. But we also observe that it does not exhibit the Nehalem behavior related to floating point values.

Obviously, the same conclusion could be drawn by adding heartbeats to the application. This requires that the source code be available, and the programmer be willing to dive into a possibly complex application to insert the markers at the right locations. Even though R is open-source, it is worth noting that we entirely handled the analysis as if it was close-source, showing that we can handle such environments.

*Tiptop* makes it straightforward to identify in real time a sudden change in application behavior, and helps focus the analysis. In this particular case, it is interesting that a performance problem exposed a latent bug in the algorithm, which should not have diverged towards infinite values.

## 3.2 Application Phases

In many scenarios, it is key to monitor applications phases. Most often, CPU usage remains close to 100 % and cannot provide much insight. Simple metrics derived from hardware counters can provide more detailed information. Figures 6 and 7 illustrate the IPC of several SPEC 2006 benchmarks on Intel Nehalem and Core micro-architectures, as well as PowerPC PPC970. The horizontal axis is time, with one sample per second, the vertical axis is IPC.

Such capability lets users compare the behaviors in various conditions. Figures 6 (a) and (b) and 7 (a) show that the selected benchmarks have similar behaviors on different architectures, except for the actual value of the IPC and hence the total run time. Still, slight differences exist, for example *435.gromacs* (Figure 7 (b)) shows small but still noticeable variations on Nehalem, and the relative IPC of the last phases of *473.astar* differ on PowerPC.

With this information, developers can quickly identify benchmarks and data sets that result in lower performance than expected, in a limited number of runs. Moreover, this can be done at full program speed, instead of using simulators or emulators. This is in contrast to earlier work on phase tracking, such as Sherwood et al. [34], who chose to limit their study to a subset of the benchmarks because of lengthy simulations.

More advanced users can also start running their applications at full speed, and attach a debugger or analyzer (such as a Pintool [26]) when a particular phase has started. Similarly, many papers in computer architecture are based on simulators, and benchmarks are run after skipping the first billion instructions or so to avoid the initialization phase. Carefully looking at performance profiles can help define a more accurate number of instructions for each particular combination of architecture, compiler, and compiler flags. Figure 8 illustrates the IPC for *473.astar* as a function of the number of executed instructions, for three different processors. Both Intel processors execute the same binary. The PowerPC slightly shifts compared to the other two. This very simple graph helps focus the analysis on the relevant parts of the execution, making the experiments faster and the results more sensible, for example by choosing SimPoints [33] based not only on the similarity of basic block vectors, but also on some dynamic properties.

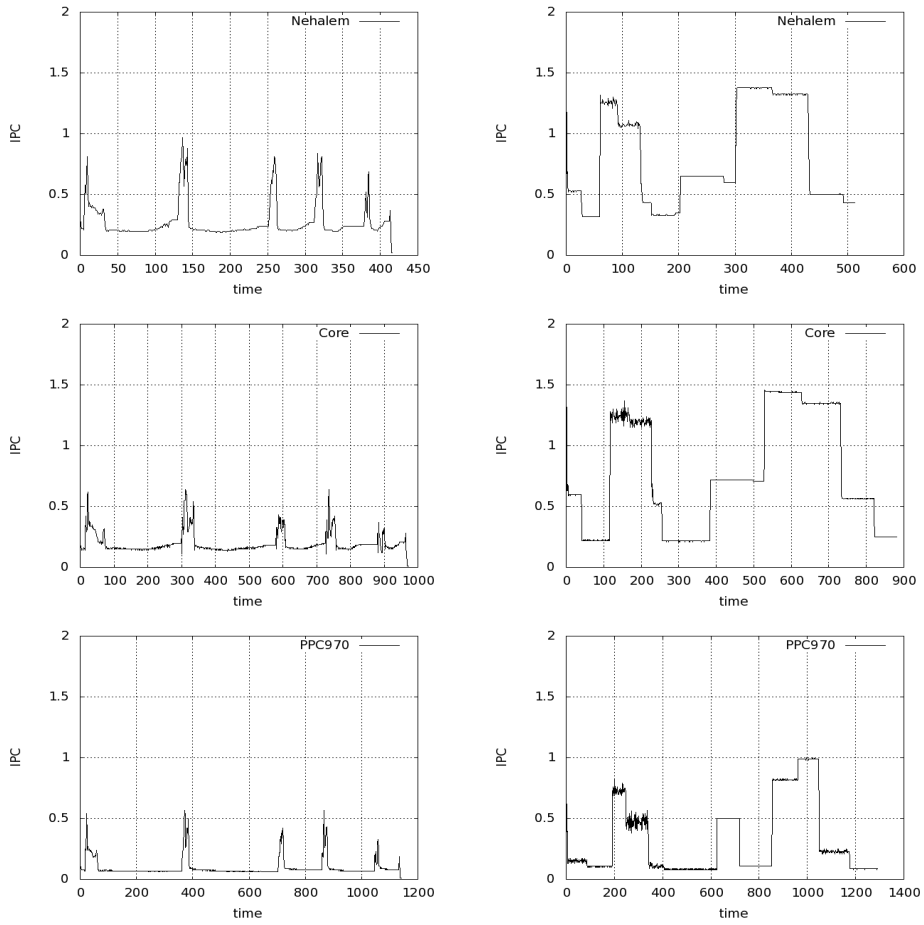
Obviously, phase detection can be (and has been) done in many different ways. Alternatives include code instrumentation, simulation, and other performance monitoring tools. We show here that our tool achieves the goal with unprecedented ease of use and speed.

## 3.3 Impact of Code Generation

Jayaseelan et al. [25] recently published work in which they study the impact of compiler technology on the performance. In particular, they measure the total number of cycles and executed instructions for the SPEC INT 2006 benchmarks, and compute the respective IPC. Among other things, they observe — not surprisingly — that the highest IPC does not necessarily characterize the fastest program.

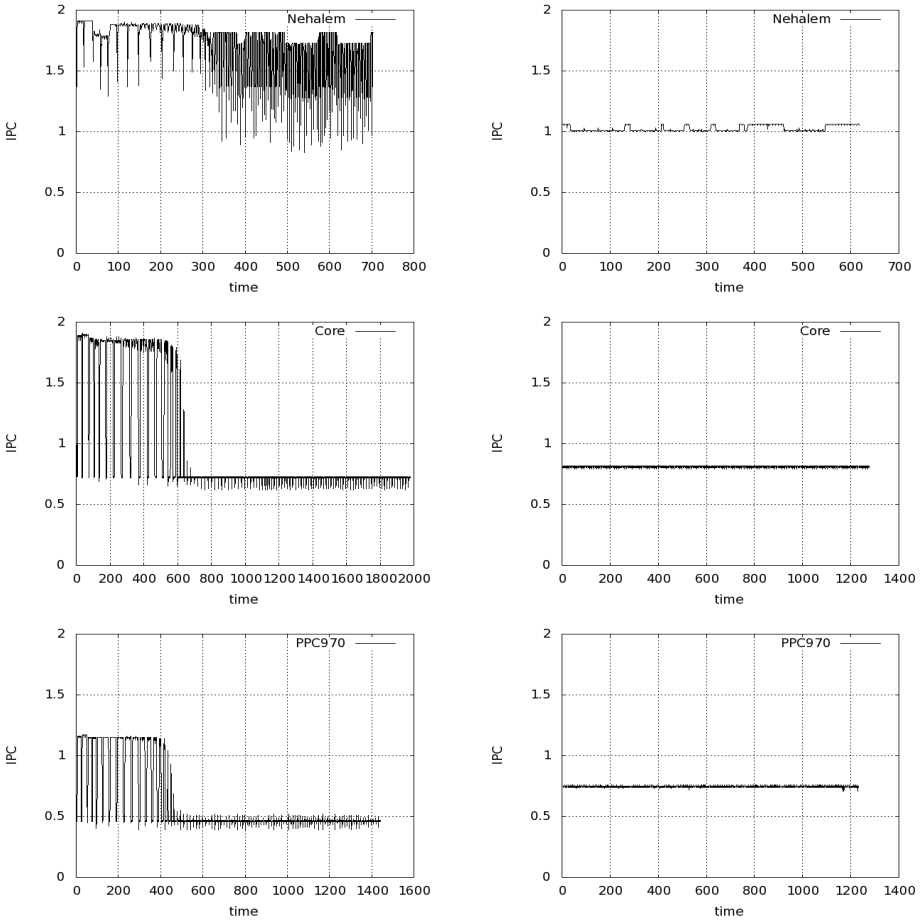
*Tiptop* makes it very easy to run such an experiment and even gather more insights. We simply run the benchmarks with the reference input set while collecting the performance counter values from the observing process. The machine is an Intel Xeon W3550 (Nehalem), and the compilers are GCC 4.4.3 and icc 11.0. Results are reported in Figure 9. Similarly to Jayaseelan et al., we observe cases where a higher IPC yields better performance (cf. Figure 9 (a), *456.hmmer*), and cases where performance is better despite a lower IPC (cf. Figure 9 (b), *482.sphinx3*). However, we can also observe slightly more complex behaviors, such as *464.h264ref* (Figure 9 (c)) where both running times are close, but two different phases are clearly visible. In the first and shortest phase, GCC produced a higher IPC, in the second phase it produces the lowest IPC. This inversion phenomenon was not visible to the authors of the original paper [25] because they observe data aggregated over the entire run of the benchmark. Finally, the executables produced by GCC and icc for *433.milc* execute exactly at the same speed, even though the IPC produced by GCC is constantly higher that icc's (Figure 9 (d)).

Our purpose here is not to entirely reproduce this previous work, but to show how the approach we propose simplifies the experimental setup. These authors had to first extract traces from different parts of the program, and then validate that they capture program behaviors before running them through their in-house cycle-accurate simulator. In contrast, our approach lets us run the entire program at full speed to collect data, with much less experimental burden.




(a) 429.mcf (b) 473.astar

Figure 6: IPC of 429.mcf and 473.astar




(a) 410.bwaves (b) 435.gromacs

Figure 7: IPC of 410.bwaves and 435.gromacs

## 3.4 Process Conflicts

Many institutions use dedicated servers for long running computations. These facilities are often referred to as *data center*, *compute farms*, *grids*, or more recently *clouds*. Users access these facilities by submitting jobs to execution queues. A runtime system assigns priorities to the jobs and runs them according to predefined policies. Nodes are typically binary compatible, for ease of dispatching jobs, but rarely identical, simply because they were not acquired at the same time, some fail and need to be replaced, etc. The goal of the scheduler is to maximize average throughput while maintaining some level of fairness, and accommodating special requests (task x needs a 64-bit OS, task y needs at least some amount of memory and a quad-core only...) Many heuristics apply, such as increasing priority of short running processes, dedicating some nodes for long running tasks, and so on. A sensible rule of thumb is to load a node with as many jobs as there are logical cores, and to keep memory usage below the available physical memory.

Our lab's setup consists in about 100 nodes. Each node is a bi-Intel Xeon. Configurations include dual-cores and quad-cores, and clock frequencies range from 1.6 GHz to 3.4 GHz. This is a production environment where researchers submit their real workloads. The scheduler is based on Sun Grid Engine (SGE) 6.2u5. It defines sixteen queues for jobs of different wall-clock run time, memory requirements, and urgency (ASAP vs. overnight). Jobs are spawned in order in each queue, the number of concurrently running jobs is limited by the number of logical cores of each node.

We have been using *tiptop* to monitor the behavior or workloads on all nodes. Figure 10 illustrates a snapshot of the real life of one of the nodes. Each time step represents ten seconds of execution. This particular node is an bi-quad-core Intel Xeon E5640 clocked at 2.67 GHz (Nehalem Westmere micro-architecture), with hyperthreading. We observe that `user1` has two jobs running over the entire time interval, and `user2` suddenly has five jobs scheduled for roughly one hour. This is a total of seven jobs, on an 8-core machine, and we controlled that the CPU usage is above 99.3 % at all times.

It is clear that the start of the second user's jobs coincides with a drop of the IPC of the first user's jobs. Between time steps 2350 and 2580 (38 minutes), they drop respectively from 1.3 to 1.05, and from 1 to 0.8, a 20 % slowdown for both, because of additional contention of the last-level cache (not shown in the figure).

We also observe an apparent interaction between the two processes of the first user, for which we do not have any explanation. The data center being a production environment, we cannot reproduce past experiments.

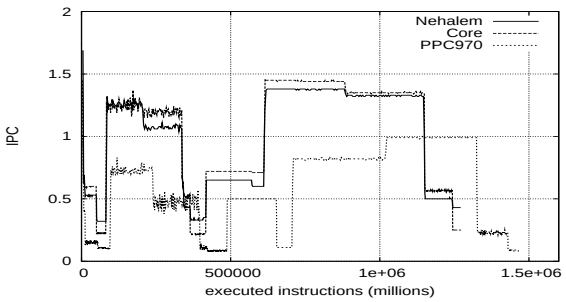
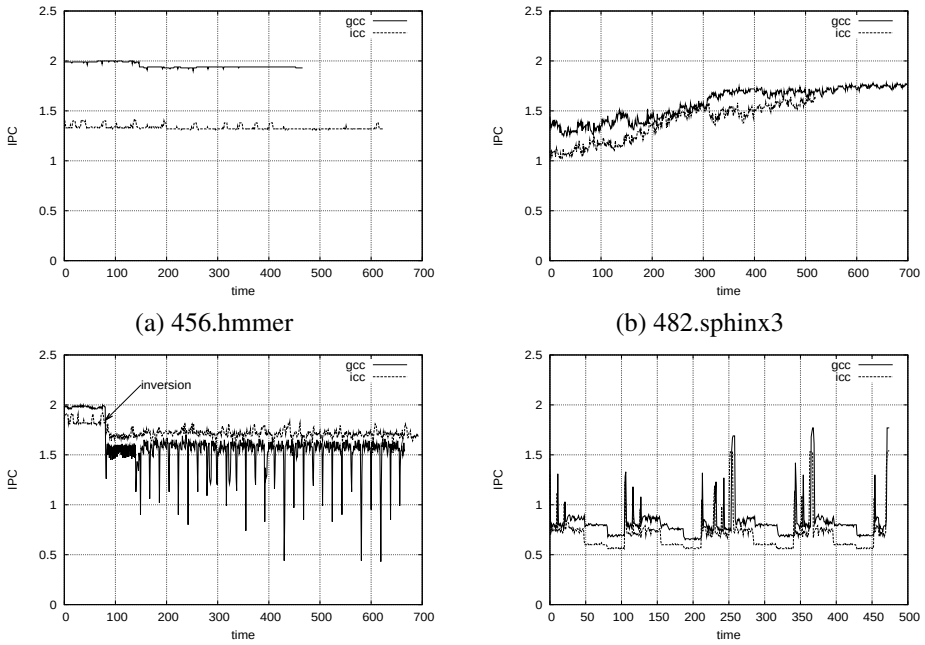



Figure 8: IPC versus the number of executed instructions for 473.astar




(a) 456.hmmer (b) 482.sphinx3

(c) 464.h264ref (d) 433.milc

Figure 9: IPC produced by different compilers

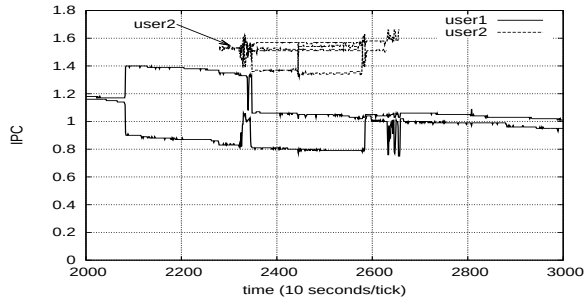
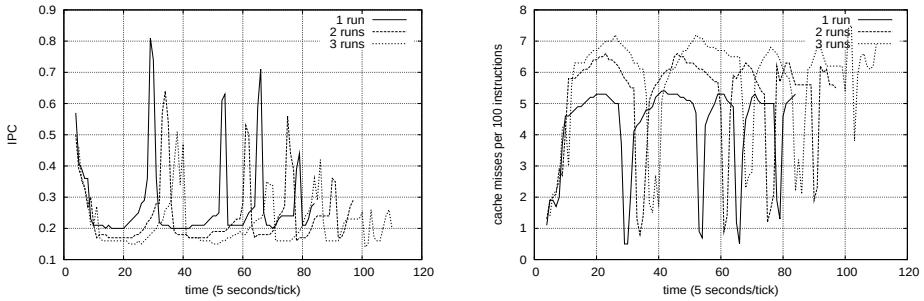
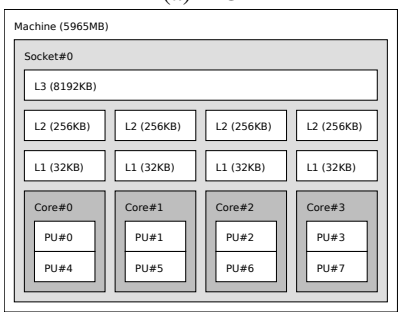
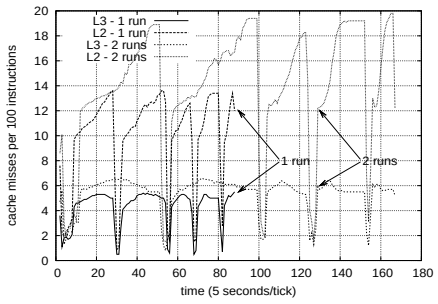



Figure 10: Snapshot of the load on one node of our data center




(a) IPC (b) cache misses




(c) machine topology (d) cache misses, same physical core

Figure 11: Cross-core interferences for 429.mcf on quad-core Nehalem

To analyze this behavior in a controlled environment, we considered the benchmark *429.mcf* from SPEC 2006, known for exercising the memory hierarchy. We first ran *mcf* on our quad-core Nehalem and measured the IPC. Then, we ran two copies of *mcf* in parallel. Finally we ran three copies in parallel (running four copies exceeds the memory). The topology of our machine is illustrated on Figure 11 (c), as produced by the *hwloc* software [7]. When multiple copies run, we bond each instance to a given physical core using the *taskset* Linux utility. Figure 11 (a) reports our findings. We checked again that, at all times, the CPU usage is above 99.3 %. Still, execution time increases and IPC decreases as the number of instances increases, with up to 30 % slowdown in the case of three instances. The reason is that all cores share the last-level cache, and processes interfere, even though they run on different physical cores. The increased contention on the cache is illustrated in Figure 11 (b): each curve shows the number of cache misses on the last-level cache per hundred instructions.

As a last experiment, we ran two instances of *mcf* on the same physical core (i.e. on logical cores 0 and 4). While the situation is fictitious when all other cores are idle, it is realistic when the number of threads exceeds the number of physical cores. In this case, the number of L3 misses is similar to having the two processes on different cores, which is easily explained by the fact that the L3 cache is shared across all cores. The number of L2 misses increases dramatically, and causes a 2× slowdown. Figure 11 (d) illustrates the number of L2 or L3 misses per hundred instructions in the cases of one and two processes on a single core. This example shows that execution can incur a 30 % slowdown even though the CPU usage is at the maximum.

Previous works have investigated the interference between jobs scheduled on a SMT processor. Snavely and Tullsen [35] studied *symbiotic* scheduling. Eyerman and Eeckhout [17] later introduced probabilistic job symbiosis modeling. Both rely on

simulation for the experimental evaluation, and the latter requires extra hardware for the cycle accounting. In contrast, we can detect pathological (or *anti-symbiotic*) interaction for free and in a real system, not only at the SMT level, but also at the multicore level. Mars et al. [27] also study cross-core interferences by generating contention from a neighboring core, and measuring the sensitivity of the IPC thanks to performance counters. We differ in the fact that we do not use any contention generator, but rather observe the behavior in its real context. Also, we do not attempt to relate the contention to any particular region of the source code, but focus on simple and coarse grain metrics.

Moscibroda et Mutlu [30] study the contention at the DRAM level. We currently cannot observe this event for which there is no hardware counter. However, recent processors [24] have counters for the latency of memory accesses. We plan to use them in the future to detect similar situations.

# 4 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 [36]. The number of available countable events, and counters vary greatly across architectures [28]. Counters can be used in two modes: counting or sampling. Moore [29] compares efficiency, accuracy and bias of each method in the PAPI library [8]. The paper shows that even though sampling is known to be less accurate, counting can also be inaccurate. Our approach is currently based on counting, however setting up and reading counters is done outside the monitored applications, and we read counters at coarse time intervals, in the order of seconds, making the overhead insignificant (see also Section 2.5 about overhead and perturbation).

Several approaches have been taken to give users access to the hardware counters. PAPI [8] is a library that encapsulates low-level access to the hardware and provides an API to programmers for setting up, starting, stopping, and reading the counters. PAPI also abstracts common events and provides a convenient cross-platform standard naming for many useful events, such as cycle count, floating point instructions, etc. Other tools include Rabbit [19], OProfile [2], libpfm and perfmon2 [1]. LiMiT [13] is a very recent proposal to reduce the overhead of reading counters, by directly reading machine registers and avoiding the system call. The authors report much lower perturbation of the analyzed applications and thus observe different behaviors on commercial workloads. Demme and Sethumadhavan [13] also provide a history and review of performance counters. PAPI and LiMiT both require access to the source code of the application to be monitored. LiMiT has a much faster read, but requires changes to the Linux kernel. In comparison, our approach runs on unmodified kernel and applications. Already running applications do not need to be restarted. In particular, source code is not needed, making it possible to analyze closed-source applications or libraries.

Some tools integrate the access to the performance counters with a graphical interface. Intel offers the VTune [22] performance analyzer, which samples the execution based on hardware or operating system events and combines the results with other analyzes to provide tuning advices. Similar to our approach, VTune does not require recompilation. On Windows, it can also attach to an already running process. For most events to be available, however, installation must be done by `root`. WAIT [3] is another tool developed by IBM to diagnose idle time in commercial workloads. It is similar to our approach in many respects: there is no need to recompile applications,

or even restart them — a key requirement in commercially deployed setups — and reasoning is based on a set of simple metrics. WAIT collects its information from the Java VM, whereas we rely on hardware performance counters.

Related work [20, 31, 32] study how the insertion of probes impacts the execution of the monitored application. We discuss them in Section 2.5. In brief, our approach avoids these phenomena by not modifying the application or its execution environment.

Performance counters have also been studied in the context of multicore: Diamond et al. [14] identify a few metrics that characterize the behavior of a workload on a multicore target. They also observe that traditional optimizations may be detrimental in the *multicore regime*. The new metrics lead them to propose a new optimization, called microfission, that specifically addresses multicore bottlenecks. The accompanying technical report [15] has a survey of performance measurement tools. Another aspect of parallel programs is data races. Greathouse et al. [18] propose to rely on hardware counters to limit the overhead of dynamic race detectors. Our approach focuses on simple metrics that can be of immediate use to the vast majority of users.

Optimization developers also have an interest in performance counters. Hundt et al. [20] rely on counters to understand how apparently benign modifications of the assembly code impacts performance on complex x86/64 micro-architecture, and to drive the development of assembly-level optimizations in the MAO tool. Cavazos et al. [10] have been using counters to characterize benchmarks. They use machine learning to derive a good set of code transformations. While we do not propose any optimization in our work, we also use counters to hint at possible causes for bad performance. Jayaseelan et al. [25] study the effect of compiler technology on integer SPEC workloads. Their study reports total number of occurrences for each particular event. As shown in Section 3.3, our approach could give more insight by showing real-time evolution of the metrics and comparing phases instead of only a global aggregated value.

The tools that appear closest to our proposal are Linux's perf [9] and Intel's PMC. Perf runs a command and records performance counter statistics from it. Raw data can then be processed to produce various reports, such as the number of events in each function or the time spent in kernel functions. In contrast to our solution, perf needs to start the application itself, and cannot attach to an already running process. Intel's PMC is also similar to our tool in that it periodically displays the number of occurring events, such as IPC, and cache misses at various levels or the memory hierarchy. However it requires `root` privilege, it supports only the Westmere/Nehalem micro-architecture, and it shows the total number of events per logical core, not per process.

# 5 Conclusion

This paper is concerned with users facing the increasing complexity of computing systems at many levels: architecture, hypervisor, operating systems, compilers, etc. On the one hand, complexity is the means for increased performance. On the other hand, it is becoming more difficult for users to have an understanding of how well applications perform. We advocate that performance monitoring counters can be helpful even for non-expert users to obtain a better understanding of the relative performance of applications, and we propose *tiptop*: a new tool, similar to the UNIX *top* utility, that requires no special privilege and no modification of applications. *Tiptop* provides more informative estimate of the actual performance than existing UNIX utilities, and better ease of use than current tools based on performance monitoring counters. With several use cases, we have illustrated possible uses of such a tool.

## Acknowledgements

We would like to thank our colleagues from the ALF project-team for the fruitful discussions, as well as the staff operating the compute grid for their support in the collection of data.

## References

[1] perfmon2. http://perfmon2.sourceforge.net.

[2] Oprofile. http://oprofile.sourceforge.net, 2011.

[3] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 739–753, 2010.

[4] 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, EECS Department, University of California at Berkeley, December 2006.

[5] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[6] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. *International Conference on Information Technology: Coding and Computing*, 1:586–591, 2005.

[7] 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 *18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, Pisa, Italy, February 2010.

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

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

[10] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, Washington, DC, USA, 2007.

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

[12] Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050/2007 of *Lecture Notes in Computing Science*, chapter 1, pages 5–29. Springer Berlin / Heidelberg, 2007.

[13] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[14] Jeff Diamond, Martin Burtscher, John D. McCalpin, Byoung-Do Kim, Stephen W. Keckler, and James C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[15] Jeff Diamond, John D. McCalpin, Martin Burtscher, Byoung-Do Kim, Stephen W. Keckler, and James C. Browne. Making sense of performance counter measurements on supercomputing applications. Technical Report TR-10-25, University of Texas at Austin, Department of Computer Science, 2010.

[16] Stéphane Eranian. Linux new monitoring interface: Performance counter for Linux. In *CScADS Workshop on Performance Tools for Petascale Computing*, Lake Tahoe, CA, USA, July 2009.

[17] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–102, New York, NY, USA, 2010.

[18] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Franck, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In *International Symposium on Computer Architecture (ISCA)*, June 2011.

[19] Don Heller. Rabbit: A performance counters library for Intel/AMD processors and Linux. http://www.scl.ameslab.gov/Projects/Rabbit.

[20] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. MAO – an extensible micro-architectural optimizer. In *International Symposium on Code Generation and Optimization (CGO)*, April 2011.

[21] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

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

[23] Intel. *Intel64 and IA-32 Architectures Optimization Reference Manual*, 248966-024 edition, April 2011.

[24] Intel. *Intel64 and IA-32 Architectures Software Developer's Manual – System Programming Guide*, 325384-039us edition, May 2011.

[25] Ramkumar Jayaseelan, Anasua Bhowmik, and Roy D. C. Ju. Investigating the impact of code generation on performance characteristics of integer programs. In *Workshop on Interaction between Compilers and Computer Architecture (INTERACT)*, pages 4:1–4:8, New York, NY, USA, 2010.

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, New York, NY, USA, 2005.

[27] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 167–176, New York, NY, USA, 2011.

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

[29] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *ICCS*, pages 904–912, 2002.

[30] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *16th USENIX Security Symposium*, pages 18:1–18:18, 2007.

[31] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. We have it easy, but do we have it right? In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–7, April 2008.

[32] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–276, 2009.

[33] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS-X, pages 45–57, 2002.

[34] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *30th International Symposium on Computer Architecture (ISCA)*, pages 336–349, New York, NY, USA, 2003.

[35] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, New York, NY, USA, 2000.

[36] Brinkley Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64–71, Jul/Aug 2002.

[37] V.M. Weaver and S.A. McKee. Can hardware performance counters be trusted? In *International Symposium on Workload Characterization (IISWC)*, pages 141–150, September 2008.

[38] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, April 2009.

# Contents

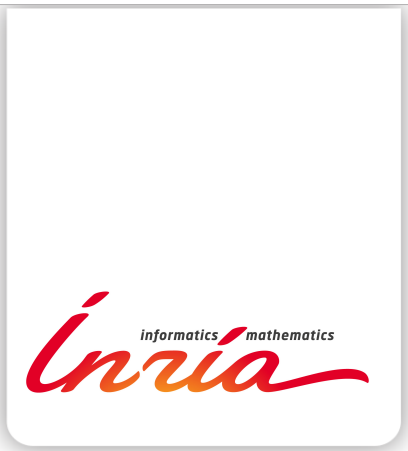
**1 Introduction** **3**

**2 Tiptop** **4**
2.1 Description . . . . . 4
2.2 Features . . . . . 4
2.3 Implementation . . . . . 5
2.4 Validation . . . . . 6
2.5 Perturbation . . . . . 6
2.6 Metrics and Methodology . . . . . 7

**3 Use Cases** **8**
3.1 Evolutionary Algorithm . . . . . 8
3.2 Application Phases . . . . . 10
3.3 Impact of Code Generation . . . . . 11
3.4 Process Conflicts . . . . . 14

**4 Related Work** **17**

**5 Conclusion** **18**




**RESEARCH CENTRE**
**RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399