L4oprof: A System-Wide Profiler Using Hardware PMU in L4 Environment

Jugwan Eom, Dohun Kim, and Chanik Park

Department of Computer Science and Engineering
Pohang University of Science and Technology
Pohang, Gyungbuk 790-784, Republic of Korea
{zugwan,hunkim,cipark}@postech.ac.kr

Abstract. The recent advance of L4 microkernel technology enables building a secure embedded system with comparable performance to a traditional monolithic kernel-based system. According to the different system software architecture, the execution behavior of an application in microkernel environment differs greatly from that in traditional monolithic environment. Therefore, we need a performance profiler to improve performance of the application in microkernel environment. Currently, L4's profiling tools provides only program-level information such as the number of function calls, IPCs, context switches, etc. In this paper, we present L4oprof, a system-wide statistical profiler in L4 microkernel environment. L4oprof leverages the hardware performance counters of PMU on a CPU to enable profiling of a wide variety of hardware events such as clock cycles and cache and TLB misses. Our evaluation shows that L4oprof incurs 0~3% higher overhead than Linux OProfile. Moreover, the main cause of performance loss in L4Linux applications is shown compared with Linux applications.

Keywords: L4 microkernel, performance analysis, performance measures, performance monitoring, statistical profiling, hardware PMU.

1 Introduction

To analyze a program's performance, its execution behavior is investigated by monitoring runtime information. Monitoring program execution is important, because it can find the bottlenecks and determine which parts of a program should be optimized. The type of collected information depends on the level at which it is collected. It is divided into the following two levels.

1. The program level: the program is instrumented by adding calls to routines which gather desired information such as the number of time a function is called, the number of time a basic block is entered, a call graph, and an internal program state like queue length changes in the kernel block layer.
2. The hardware level: the program does not need to be modified. CPU architecture like caches, pipelines, superscalar, out-of-order execution, branch prediction, and speculative execution can lead to large differences between the best-case and the

Y.-H. Lee et al. (Eds.): ICESS 2007, LNCS 4523, pp. 548–559, 2007.

worst-case performances, which must be considered to increase program performance. Most of current CPUs have a hardware component called PMU (Performance Monitoring Unit) for programmers to exploit the information on CPU. The PMU measures the micro architectural behavior of the program such as the number of clock cycles, how many cache stalls, how many TLB misses, which is stored in performance counters.

The goal of performance analysis is to find where time is spent and why it is spent there. Program-level monitoring can detect performance bottlenecks, but finding their cause is best solved with hardware-level monitoring, which may result from function calls, algorithmic problems, or CPU stalls. Therefore, the two levels of monitoring must be used complementarily for proper performance analysis.

The recent advance of L4 microkernel technology enables building a secure embedded system with comparable performance to a traditional monolithic kernel-based system. Industry such as QUALCOMM sees L4 microkernel's potential as a solution to the security problems of embedded systems. In a microkernel-based system, the basic kernel functions such as communication, scheduling, and memory mapping are provided by the microkernel and most of OS services are implemented as multiple user level servers on top of the microkernel, in which the execution behavior of an L4 application differs greatly from that in traditional monolithic environment. Therefore, we need a performance profiler to improve performance of the application in microkernel environment. However, L4 microkernel provides only program-level profiling tools which do not utilize the PMU information that is also valuable for fine-grained performance profiling, which enables to locate the cause of performance inefficiency in an application.

In this paper, we present L4oprof, a system-wide statistical profiler in L4 microkernel environment. L4oprof leverages the hardware performance counters of PMU on a CPU to enable profiling of a wide variety of hardware events such as clock cycles and cache and TLB misses without program modification. L4oprof can profile applications in the system and the L4 microkernel itself. This paper also shows the main cause of performance loss in L4Linux applications compared that in Linux applications. L4oprof has been modeled after the OProfile [5] profiling tool available on Linux systems.

The remainder of the paper is organized as follows. Section 2 describes related work. We describe the aspects of L4 and OProfile as background for our work in Section 3. Section 4 describes the design and implementation of L4oprof. Then, we present L4oprof's performance in Section 5. Finally, we summarize the paper and discuss the future work in Section 6.

2 Related Work

Several hardware monitoring interfaces [8, 9] and tools [6, 7, 10, 15] have been defined for using hardware performance monitoring on different architectures and platforms. Xenoprof [15] is a profiling toolkit for the Xen virtual machine environment, which has inspired the current L4oprof's approach.

In L4 microkernel-based environment, a few performance monitoring tools are available. Fiasco Trace Buffer [11] collects the kernel internal events such as context

switches, inter process communications, and page faults. It is attached with the kernel and configured via L4/Fiasco kernel debugger. rt_mon [14], GRTMon [12] and Ferret [13] are user-space monitoring tools, which provide a monitoring library and sensors that store the collected information. Currently, existing monitoring tools in L4 environment only use program level information via instrumentation and cannot help pinpoint problems in how software uses the hardware features. L4oprof extends the profiling abilities of L4, allowing hardware level performance events to be counted across all system activities. Utilizing the information collected from the PMU enables any application to be profiled without any modification.

3 Background

In this section, we briefly describe the L4 microkernel based environment and the OProfile for statistical profiling on Linux.

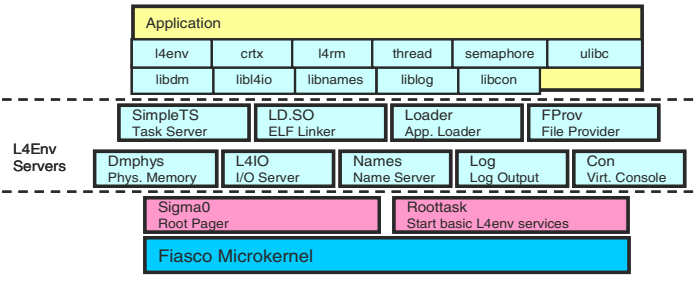

Fig. 1. L4 microkernel-based Environment

3.1 L4 Microkernel-Based Environment

L4 is a second generation microkernel ABI [2]. Our work uses the Fiasco [17] microkernel-based environment. As shown in Figure 1, this environment consists of a set of cooperating servers running on top of the microkernel. All servers use the kernel-provided synchronous interprocess communication mechanism for communication.

L4Env [4] is a programming environment for application development on top of the L4 microkernel family. It provides a number of servers and libraries for OS services such as thread management, global naming, synchronization, loading tasks, and resource management. L4Linux [3] is a para-virtualized Linux running on top of the microkernel using L4Env, which is binary-compatible with the normal Linux kernel. The Linux kernel and its applications run as a server in user mode and system calls from Linux applications are translated into IPC to the L4Linux server. Current L4Linux is unable to configure some features such as ACPI, SMP, preemption, APIC/IOAPIC, HPET, highmem, MTRR, MCE, power management and other similar options in the Linux.

3.2 OProfile

OProfile [5] is a low-overhead system-wide statistical profiler for Linux included in the 2.6 version of the kernel. The Linux kernel supports OProfile for a number of different processor architectures. OProfile can profile code executing at any privilege level, including kernel code, kernel modules, user level applications and user level libraries.

OProfile can be configured to periodically take samples to obtain time-based information for indicating which sections of code are executed on the computer system. Many processors include a dedicated performance monitoring hardware component called PMU (Performance Monitoring Unit), which allows to detect when certain events happen such as clock cycles, instruction retirements, TLB misses, cache misses, branch mispredictions, etc. The hardware normally takes the form of one or more counters that are incremented each time an event takes place. When the counter value "rolls over," an interrupt is generated, making it possible to control the amount of detail (and therefore, overhead) produced by performance monitoring. OProfile uses this hardware to collect samples of performance-related data each time a counter generates an interrupt. These samples are periodically written out to disk; later, the statistical data contained in these samples can be used to generate reports on system-level and application-level performance.

OProfile can be divided into three sections: the kernel support, the daemon, and the sample database with analysis programs. The kernel has a driver which controls the PMU and collects the samples. The daemon reads data from the driver and converts it into a sample database. The analysis programs read data from the sample database and present meaningful information to the user.

4 L4oprof

In this section, we describe the design and implementation of L4oprof which we have developed for the L4 microkernel based environment. The L4oprof has similar capabilities to OProfile. It uses the hardware PMU to collect periodic samples of performance data. The performance of applications running in L4 environment depends on interactions among the servers for OS services, for example, the L4Linux server in case of the Linux application, and the L4 microkernel. In order to achieve a proper performance analysis, the profiling tool must be able to determine the distribution of performance events across all system activities.

Figure 2 shows an overview of the L4oprof. At an abstract level, the L4oprof consists of a L4 microkernel layer which services performance-counter interrupts and an OProfile server layer which associates samples with executable images, and merges them into a nonvolatile profile database and a modified system loader and other mechanisms for identifying executable images. As shown in Figure 2, the L4oprof reuses the OProfile code and extends its capabilities to be used in the L4 environment instead of starting from scratch. The remainder of this section describes these pieces in more detail, beginning with the L4 microkernel layer.

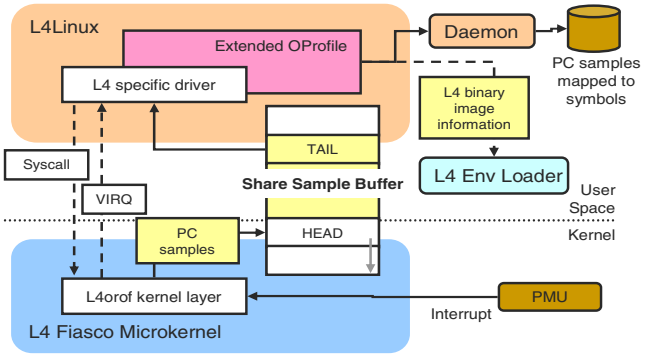

Fig. 2. L4oprof overview

4.1 L4 Microkernel Layer

The L4 microkernel layer of the L4oprof uses the same hardware counter based sampling mechanism as used in OProfile. It defines a user interface which maps performance events to the physical hardware counters, and provides a system call for OProfile server layer to setup profiling parameters, start and stop profiling. L4oprof can be configured to monitor the same events supported by OProfile.

When a performance counter overflows, it generates a high-priority interrupt that delivers the PC of the next instruction to be executed and the identity of the overflowing counter. When the interrupt handler in the L4 kernel layer handles this interrupt, it records the sample that consists of the task identifier (L4 Task ID) of the interrupted process, the PC delivered by the interrupt, and the event type that caused the interrupt. The L4 kernel layer hands over the PC samples collected on counter overflow to the OProfile server layer for further processing. PC samples are delivered via a shared sample buffer synchronized with a lock-free method in order to support high-frequency performance counter overflow interrupt and reduce profiling overhead. Next, the L4 kernel layer notifies the OProfile server layer of generating a sample via a virtual interrupt. However, if current user process is not the Oprofile server, it delays executing the virtual interrupt handler in OProfile server layer, resulting in L4oprof's poor performance. Therefore, we separated actual data delivering from counter overflow notification.

4.2 OProfile Server Layer

The OProfile server layer extracts samples from the shared sample buffer and associates them with their corresponding images. The data for each image is periodically merged into compact profiles stored as separate files on disk.

The OProfile server layer operates in a manner mostly similar to its operation in OProfile on Linux. For low level operations, such as accessing and programming performance counters, and collecting PC samples, it is modified to interface with the L4 microkernel layer of L4oprof. The high level operations of the OProfile server in the layer remain mostly unchanged.

Architecture-specific components in OProfile are newly implemented to use the L4 kernel layer virtual event interface. The interrupt thread in L4Linux waits until the L4 kernel layer signals a generating sample. After copying PC samples from the shared sample buffer, the OProfile server layer determines the routine and executable image corresponding to the program counter on each sample in the buffer. In case of Linux kernel and applications, this is determined by consulting the virtual memory layout of the Linux process and Linux kernel, which is maintained in the L4Linux kernel server. Since PC samples may also include samples from the L4 kernel address space, the OProfile server layer is extended to recognize and correctly attribute L4 kernel's PC samples. In order to determine where the images of other L4 applications including the L4Env servers are loaded, the L4Env loaders are modified. There are two loaders in L4Env: the Roottask server starts applications using boot loader scripts according to Multi Boot Information [18] and the Loader server supports dynamic loading. A modified version of each loader handles requests from the OProfile server layer. The response from the loader contains the L4 task ID, the address at which it was loaded, and its file system pathname.

The OProfile server layer stores samples in an on-disk profile database. This database structure is the same as the Linux OProfile's database structure. Thus, post-profiling tools in OProfile can be used without any modification.

5 Evaluation

The profiling tools must collect many thousands of samples per second yet incur sufficiently low overhead so that their benefits outweigh their costs. Profiling incurs performance slowdown of applications compared to the performance of applications without profiling. In this section, we summarize the results of experiments designed to measure the performance of our profiler. The hardware used for evaluation was a Pentium 4 1.6GHz processor with 512MB of RAM, and Intel E100 Ethernet controller. For the comparison, OProfile in Linux 2.6.18.1 is used. Table 1 shows the workloads used.

Table 1. Description of Workloads

Workload	Description
Multiply	Multiplies two numbers by using two different methods in the loop, CPU-bound
Tar	Extracts Linux kernel source, Real workload
Iperf [16]	Measures network bandwidth, IO-bound

5.1 Profiling Test and Verification

Prior to presenting the performance overhead, we demonstrate that L4oprof works correctly. We tested and verified our profiling tool under time-biased configuration, i.e. GLOBAL POWER EVENTS event that is the time during which the processor is not stopped, in which L4oprof monitors the clock cycles by running the Multiply workload. This configuration generates the distribution of time spent in various

routines in Multiply workload. We compared the profiling result of L4oprof to that of OProfile and GNU gprof. GNU gprof enables us to know the exact number of times a function is called using the instrumentation; it is enabled using the `-pg` option of GNU cc. Figure 3 shows the result which each profiler has produced. All profilers have a similar distribution of time spent in three functions, `main`, `slow_multiply`, and `fast_multiply`.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 99.05      3.14     3.14  4000000   785.00   785.00  slow_multiply
  0.63      3.16     0.02                             main
  0.32      3.17     0.01  4000000     2.50     2.50  fast_multiply
```

(a)

```
CPU: P4 / Xeon, speed 1513.55 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is
not stopped) with a unit mask of 0x01 (mandatory) count 100000
samples  %        image name               symbol name
824571   99.3596  multiply                 slow_multiply
4920      0.5929  multiply                 main
395       0.0476  multiply                 fast_multiply
```

(b)

```
CPU: P4 / Xeon, speed 1513.55 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is
not stopped) with a unit mask of 0x01 (mandatory) count 100000
samples  %        image name               symbol name
827170   99.4093  multiply                 slow_multiply
4533      0.5448  multiply                 main
391       0.0458  multiply                 fast_multiply
```

(c)

Fig. 3. Comparison of profiling result from the three profiles (a) GNU gprof (b) OProfile (c) L4oprof

5.2 Profiling Performance

The accuracy and overhead of the profiling can be controlled by adjusting the sampling interval (i.e., the count when notifications are generated). We evaluated our profiling tool's performance in terms of profiling overhead caused by handling overflow interrupts. To measure the overhead, we ran each workload at least 10 times varying the frequency of overflow interrupts. The PMU was programmed to monitor GLOBAL POWER EVENTS event, because it basic event type. We compared the profiling overhead of L4oprof to that of OProfile. OProfile is well known for its low overhead and is one of the most popular profilers for Linux.

Figures 4, 5, and 6 show the profiling overhead of L4oprof and OProfile under running Multiply, Tar, Iperf workload respectively. L4oprof incurs 0~3% higher profiling overhead than OProfile under the default settings which cause around 15000, 13050, and 750 interrupts per seconds with Multiply, Tar, and Iperf workload respectively. These figures also indicate that the performance of L4oprof is more sensitive than that of OProfile to the frequency of the generated interrupts since the overhead gap between L4oprof and OProfile is increasing as the interrupt frequency increases.

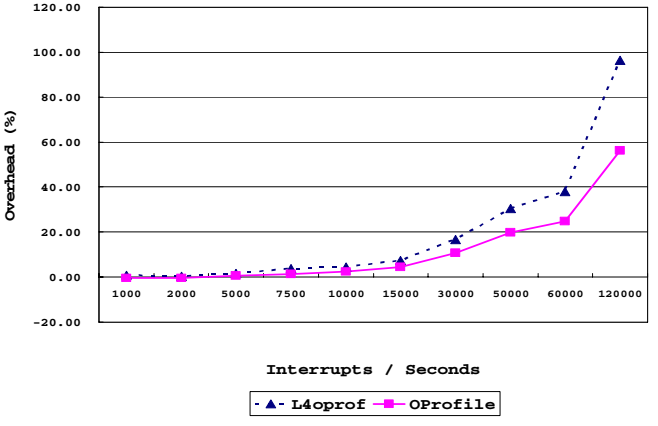

Fig. 4. Profiling overhead, L4oprof vs. OProfile (Multiply)

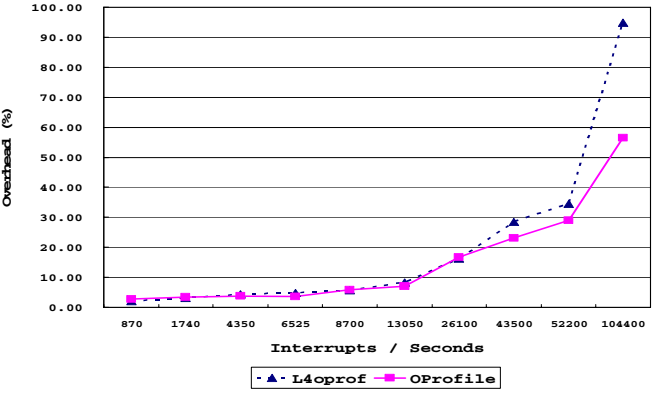

Fig. 5. Profiling overhead, L4oprof vs. OProfile (Tar)

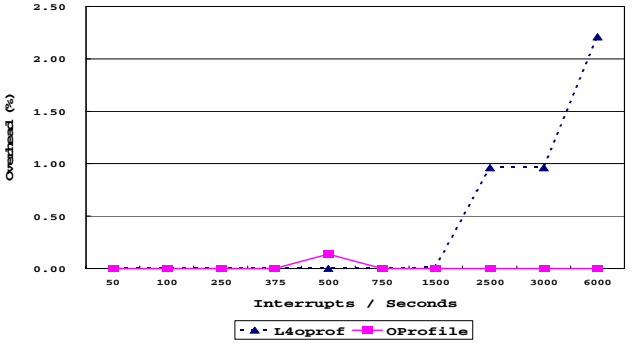

Fig. 6. Profiling overhead, L4oprof vs. OProfile (Iperf)

5.3 The Cause of Profiling Overhead

There are two main components to L4oprof's overhead. First is the time to service performance-counter interrupts. Second is the time to read samples and merge the samples into the on-disk profiles for the appropriate images. To investigate the cost of the first component, we gathered the number of cycles spent in the interrupt handlers of L4oprof and OProfile. We believe that the second component does not contribute to the overhead gap between L4orpof and OProfile, since the operation of the second component in L4oprof was not changed greatly from that in OProfile. In order to optimize performance, however, it must be fine-tuned, which will be our future work. During handling performance-counter interrupts, OProfile directly accesses the performance counters and collects the PC samples itself. But in the case of L4oprof, this operation is divided into the interrupt handler in the L4 microkernel and the virtual interrupt handler in the L4Linux kernel server of the OProfile server layer. Therefore, we read the Time Stamp Counter (TSC) values increased each clock signal at the beginning and the end of each interrupt handler and used the difference as the elapsed cycles for the interrupt service. We collected a total of 2327 samples.

Figure 7 shows that L4oprof's interrupt service time and its variance are larger than OProfile. As Figure 8 shows, the virtual interrupt handler in the L4Linux kernel server causes large variance of time to interrupt service in L4oprof, which is why L4oprof has lower performance than OProfile when the interrupt is generated more frequently. If the interrupt frequency is becoming higher and higher, L4oprof can become more easily overloaded with counter interrupts than OProfile.

Finally, we must answer the question, "why this happen in L4oprof?" This is mainly caused by running Linux in user mode on top of the microkernel. For L4Linux to receive a virtual interrupt IPC, context switching may be needed. And, in the middle of handling an interrupt, the L4Linux kernel server can be preempted. This phenomenon cannot occur in normal Linux. In other words, the current performance overhead gap between L4oprof and OProfile is neither caused by L4oprof' design nor its implementation being defective.

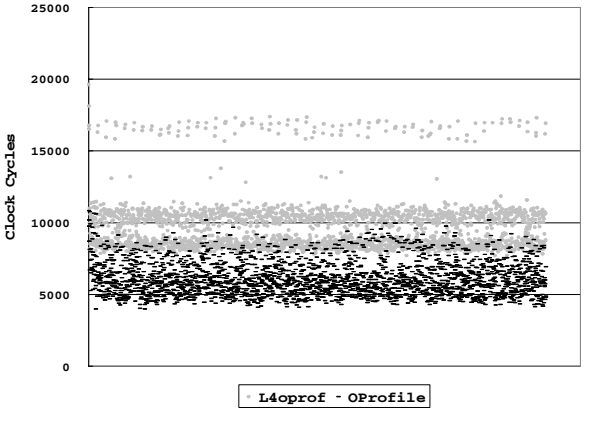

Fig. 7. The clock cycle distribution for interrupt service, L4oprof vs. OProfile

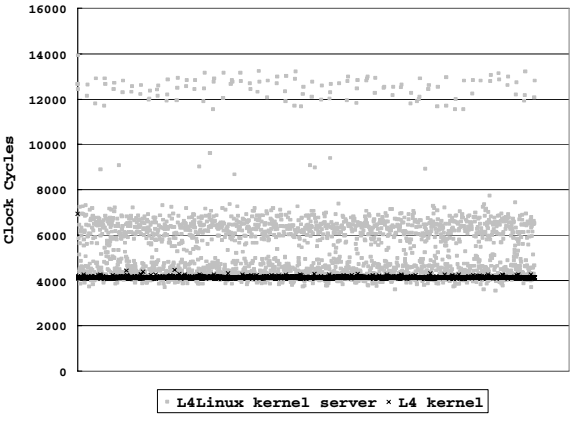

Fig. 8. The components of clock cycle distribution for interrupt service in L4oprof

5.4 Performance Analysis of L4Linux Applications Using L4oprof

An application's performance in L4linux differs from its performance in normal Linux. Figure 9 compares the performance of the L4Linux and Linux for Multiply and Tar workloads.

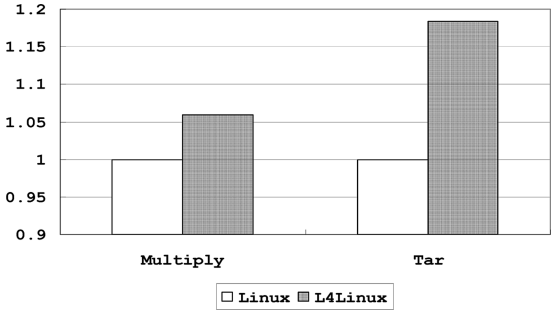

Fig. 9. Relative user application's performance on L4linux and Linux

In L4linux, an application's performance is about 5~17% slower than Linux. We profile the Linux and L4linux, and compare the aggregate hardware event counts in the two configurations for the following hardware events: instruction counts, L2 cache misses, data TLB misses, and instruction TLB misses. Figure 10 and 11 show the normalized values for these events relative to the Linux numbers.

Figures 10 and 11 show that L4linux has higher cache and TLB miss rates compared to Linux. Because the L4 kernel considers L4Linux's applications as user process, each L4Linux system call leads to at least two context switches, i.e., one context switch from the application to the L4Linux server and the other context switch back to the application, resulting in more TLB flushes and performance degrade. It is the main cause of performance degrades in L4Linux.

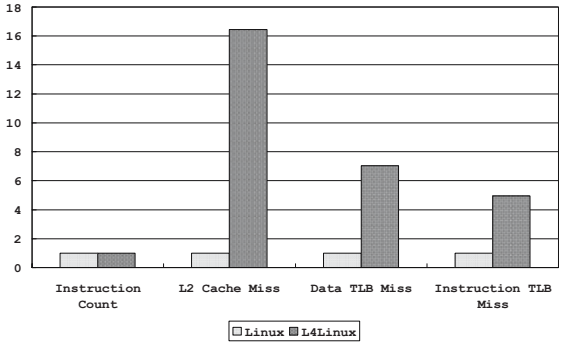

Fig. 10. Relative hardware event counts in L4Linux and Linux for Multiply workload

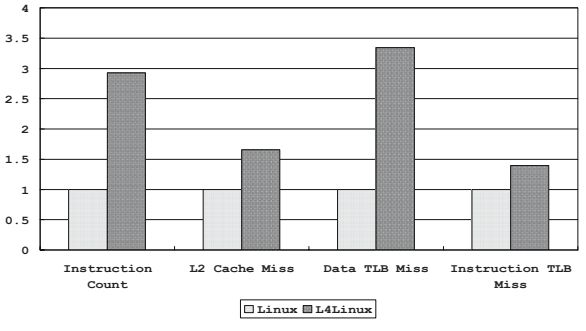

Fig. 11. Relative hardware event counts in L4Linux and Linux for Tar workload

6 Conclusions

In this paper, we presented L4oprof, a system-wide statistical profiler in the L4 microkernel environment. L4oprof leverages the hardware performance counters of PMU on a CPU to enable profiling of a wide variety of hardware events such as clock cycles and cache and TLB misses. It is the first performance monitoring tool using the hardware PMU in an L4 microkernel based environment. We reused the OProfile in Linux and extended its capabilities to be used in the L4 environment instead of starting from scratch. L4oprof can help in building a L4 microkernel-based secure embedded system with good performance.

Our evaluation shows that L4oprof incurs 0~3% higher overhead than Linux OProfile, depending on sampling frequency and workload. We discovered that the major overhead of L4oprof is caused by running Linux in user mode on top of the microkernel. Moreover, it is also shown that profiling user applications running on L4Linux by L4oprof allows locating the main cause of performance loss compared to the same applications running on Linux.

Currently, L4oprof only supports Intel Pentium 4 CPU. We will port it to additional CPU modes such as ARM/Xscale and AMD64. Supporting multiple

L4Linux instances is also part of our future work. To reduce the performance overhead gap compared to OProfile, we will find an alternative structure, for example, moving the operations in the current OProfile server layer into the L4 microkernel layer, to eliminate the user-mode running effects.

Acknowledgement

This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (HY-SDR Research Center).

References

1. Intel. IA-32 Architecture Software Developer's Manual. Vol 3. System Programming Guide, 2003
2. J. Liedtke. L4 reference manual (486, Pentium, PPro). Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.
3. Adam Lackorzynski. L4Linux Porting Optimizations. Master's thesis, TU Dresden, March 2004.
4. Operating Systems Group Technische Universitat Dresden. The l4 environment. http://www.tudos.org/l4env.
5. J. Levon. OProfile. http://oprofile.sourceforge.net.
6. J. M. Anderson, W. E. Weihl, L. M. Berc, J. Dean, S. Ghemawat. Continuous profiling: where have all the cycles gone? In ACM Transactions on Computer Systems, 1997
7. Intel. The VTuneTM Performance Analyzers. http://www.intel.com/software/products/vtune.
8. S. Eranian. The perfmon2 interface speciation. Technical Report HPL-2004-200(R.1), HP Labs, Feb 2005.
9. M. Pettersson. The Perfctr interface. http://user.it.uu.se/mikpe/linux/perfctr
10. ICL Team University of Tennessee. PAPI: The Performance API.,http://icl.cs.utk.edu/papi/index.html.
11. A. Weigand. Tracing unter L4/Fiasco. Grober BelegTechnische Universitat Dresden, Lehrstuhl fur Betriebssysteme, 2003.
12. T. Riegel. A generalized approach to runtime monitoring for real-time systems. Diploma thesis, Technische Universitat Dresden, Lehrstuhl fur Betriebssysteme, 2005.
13. M. Pohlack, B. Dobel, and A. Lackorzynsiki. Towards Runtime Monitoring in Real-Time Systems. In Eighth Real-Time Linux Workshop, October 2006
14. M. Pohlack. The rt_mon monitoring framework, 2004.
15. A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In First ACM/USENIX Conference on Virtual Execution Environments, June 2005
16. The University of Illinois. Iperf. http://dast.nlanr.net/Projects/Iperf.
17. Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002. I, 2
18. Free Software Foundation, Multiboot Specification, http://www.gnu.org/software/grub/manual/multiboot/multiboot.html