CS533 Project Report: Tiptop

Kevin Dyer and Paul Vu

November 26, 2012

1 Introduction

Hardware performance counters are architecture-specific special-purpose registers integrated into modern central processing units (CPUs) used for performance profiling. As an example, modern Intel processors support hardware counters that report events such as CPU utilization, L1 cache misses, branch mispredictions and a host of other CPU-specific performance indicators. This information is invaluable when profiling low-level software performance. What is more, compared to traditional software-based profiling, hardware performance counters incur negligible overhead because they are integrated into the hardware. However, support for hardware counters is still in early stages, and was not integrated into the Linux kernel until the 2.6.31 release. In addition, some architectures support hardware counters, but have a hard limit on the number of events, sometimes only two or four, that can be monitored at a time.

Tiptop [3] is an application developed by Erven Rohou[1] and is an attempt towards an intuitive interface for accessing hardware performance counters in modern CPUs. The interface for tiptop is similar to the popular top[2] utility. In tiptop's default configuration, it provides real-time per-process hardware counter statistics, such as the number of CPU instructions executed per-process. The values that appear on the real-time screen are configurable by the user. Through use of an XML file, multiple screens can be configured and flipped through. In Figure 1 we have a screenshot of tiptop's default screen, and highlight a few of its key features.

Despite tiptop's advantages, we identified a number of limitations. As one example, the Pentium III supports 80 hardware counter events, but can track at most two in parallel [4]. However, the default tiptop screen assumes that the underlying architecture can track five hardware counters events. In addition, we found that building XML files was cumbersome and platform-specific. To compound this problem, tiptop exposes less than 20% of the hardware performance counters available across platforms. In many cases, tracking a hardware counter not currently supported by tiptop requires modification and recompilation of tiptop's C source files.

In light of these limitations, we identified two feature enhancements for tiptop. First, we enhanced tiptop to display the number of threads per-process as a default feature. Then, as a substantial feature enhancement, we integrated the cross-platform Performance Application Programming Interface (PAPI) [2] library into tiptop as an alternative API to access hardware counters. (This is backwards-compatible change, and use of the PAPI API is optional when creating an XML configuration file.) Integration with PAPI increases the breadth of platforms and kernel versions supported by tiptop. Now, tiptop can run with kernel versions prior to 2.6.31, via a PAPI-supplied kernel patch. In addition, through software multiplexing, PAPI enables the number of hardware counter

[1] `erven.rohou@inria.fr`

[2] `http://sourceforge.net/projects/unixtop/`

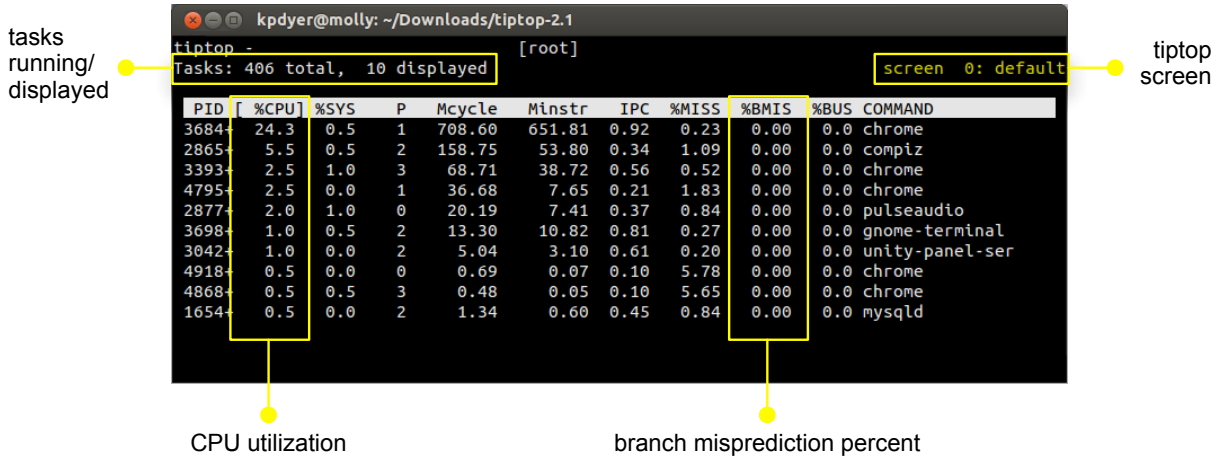

Figure 1: A screenshot of the default screen for tiptop version 2.1. Each row in the display represents a single process. In the top-left corner we have the number of running and displayed processes. By default, tiptop only displays processes that have non-zero CPU utilization. In the bottom-left we highlight the CPU utilization, reported as percentage of cycles used since the last screen refresh. In the bottom-right corner we highlight the number of branch mispredictions. Finally, in the top-right we highlight the label for the current screen. Multiple views with custom columns can be configured and flipped through (using the right and left arrow keys) in real-time.

events to exceed the hard limit imposed by the hardware. Finally, the integration with PAPI simplifies the process of creating cross-platform XML configuration files for tiptop hardware counters can be specified by using PAPI-specific macros, by abstracting architecture-specific configuration options from the user.

In addition to feature enhancements, we identified two tiptop bugs, and confirmed them through personal communications with lead developer Erven Rohou. Both bugs prevent tiptop from correctly displaying hardware counter statistics under specific conditions. For the first bug, we identified a solution that minimizes the probability that the bug occurs, and will work with Erven Rohou to include our patch into the main branch of the tiptop source. The second bug remains unresolved, but can be avoided by running tiptop as root.

In section 2 we give an overview of our high-level development cycle and the tools we used to repair and enhance tiptop. In section 3 we give further details about our feature enhancements and bugfixes. We conclude in section 4 with a discussion about additional feature improvements for future versions of tiptop.

2 Methodology

In our development we started with tiptop version 2.1, which (optionally) depends upon the libxml2[3] and ncurses[4] libraries. Tiptop is written in C [1] and all our improvements and bugfixes were implemented in C, too. As a key feature enhancement, we integrated the PAPI [2] library into tiptop. In our testing we used Python[5] and C to automate the process of verifying bug fixes and

[3] http://www.xmlsoft.org/
[4] http://www.gnu.org/software/ncurses/
[5] http://www.python.org/

feature enhancements.

Fortunately, tiptop has an integral feature called *batch mode*, which enables tiptop to output statistics to stdout, rather than the default real-time display. The batch mode feature greatly simplified the testing process and enabled us to easily extract process statistics from tiptop.

Roughly, we can describe our methodology as a three-step process.

1. Identify feature or bug.
2. Develop code to enhance or fix tiptop.
3. Use Python (and sometimes C) to develop unit tests that execute tiptop, then parse and verify the output, and ensure that the enhancement produces expected output, or the bug is resolved.

In many cases we face the challenge of knowing what the "correct" statistics are for an application. As an example, we are not aware of a trivial way to know the correct value for the number of threads for a Chrome browser process. In such cases we constructed applications with known values, such as a simple C program that spawns N threads, and verified that tiptop reported the values correctly for our simple, and deterministic, application.

3 Results

In this section we start with an overview of our feature enhancements. Then, we follow with a discussion of the bugs we identified in tiptop.

3.1 Feature Enhancements

Roughly, we enhanced tiptop with two new features. First, we enhanced the default screen to include the number of threads per process. Then, we added into tiptop the the cross-platform Performance Application Programming Interface (PAPI) [2] library, this enables a new, and more robust way, for tiptop to access performance counters. We will discuss each of these feature in turn.

3.1.1 Thread count on main screen

The primary focus for tiptop is to enable a user to view hardware counters in real-time. However, there are other variables that can indirectly influence hardware performance. As an example, say a developer wants to investigate the effect of inter-thread cache contention on the performance of their application. Especially in cases when the number of threads in a process is not deterministic, it would be valuable to have the thread count on the tiptop screen. Therefore, we enabled tiptop to display the number of threads per process, and we included this feature on the default screen.

In our implementation we added hooks to the XML configuration logic to enable the ability to display the number of threads per process. This involved adding code at multiple layers, including a call that exposed thread count from tiptops's data-gathering layer to the layer that handles presentation logic. In addition, tiptop has multiple modes, which influence the filter that is applied to the processes/threads to be displayed. As an example, tiptop can be run with the `-H` flag, in order to show per-thread information, rather than per-process. In such a case we do not want to display thread count, and our implementation includes logic such that thread count only appears in the case when we filter by processes.

In the testing of this feature we did the following, using Python and C.

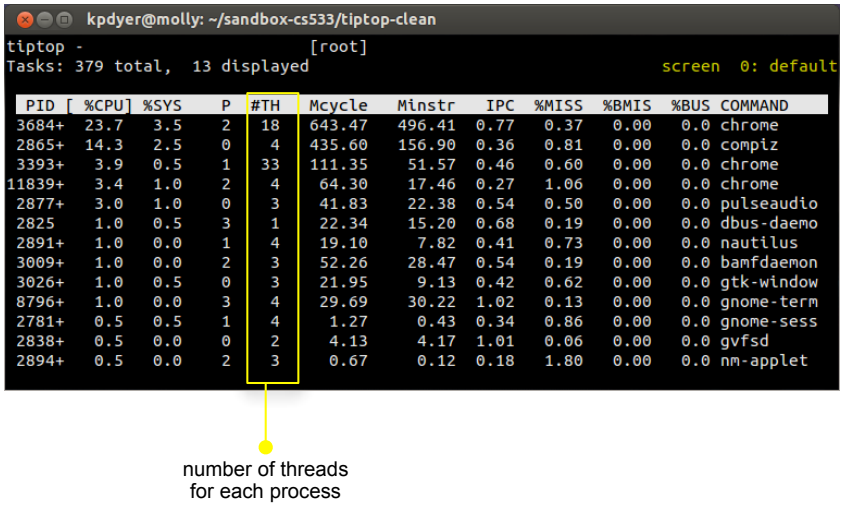

Figure 2: A screenshot of the enhanced tiptop homescreen, showing the number of threads in each process.

1. Execute a C program that spawns N threads.
2. Start tiptop in batch mode.
3. Execute tiptop, parse the output and verify that tiptop displays N threads for the long-running C process.

Our tests passed on Ubuntu 12.04 executing a simple C program that invokes threads through the pthreads library, for values $N \in \{1, 10, 100\}$. In Figure 2 we have a screenshot of the enhanced default screen with the thread-count column, `#TH`, highlighted.

3.1.2 Integration with the PAPI library

As our next feature for tiptop, we integrated support for the Performance Application Programming Interface (PAPI) library. The PAPI library is a cross-platform API for accessing hardware performance counters. Roughly, it is a interface that removes the complexity involved in accessing hardware performance counters.

Consider the following concrete example for why integration with PAPI enhances tiptop. The man page for tiptop gives the following example for a way to manually specify a counter that records the number of issued mico-ops on Sandy Bridge:

```
<counter alias="uops_issued"
         config="0x010e"
         type="RAW"
         arch="x86"
         model="06_2A" />
```

Then we must construct the column on the tiptop screen, too, which references the `uops_issued` counter:

```
<column header=" U Ops"
```

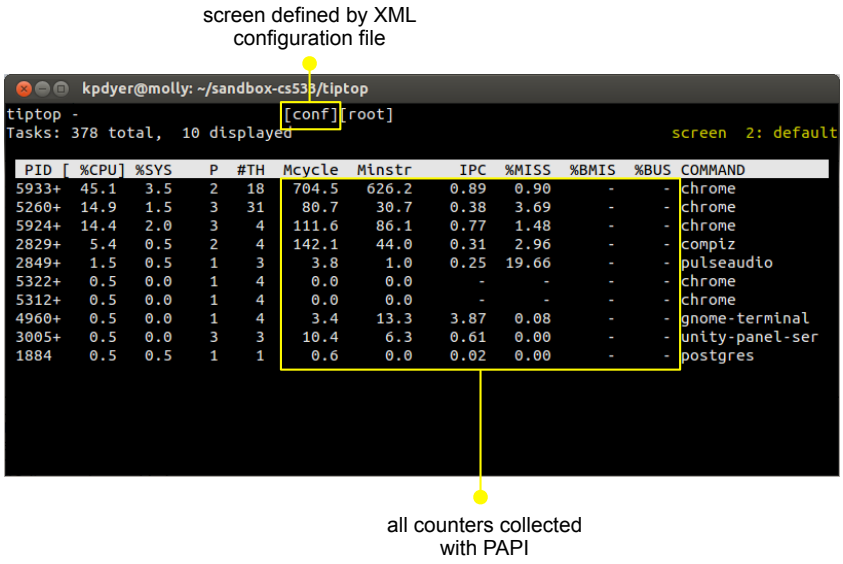

Figure 3: The tiptop home screen implemented using the PAPI API. One side effect of using PAPI is that zero-values appear as - on the tiptop output. This was an inadvertent side-effect, but makes the screen more readable.

```
        format="%5.1f"
        expr="uops_issued" />
```

The values for `config` and `model` in the `counter` tag are hardware-specific, and require that the user identifies the appropriate values in architecture-specific documentation in order to implement an alias, which can then be used to construct a column in a custom tiptop screen. Indeed, it is less than ideal that a user must specify magic numbers for their CPU. What is more, this is not portable, an alias for `uops_issued` on a different platform will require different `config` and `model` values. This is certainly a nightmare when developing a tiptop screen that may be used across multiple platforms.

Fortunately, PAPI makes life much easier. Instead of having to specify a hardware-specific counter macro, then reference the macro for display, our new feature enables a user to specify the PAPI event directly. There are 103 counter events specified in PAPI 5.0.1, and our integration with PAPI enables a user specify the PAPI-defined macro. As an example, the following declaration would be able to integrate directly into the tiptop XML file

```
<column header=" U Ops"
        format="%5.1f"
        expr="PAPI_FP_OPS" />
```

It is then up to PAPI to determine the correct hooks for a hardware counter, and the burden is not on the user to identify magic numbers. This is cross-platform, as far as PAPI supports the specific counter specified. Hence, the user does not have to specify a custom hardware-specific `counter` macro, and a single XML file can be specified and deployed across multiple system. In addition, this is backwards compatible with previous versions of tiptop configurations files, and the prefix `PAPI_` for an `expr` attribute in a `column` tag, is a flag for tiptop to call PAPI, rather than using a direct kernel call for hardware counters.

field	default expression	PAPI expression
Millions of cycles (Mcycle)	`delta(CYCLE) / 1000000`	`PAPI_TOT_CYC / 1000000`
Millions of instructions (Minstr)	`delta(INSN) / 1000000`	`PAPI_TOT_INS / 1000000`
Instructions per Cycle (IPC)	`delta(INSN) / delta(CYCLE)`	`PAPI_TOT_INS / PAPI_TOT_CYC`
Instruction cache miss % (%MISS)	`100 * delta(MISS) / delta(INSN)`	`100 * PAPI_L1_ICM / PAPI_TOT_INS`
Branch misprediction % (%BMIS)	`100 * delta(BR) / delta(INSN)`	`100 * PAPI_BR_MSP / PAPI_TOT_INS`

Figure 4: The expression changes required in the tiptop XML configuration file in order to use PAPI, instead of the default direct call to the Linux kernel. The previous expression still work, and the new PAPI macros simply extend the range of hardware counters that can be accessed from tiptop.

In addition to a cleaner interface for accessing hardware counters, PAPI implements multiplexing for hardware counters, which enables a user to specify more counters that are permitted by the hardware. As previously mentioned, if the underlying architecture is a Pentium III, tiptop is limited to two hardware counter events. However, PAPI implements multiplexing via high-precision timeslice sharing. This means that hardware counters are rotated, by default, every 100ms such that values can be obtained for all requested counters. If we, say, specify four hardware events to monitor, in a 200ms window PAPI would monitor two hardware events for the first 100ms, then rotate to monitor the other two hardware events for the remaining 100ms.

Multiplexing does present limitations, if a user wanted to display a value such as instructions per second, this would require additional logic in order to know the amount of time that a specific counter was active. We will leave this edge-case as future work.

As an additional feature, integration with PAPI enables tiptop to support a broader range of kernels. Currently, tiptop supports Linux kernels 2.6.31 and above. PAPI supports Linux kernels 2.6.31 and above out of the box, and supports kernels 2.6.30 and below with a kernel patch. Tiptop is not compatible with the PAPI patch for pre-2.6.31 kernels, and would require further modification to work for pre-2.6.31 kernels. Hence, the PAPI integretation enables a greater range of kernel support without directly modifying tiptop.

Challenges in integrating PAPI with tiptop. When integrating PAPI with tiptop, we encountered a number of challenges. First, PAPI's documentation was nearly non-existent when describing its ability to multiplex and monitor external processes. A correct implementation required extensive reading and searching through the PAPI mailing lists. In addition, understanding tiptop's workflow and the correct functions to call to release the hardware counters via PAPI took some effort, too.

Case Study: Building the tiptop default home screen with PAPI. In order to demonstrate the flexibility of our PAPI integration and its correctness, we implement the default tiptop screen using PAPI, instead of tiptop's default behavior of using direct kernel calls. In Figure 3 we have a screenshot of the tiptop homescreen, implemented by building an XML file that references PAPI macros. We first notice that the top of the screen the `[conf]` flag, which indicates that the tiptop screen is specified by the an XML configuration file.

In Figure 4 we have a listing of the current tiptop expression used to build the default screen, and the alternative expression that can now be used to access counter via PAPI. These expressions can be mixed and matched within the same XML file.

As an example of another benefit of PAPI exposed by building the home screen, the tiptop interface exposes `MISS` (Figure 4) as `PERF_COUNT_HW_CACHE_MISSES`, which in the Linux kernel references

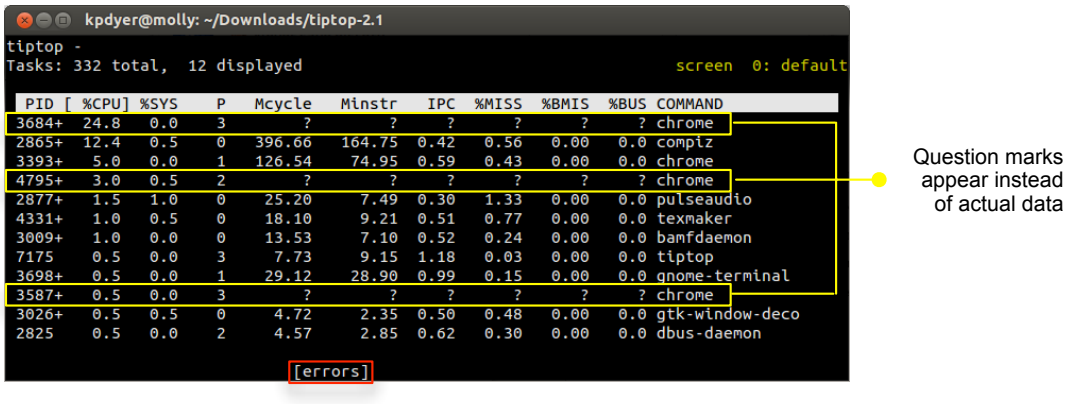

Figure 5: A screenshot of the manifestation of the two bugs identified in our development. When tiptop is unable to determine the values associated with a process it ouputs a question mark instead. As we can see, tiptop failed to gather statistics for three of the four Chrome processes in the above screenshot, at the bottom of the screen, in the red box, tiptop reports that `[errors]` occurred.

either L1 or L2 Instruction Cache misses[6]. Depending upon your kernel version, for %MISS in Figure 4 with return either Level 1 or Level 2 instruction cache misses. However, it is clear from the PAPI macro that we are returning the Level 1 instruction cache miss. If we, say, wanted to report Level 2 instruction cache misses instead, we simply replace `PAPI_L1_ICM` with `PAPI_L2_ICM`. Alternatively, if we wanted, instead, the Level 3 data cache miss, we could use the macro `PAPI_L3_DCM`.

3.2 Bug Fixes

In our initial attempt to understand tiptop and its range of features we encountered an anomaly. In some cases a question mark would appear, instead of a value from the hardware performance counter. This would happen consistently for programs such as the Chrome[7] web browser, and would happen sporadically for other applications. In Figure 5 we have a screenshot that highlights this bug. Upon investigation, we identified two separate issues that lead to this error condition.

Bug 1: Too many open file descriptors. First, a bit of background. Linux systems include a `limits.conf` file which enables a systems administrator to have fine-grained control over the system resources used by users and groups. As an example, it is possible to specify the maximum priority of a process run by a specific user or group, or even the maximum number of processes spawned by a specific user or group. Each resource has a *hard* and *soft* limit. Hard limits are enforced by the kernel and can only be modified by superusers. Soft limits are default values and can be manually raised per-user, up to the hard limit. For example, Ubuntu 12.04 has a soft limit of 1024 open file descriptors per user, and a hard limit of 4096. This means that a regular user can open up to 1024 file descriptors under normal circumstances, or can open up to 4096 file descriptors by explicitly requesting (to the kernel) for the soft limit to be increased to 4096.

In order to read hardware counters, tiptop opens five counters per process in its default configuration. For each combination of process and hardware counter, a file descriptor is opened. As we can can see in the screenshot for Figure 5, my desktop had 332 running processes. Hence,

[6] `https://lkml.org/lkml/2010/11/1/131`
[7] `http://www.google.come/chrome`

we now have a problem. We have 332 processes, five hardware counters per process, so we have exceeded our soft limit for the number of file handlers open by requesting to open $332 \cdot 5 = 1660$ file descriptors.

As a stopgap solution to this problem, we explicitly request for our soft file descriptor limit to be increased to the hard limit. This may be performed by fetching the hard limit from `/proc/N/limits`, then making a call to the `setrlimit` function, which is provided by `sys/resource.h`. Unfortunately, increasing the open file descriptor limit beyond the hard limit requires superuser permission. Our solution, raising the soft limit to the hard limit, is consistent with other open source project that have encountered this issue, such as Wine[8]. A more robust solution for this bug is an open problem and would require fundamental changes to the design of tiptop, we talk about possible solutions in section 4.

The testing for this was not straightforward due to a different bug, which we will discuss next. In order to verify our fix for this bug we ran tiptop as root, with root having a hard number of file descriptor set to 4096 and soft set to 1024. (By default, root has a soft limit of 4096 and hard limit of 8192 on Ubuntu 12.04.) We may then run the version of tiptop that uses the soft limit and we get the following output.

```
19884+   4.6   0.0    0        ?        ?     ?      ?      ?     ? chrome
...
19818+   0.5   0.5    0        ?        ?     ?      ?      ?     ? firefox
```

Simultaneously, we run the enhanced version of tiptop that explicitly requests a file descriptor limit to be raised to the hard limit, and we get the following output.

```
19884+   4.3   0.0    0 19327.35 19327.35  1.00   0.00   0.00   0.0 chrome
...
19818+   0.5   0.5    0  8589.93  8589.93  1.00   0.00   0.00   0.0 firefox
```

We verify by that this bug is solved by checking that no value in the tiptop output has a question mark. A more robust testing strategy will require an enhanced error reporting layer for tiptop, which distinguishes between different types of hardware counter read failures. However, an enhanced error reporting layer is beyond the scope of this project.

Bug 2: `EACCES`, permission denied. A second cause of "questions marks" in the tiptop output has been traced to an `EACCES` permission denied error returned by the kernel, and is isolated to only a few applications, such as Chrome and the gnome-keyring-daemon, to name a few. This condition only occurs when running tiptop as a regular user. We contacted the lead developer of tiptop, Erven Rohou, in order to determine the cause of this problem. This was an unknown bug, but it can be avoided by running tiptop as root. Unfortunately, we have no further progress towards a solution to this bug.

4 Future Work

Tiptop is a useful application, and our extensions further enhance its utility. However, aside from the unresolved bug, there are four major points that require further investigation.

First, our solution to the "Too many open file descriptors bug" is unfortunately not comprehensive. On systems that have more than 819 processes and a hard file descriptor limit of 4096, this

[8] `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/663090`

will still be an issue. We do not believe that it is reasonable to require superuser permissions to solve this problem. Therefore, as one solution, it may be possible to develop a heuristic algorithm that only opens a file descriptor for processes of interest. As an example the algorithm can create a hardware counter (i.e., open a file descriptor) for any process that has used the CPU in the last N seconds. Of course, this will require process monitoring that does not use hardware performance counters. However, we believe that the overhead of such a strategy is justified, in order to minimize the probability of this bug occurring.

Second, we found that tiptop is not well-supported in virtualized environments. The man page briefly mentions this, but does not elaborate further. Our testing confirms that data is often omitted or not reported correctly in VM instances. We believe this requires further investigation to understand why this is the case. Certainly, for testing purposes, the ability to test tiptop without requiring physical hardware would greatly increase the ability to verify that features are correctly implemented across CPU architectures.

Third, further investigation is require to understand if tiptop needs superuser access.

Finally, our integration with PAPI opens tiptop to a much wider range of architectures and kernel versions. The tiptop application would benefit from a more robust testing framework that is able to validate the correctness of performance counter. As an example, it may be possible to create low-level assembly programs that execute a deterministic number of instructions. Then, this testing framework should be deployed and executed on a wide variety of configurations, in order to test the extensibility of tiptop.

References

[1] Kernighan, B. W. *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[2] Mucci, P. J., Browne, S., Deane, C., and Ho, G. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference* (1999), pp. 7–10.

[3] Rohou, E. Tiptop: Hardware Performance Counters for the Masses. Rapport de recherche RR-7789, INRIA, Nov. 2011.

[4] Weaver, V., and McKee, S. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (sept. 2008), pp. 141 –150.