# L4oprof: a performance-monitoring-unit-based software-profiling framework for the L4 microkernel.

Jugwan Eom
SK Telecom
**2** PUBLICATIONS **2** CITATIONS
SEE PROFILE

Chanik Park
Pohang University of Science and Technology
**59** PUBLICATIONS **275** CITATIONS
SEE PROFILE

# L4oprof: A Performance-Monitoring-Unit-Based Software-Profiling Framework for the L4 Microkernel [1]

Dohun Kim
Department of Computer Science and Engineering, POSTECH
Pohang, Gyungbuk 790-784,
Republic of Korea
+82-54-279-5668
hunkim@postech.ac.kr

Jugwan Eom
Department of Computer Science and Engineering, POSTECH
Pohang, Gyungbuk 790-784,
Republic of Korea
+82-54-279-5668
zugwan@postech.ac.kr

Chanik Park
Department of Computer Science and Engineering, POSTECH
Pohang, Gyungbuk 790-784,
Republic of Korea
+82-54-279-2248
cipark@postech.ac.kr

## ABSTRACT

These days, the L4 microkernel is expanding its domain towards embedded systems since it is showing a comparable performance with traditional monolithic kernels. The L4 microkernel shows a greatly different execution behavior of user applications from that in a traditional monolithic environment because most operating-system services are run as user-level applications. Therefore, we need a profiling framework to obtain a better understanding of performance bottlenecks for software optimization. However, current L4 profiling tools provide only higher-level information, such as the number of function calls, IPCs, and context switches. In this paper, we present a software profiling framework which gathers system-wide statistical information in the L4 microkernel environment. In order to support profiling lower-level information such as clock cycles, cache misses, and TLB misses, our profiling framework uses the hardware performance counters of the PMU (Performance Monitoring Unit) which most CPUs support. In this paper, we show that our profiling framework incurs less than 3 % overhead below 15000 interrupts per second compared to the existing Linux profiling tool. Moreover, as a case study, we show the main cause of performance loss in L4Linux applications compared with Linux applications.

## 1. INTRODUCTION

These days, the L4 microkernel is expanding its domain towards embedded systems because it is showing a comparable performance with traditional monolithic kernels. For example, QUALCOMM adopts the L4 microkernel as its operating system because it can solve security and performance problems of mobile embedded systems. In the L4 microkernel system, most operating-system services are run as user-level applications on top of the microkernel's core-kernel service, resulting in showing a greatly different execution behavior of user applications from that in a traditional monolithic environment. Therefore, we need a profiling framework to obtain a better understanding of performance bottlenecks for software optimization. However, the current L4 microkernel provides only program-level profiling tools which do not utilize the PMU (Performance Monitoring Unit) information that is also valuable for fine-grained performance profiling and analyzing the cause of performance inefficiency in an application.

There has been much research on analyzing the application's execution behavior by monitoring the runtime information [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Each approach tries to monitor program execution because it can analyze the performance bottlenecks, which is valuable for software optimization. Each approach can be summarized into two levels based on the gathering location:

**The program level:** The program is explicitly instrumented by adding function calls which gather desired information such as the number of time a function is called, the number of time a basic block is entered, a function call graph, and an internal program state like queue length changes in the kernel block layer.

**The hardware level:** All performance information is gathered by the hardware without the modification of program source code. This approach collects CPU specific events such as cache operations, pipelines, superscalar, out-of-order execution, branch prediction, and speculative execution which must be considered to optimize program performance. These days, most CPUs support a hardware unit called PMU in order for software developers to exploit the information gathered by the CPU. The PMU usually helps to measure the micro-architectural behavior such as the number of clock cycles, cache stalls, TLB misses, and cache hits/misses which are stored in performance counters.

In order to understand a program's behavior correctly, software profiling tools must not only analyze program-level information but also hardware-level information. Sometimes, a program's performance degradation may result from complex causes like algorithmic problems with frequent CPU stalls. In such case, program-level monitoring helps to detect performance bottlenecks and hardware-level monitoring helps to find causes of bottlenecks. Therefore, the two levels of monitoring must be used complementarily for proper performance analysis.

A microkernel-based system consists of multiple components such as L4 microkernel, L4Env servers, L4Linux server, and user applications. Since multiple components can be related to execute a single user application in this system, it is critical to be able to profile multiple components for performance optimization in user applications. Therefore, an L4-based profiling tool must have the capability of aggregated monitoring.

In this paper, we present a software profiling framework which gathers system-wide statistical information, and our

[1] An earlier version of this paper has been presented in International Conference on Embedded Software and Systems, Taegu, Korea, 2007.

framework is designed and implemented by using L4/Fiasco 1.2 and L4Linux 2.6.18. Our profiling framework has been modeled after the OProfile [5] profiling tool available on Linux systems. In order to support profiling lower-level information such as clock cycles, cache misses, and TLB misses, our profiling framework uses the PMU hardware performance counters which most CPUs support without program modification. Currently, our profiling framework can profile applications and the L4 microkernel itself with less than 3 % overhead below 15000 interrupts per second. As a case study, this paper shows the main cause of performance loss in L4Linux applications which results from frequent L2 cache misses, data TLB misses, and instruction cache misses compared with traditional Linux applications.

The remainder of this 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. Next, we present L4oprof's performance in Section 5. As a case study, we investigate the performance degradation in L4Linux applications in Section 6. Finally, we summarize the paper and discuss future work in Section 7.

## 2. Related Work

Several hardware monitoring interfaces [8, 9] and tools [6, 7, 10, 15] have been developed to support hardware performance monitoring on different architectures and platforms. Especially, perfmon2 [8] supports unified PMU configuration and data register regardless of CPU architecture. Xenoprof [15], which inspired our approach, is a profiling toolkit expanded from Oprofile [5] for the Xen virtual machine environment.

In the L4 microkernel environment, a few performance monitoring tools are developed. Fiasco Trace Buffer [11] collects the kernel-internal events such as context switches, inter process communications (IPC), and page faults. It can configure events to be monitored via the L4/Fiasco kernel debugger, and we can check the gathered information via kernel debugger or user space. rt_mon [14], GRTMon [12] and Ferret [13] are user-space monitoring tools which provide libraries and sensors to store the collected information. Currently, existing monitoring tools in the L4 environment use program-level information via instrumentation. Therefore, it cannot detect hardware-related problems. Our profiling framework extends the profiling abilities of the L4 microkernel to monitor hardware-level performance events across all system components such as L4 microkernel, L4Env servers, L4Linux server, and user applications, which enables any application to be profiled without any modification.

## 3. Background

To help in obtaining a better understanding of our work, we briefly describe OProfile and the L4 microkernel in this section.

### 3.1 Oprofile

OProfile [5], included in Linux kernel 2.6, is a profiling tool, which measures system-wide statistics with low overhead. It can profile software components such as kernel, kernel modules, user-level applications, and user-level libraries. Moreover, it already has been ported to various CPU types.

According to the functionality, Oprofile can be divided into three sections: the kernel driver, the user daemon, and analysis utilities. The kernel driver can be divided into two sections: lower layer and higher layer. The lower layer controls the CPU-specific PMU and collects the samples. The higher layer is in charge of maintaining PMU configuration and providing interface with the user daemon. The user daemon reads the sampled data from the kernel driver and stores it into a sample database, and the sample database can be maintained by profiling a session. The analysis programs read data from the sample database and present meaningful information to the user.

OProfile can be configured to periodically obtain samples. Conventional CPUs include a PMU which allows detecting when certain events occur, such as clock cycles, instruction retirements, TLB misses, cache misses, branch mispredictions, and so on. In general, the PMU has one or more counters that are incremented with each clock cycle or an event taking place. When the counter value "rolls over," that is, it overflows, a PMU interrupt is generated. OProfile gathers event samples by using PMU interrupts. The gathered samples are periodically written out to disk, and the statistical data of these samples is used to generate reports on system-level or application-level performance.

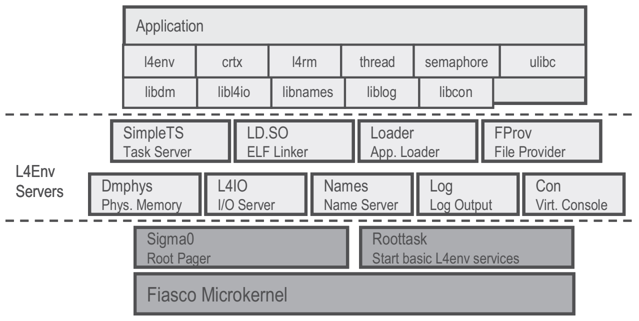

**Figure 1.** The L4 microkernel Environment

### 3.2 The L4 microkernel environment

We adapted the Fiasco [17] microkernel environment, one of the second generation microkernel implementations [2]. The Fiasco environment consists of the microkernel itself and kernel service applications (servers) that run on top of the microkernel. All servers use the synchronous IPC mechanism provided by the L4 microkernel for communication.

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

## 4. L4oprof: Profiling Framework

In this section, we describe the design and implementation of our profiling framework based on the L4 microkernel.

Figure 2 shows an overview of L4oprof. The L4Linux server associated with the operation of L4oprof can be regarded as a special L4 server running on top of L4 since it is configured not to support any user applications other than profiling; in other words, current L4oprof is L4 native and does not depend on L4Linux. Moreover, it can profile L4-native applications.

A microkernel-based system consists of multiple components such as L4 microkernel, L4Env servers, L4Linux server, and user applications. This means that multiple components will be related to execute a single user application in this system. Therefore, it is critical to be able to profile multiple components for performance optimization in user applications.

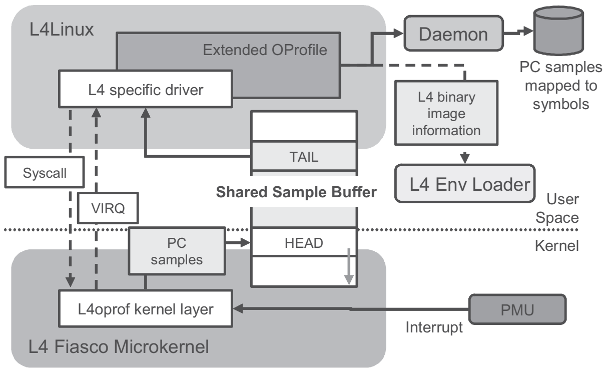

**Figure 2. L4oprof overview**

L4oprof consists of two layers: L4 kernel layer and Oprofile server layer. The L4 kernel layer maintains PMU events and PMU interrupts while the OProfile server layer associates samples with executable images and merges them into a nonvolatile profile database. To identify each application's binary image, the system loader and other mechanisms are modified. L4oprof reuses OProfile's code and extends its capabilities to be used in the L4 environment instead of starting from scratch.

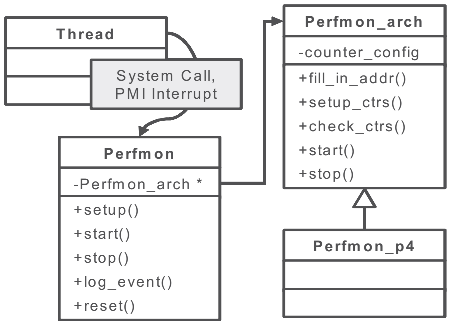

**Figure 3. L4oprof kernel layer structure**

### 4.1 L4 kernel layer

The L4 kernel layer uses the same sampling mechanism used in Oprofile. It defines a user interface which maps performance events to the physical hardware counters, and provides a system call for the OProfile server layer to set up profiling parameters and start and stop profiling. L4oprof can be configured to monitor the same events supported by OProfile. Figure 3 shows a Pentium4 case of the L4 kernel layer. Perfmon is a CPU-independent interface, which provides a unified interface to configure PMU events. Perfmon_arch maintains a CPU-specific interface and information, whereas the Perfmon_p4 contains Pentium4-related information.

Whenever a performance counter overflows, a PMU interrupt is generated. Then, the PMU interrupt delivers the PC (Program Counter) with the overflowed event information. Next, the interrupt handler (Thread in Figure 3) in the L4 kernel layer handles this interrupt. It records the sample that contains current task's identifier (L4 Task ID), PC value, and event type that caused the interrupt. Finally, the L4 kernel layer delivers the gathered samples to the OProfile server layer for further processing. The gathered samples are delivered via a shared sample buffer synchronized with a lock-free method in order to support a high sampling frequency. In order to profile the behavior of an application correctly, the sampling frequency should be set to a high value, which means that it can incur many virtual interrupts, resulting in many L4 IPC messages. In order to notify the OProfile server layer that new events have been sampled, each virtual interrupt's L4 IPC must be transmitted. Whenever a virtual interrupt is called, the sampled data should be copied from the L4 side of L4oprof (L4oprof kernel layer) to the L4Linux side of L4oprof (L4oprof driver layer). Therefore, the L4 IPC overhead is reduced by using a shared buffer which contains the samples. In fact, the L4oprof kernel layer needs each virtual interrupt's L4 IPC without payload to notify the L4oprof driver layer whenever the events are sampled. Therefore, no IPC has been omitted. However, omitting IPCs may be our future work to obtain better performance. Figure 4 shows how the L4 kernel layer's shared sample buffer is mapped into L4Linux's address space.

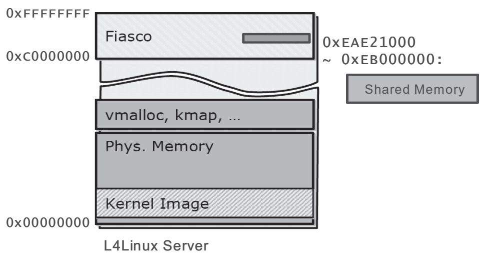

**Figure 4. Address Space of L4Linux and Shared Sample Buffer**

### 4.2 OProfile server layer

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

The OProfile server layer operates in a manner mostly similar to OProfile on Linux. According to the functionality, the OProfile server layer can be divided into three detailed layers: L4Linux

OProfile driver layer, L4Linux user daemon (oprofiled [5]), and extended L4 binary loaders.

The L4Linux user daemon which is in charge of high-level operations of the OProfile server remains mostly unchanged.

The L4Linux OProfile driver layer is in charge of maintaining access to PMU such as programming performance counters and collecting PC samples. To support access to PMU, each access operation is transformed into an L4 microkernel system call. Figure 5 shows the event buffer structure of the OProfile driver layer which contains the event samples from the L4 kernel layer. This structure is similar to that of Linux OProfile except that it uses a shared sample buffer to extract the event samples. The CPU buffer is used for storing event samples during the virtual-interrupt handling from the L4 kernel layer. After the virtual-interrupt handling, the CPU buffer is merged into the event buffer, and the event-buffer data is delivered to the user daemon. Figure 5 shows special events such as task switch, mode switch, and dcookies. According to the current CPU mode, the task switch and mode switch events are detected in following situations: L4Linux kernel/user mode, Fiasco kernel mode, and L4 user mode. Dcookies are used for identifying each application's binary image and each binary image's file path or memory address to retrieve executable file's symbol table.

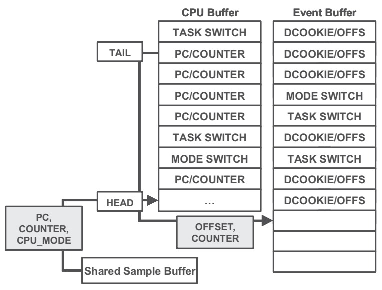

**Figure 5. L4Linux OProfile server buffer and events**

In the L4Linux OProfile driver layer, architecture-specific components are newly implemented to use the L4 kernel layer as a virtual event interface. The interrupt thread in the L4Linux waits for the L4 kernel layer signals to inform of sample events. After copying PC samples from the shared sample buffer, the OProfile server layer determines the executable image which corresponds to the program counter on each sample in the buffer. In case of Linux kernel and applications, it 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 the L4 kernel's PC samples.

In the L4 microkernel system, L4 applications and L4Env servers are loaded before L4Linux is loaded. Therefore, in order to determine where the binary images of other L4 applications and L4Env servers are loaded, the L4Env loaders are modified. There are two application loaders in L4Env: Roottask and Loader. The Roottask server starts applications by using boot loader scripts according to Multi Boot Information [18] and the Loader server supports dynamic loading. A modified version of each loader handles the request for identifying the binary image path from the OProfile server layer. The response from the loader contains a corresponding L4 task ID, the address at which it was loaded, and a file system pathname.

The OProfile server layer stores samples in an on-disk profile database. Since the database structure is the same as that of the Linux OProfile, the profile analysis tools of the Linux OProfile can be used without any modification.

## 5. Experimental results

Since our profiling framework gathers many samples within a short time interval, it can accompany much profiling overhead. Therefore, profiling may cause performance slowdown of applications compared to the performance of applications without profiling. In this section, we present the experimental results which explain the performance of our framework. Our experiments are conducted by using a hardware platform containing Pentium 4 1.6GHz processor, 512MB of RAM, and Intel E100 Ethernet controller. For a fair comparison of our profiling framework and Linux OProfile, L4Linux 2.6.18 and Linux 2.6.18 are used as our Linux environments. For each experiment, we used the multiply, tar, and Iperf utilities as our workload. Table 1 describes each workload.

**Table 1. Description of Workloads**

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

### 5.1 Profiling accuracy

To ensure our framework's profiling accuracy, several experiments are conducted. Each workload in Table 1 is investigated with each of the following profiling tools: GNU gprof, Linux OProfile, and our profiling framework. In fact, the difference between gprof and Oprofile is beyond our scope. Moreover, the gprof cannot measure PC samples accurately since its sampling frequency is fixed to 10 milliseconds. Our goal is to develop a profiling tool for an L4-based system that provides the same level of capabilities as Oprofile, with a reasonable amount of overhead. However, we present gprof's result since gprof is one of the representative profiling tools. For Linux Oprofile and our profiling framework, we configure each profiling tool to monitor a time-biased event, the GLOBAL POWER EVENTS, which indicates the time during which the processor is not stopped, and the frequency is set to 10000 counts, which correspond to 6.6 microseconds. Linux OProfile and our profiling framework monitor the clock cycles by running the each workload and generate the distribution of time spent in various routines in each workload. We compare these results with GNU gprof. GNU gprof enables us to know each PC sample per 10 milliseconds and the exact number of times that a function is called using the instrumentation by using the -pg option of the GNU C compiler.

Figure 6 shows the Multiply result which each profiling tool has generated. The Multiply is a simple program without system calls. Therefore, Figure 6 shows that all profiling tools have similar time distribution in three functions; main, slow_multiply, and fast_multiply.

Since the Tar workload contains many function symbols, Figure 7 shows only a few of the symbols. GNU gprof shows a significantly different result compared to Linux Oprofile and our profiling framework. Since it has the coarse-grained sampling frequency of 10 milliseconds, its time-biased samples are inaccurate. However, Linux OProfile and our profiling framework can have a fine-grained sampling frequency and show a similar result. Therefore, our profiling framework shows comparable accuracy to the Linux Oprofile.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 98.43    470.88   470.88 40000000    11.77    11.77  slow_multiply
  1.12    476.22     5.34                             main
  0.45    478.38     2.17 40000000     0.05     0.05  fast_multiply
```

(a)

```
CPU: P4 / Xeon, speed 1513.59 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is
not stopped) with a unit mask of 0x01 (mandatory) count 10000
samples  %        symbol name
25103158 98.5019  slow_multiply
279080    1.0950  main
102708    0.4030  fast_multiply
```

(b)

```
CPU: P4 / Xeon, speed 1513.47 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is
not stopped) with a unit mask of 0x01 (mandatory) count 10000
samples  %        symbol name
25869146 98.5253  slow_multiply
272946    1.0395  main
114251    0.4351  fast_multiply
```

(c)

**Figure 6. Profiling result of Multiply workload containing no system call (a) GNU gprof (b) Linux OProfile (c) L4oprof**

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 36.39      6.27     6.27                             write
 16.37      9.09     2.82                             unlink
 13.76     11.46     2.37                             read
  9.23     13.05     1.59                             open64
  4.53     13.83     0.78                             utimes
  2.32     14.23     0.40                             chown
  2.03     14.58     0.35                             close
  1.86     14.90     0.32                             chmod
  1.10     15.09     0.19   128685     0.00     0.00  from_header
  0.81     15.23     0.14                             memcpy
  0.75     15.36     0.13    21450     0.00     0.00  tar_checksum
```

(a)

```
CPU: P4 / Xeon, speed 1513.59 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is
not stopped) with a unit mask of 0x01 (mandatory) count 10000
samples  %        symbol name
3960     2.2709   write
1539     0.8826   unlink
7905     4.5333   read
3877     2.2233   open64
2212     1.2685   utimes
2743     1.5730   chown
2034     1.1664   close
2284     1.3098   chmod
10398    5.9629   from_header
4102     2.3524   memcpy
12801    7.3410   tar_checksum
```

(b)

```
CPU: P4 / Xeon, speed 1513.47 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is
not stopped) with a unit mask of 0x01 (mandatory) count 10000
samples  %        symbol name
1479     2.2558   write
396      0.6040   unlink
2863     4.3667   read
792      1.2080   open64
778      1.1866   utimes
1182     1.6441   chown
664      1.0128   close
594      0.9060   chmod
3812     5.8142   from_header
1507     2.2985   memcpy
4556     6.9489   tar_checksum
```

(c)

**Figure 7. Profiling result of Tar workload containing system calls (a) GNU gprof (b) Linux OProfile (c) L4oprof**

## 5.2 Profiling performance

The performance of Linux OProfile and our profiling framework depends on the sampling frequency, that is, PMU interrupts. We evaluate the Linux OProfile and our profiling framework in terms of profiling the overhead caused by handling PMU interrupts. To measure the overhead, each workload is run at least 10 times by varying the frequency of overflow interrupts. The PMU is configured to monitor a time-biased event, GLOBAL POWER EVENTS, since it is a basic event type. Each profiling tool's overhead is calculated and compared with the result of non-profiling tool case and the profiling overhead of L4oprof is also compared with that of Linux OProfile.

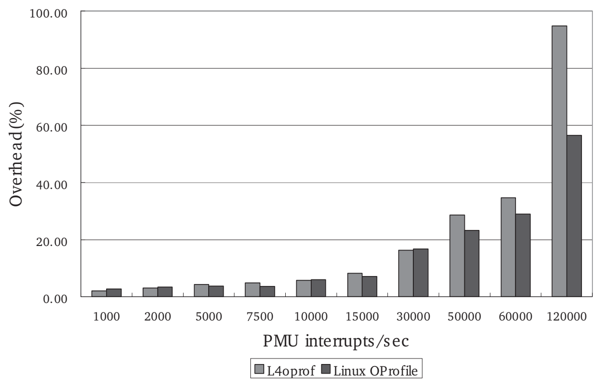

**Figure 8. Overhead comparison: L4oprof vs. Linux OProfile: Multiply**

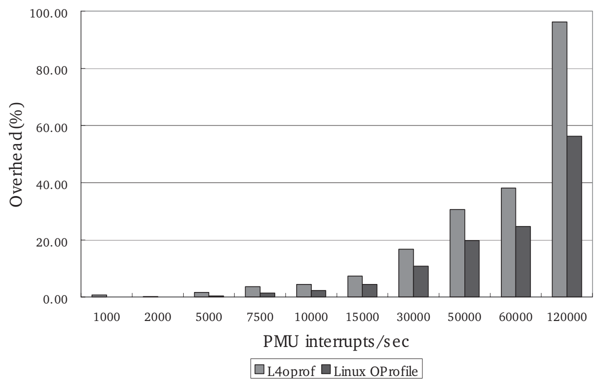

**Figure 9. Overhead comparison, L4oprof vs. Linux OProfile: Tar**

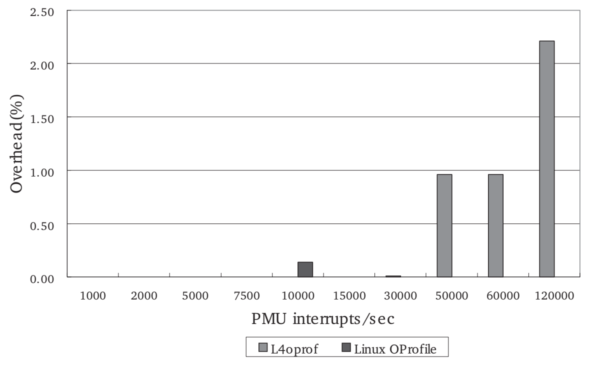

**Figure 10. Overhead comparison, L4oprof vs. Linux OProfile: Iperf**

Figures 8, 9, and 10 respectively show the overhead comparison between Linux OProfile and our profiling framework by varying the PMU interrupt rate under Multiply, Tar, and Iperf workloads. As a result, our profiling framework shows less than 3 % higher overhead than the Linux OProfile, below 15000 interrupts per second corresponding to 66 microseconds which is a very short sampling frequency. However, these figures also indicate that our profiling framework is more sensitive to the sampling frequency. We believe that it results from PMI interrupt handling overhead which transfers interrupt and sampled data form the L4 kernel layer to the L4Linux OProfile driver layer via shared sample buffer, as shown in Figures 11, 12, and 13.

Figures 11, 12, and 13 show each system component's portion of samplings. Each sampling is measured by using the same evaluation method of Figures 8, 9, and 10. Each figure shows the profiling overhead results from the time of the L4/Fiasco's core kernel.

## 5.3 Main component of the profiling overhead

In our profiling framework, there are two main causes of showing the current overhead. First is the PMU interrupt handling overhead which transfers interrupt context and sampled events to the L4Linux OProfile driver layer. Second is processing and storing the sampled event data into the disk for appropriate images. To investigate the first component, the number of cycles spent in the interrupt handlers of Linux OProfile and our profiling framework is gathered respectively. We believe that the second component does not contribute to the overhead gap of Linux OProfile and our profiling framework because the second component's operations are not different between Linux OProfile and our profiling framework. However, there is still a chance to optimize the performance in the second component's operations, which will be our future work. In the Linux OProfile case, whenever a PMU interrupt occurs, the Linux OProfile can directly access the performance counters and collect the PC samples by itself. This is a significant benefit compared with our profiling framework. However, in our framework, this operation is divided into two operations: PMU interrupt handling in the L4 kernel layer and transferring interrupt context and sampled data via virtual-interrupt handler in the L4Linux OProfile driver layer. To measure this overhead, we read the Time Stamp Counter (TSC) values at the beginning and the end of each interrupt handler and use the difference as the elapsed cycles for the interrupt service. To measure overhead, a total of about 2300 samples are collected.

Figure 14 shows that our profiling framework's interrupt service time and its variance are larger than the Linux OProfile. The virtual-interrupt handler in the L4Linux server is the reason of large time variance during interrupt handling as shown in Figure 15. This data can also explain why our profiling framework has lower performance than the Linux OProfile when the interrupt is generated more frequently (Figures 8, 9, 10). If the interrupt frequency becomes higher and higher, then our profiling framework becomes more easily overloaded with PMU interrupts. Table 2 shows the absolute values and statistics of Figures 14 and 15.

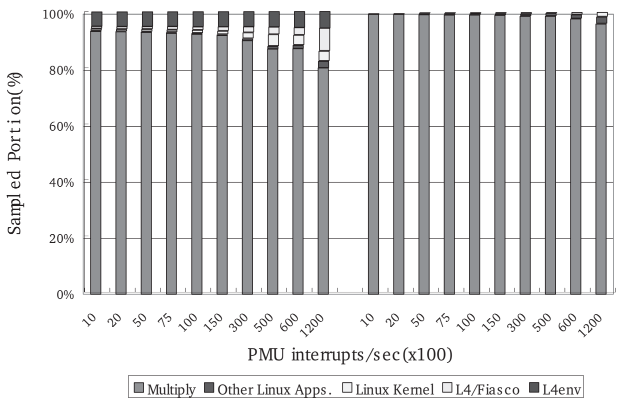

**Figure 11. Each component's portion of samplings, L4oprof (left) vs. Linux OProfile (right): Multiply**

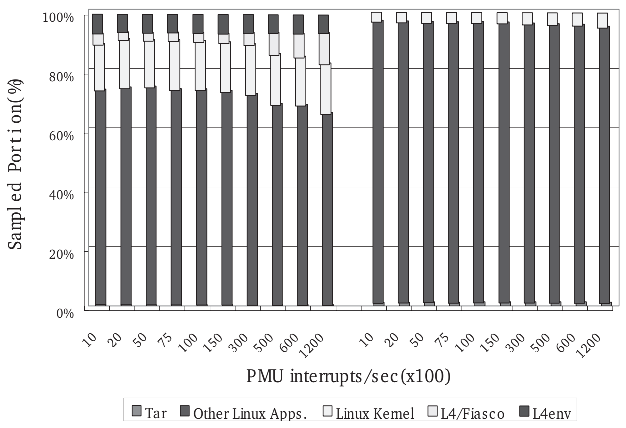

**Figure 12. Each component's portion of samplings, L4oprof (left) vs. Linux OProfile (right): Tar**

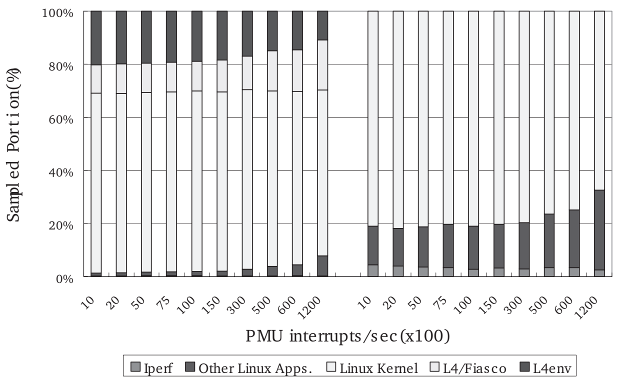

**Figure 13. Each component's portion of samplings, L4oprof (left) vs. Linux OProfile (right): Iperf**

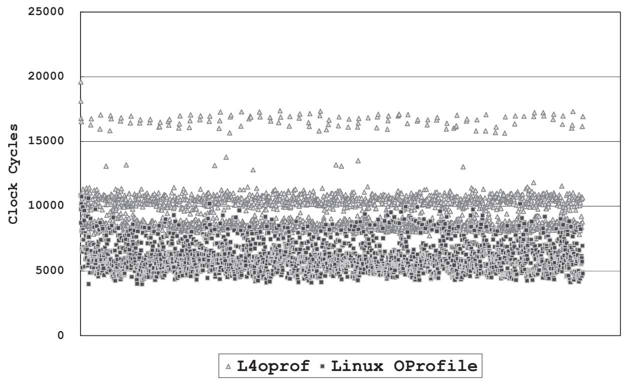

**Figure 14. The clock cycle distribution for interrupt service, L4oprof vs. Linux OProfile**

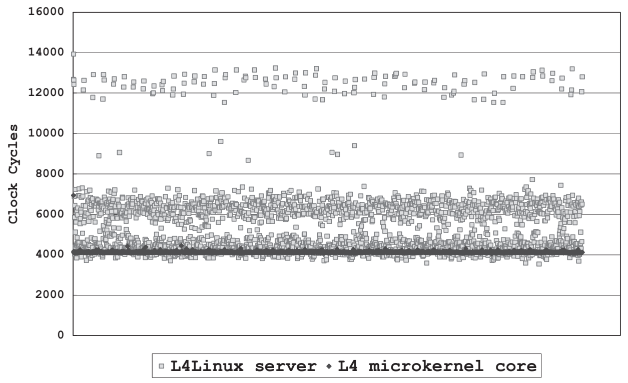

**Figure 15. The components of clock cycle distribution for interrupt service in L4oprof**

**Table 2. Interrupt handling overhead**

| | L4 microkernel core | L4Linux server | L4oprof | Linux Oprofile |
|---|---|---|---|---|
| Avg.clocks | 4135.10 | 5660.07 | 9795.17 | 6073.44 |
| Std. Dev. | 63.86 | 1920.25 | 1925.57 | 1170.40 |
| Min | 4064 | 3548 | 7668 | 3980 |
| Max | 6940 | 13924 | 19608 | 10852 |

## 6. Case study: Performance analysis of L4Linux applications

In the previous section, we showed that our profiling framework incurs more overhead than the Linux OProfile during interrupt handling because it runs the L4Linux server in user mode on top of the microkernel. In order for L4Linux to receive a virtual-interrupt IPC, a context switching may be needed. Moreover, 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 Linux OProfile and our profiling framework is neither caused by our design nor its implementation. Instead, it results from the L4 microkernel's characteristic.

To show an application's performance difference between L4Linux OProfile and normal Linux, we measured the performance of Multiply and Tar workload (Table 1). Figure 16 shows the result.

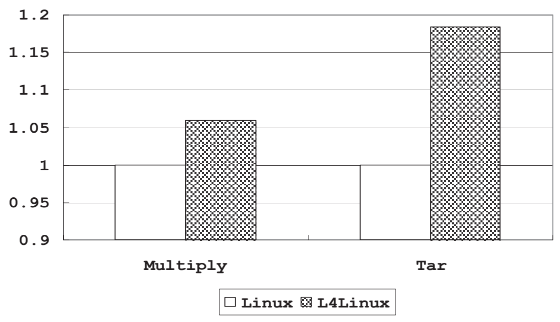

**Figure 16. Relative user application's performance on L4linux and Linux**

Figure 16 shows that the performance in L4Linux is about 5–17 % slower than normal Linux. Therefore, we profiled each application using Linux OProfile and our profiling framework, and compared the hardware events such as instruction counts, L2 cache misses, data TLB misses, and instruction TLB misses. Figures 17 and 18 show the normalized values for these events relative to the Linux.

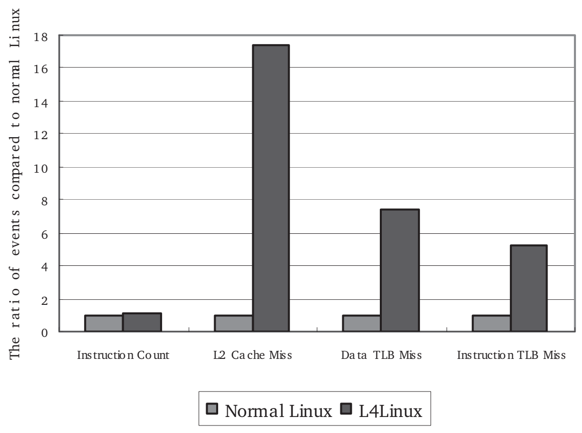

**Figure 17. Relative hardware event counts in L4Linux and Linux for Multiply workload**

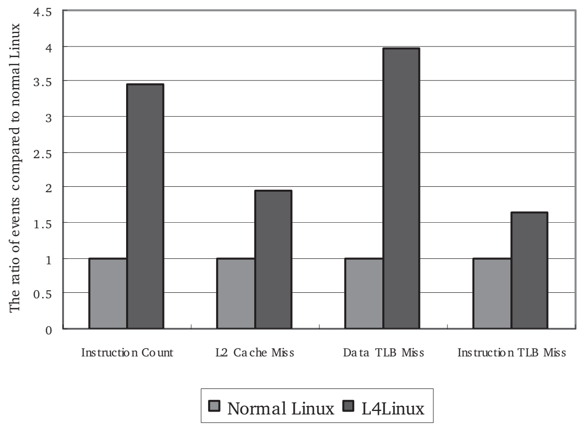

**Figure 18. Relative hardware event counts in L4Linux and Linux for Tar workload**

Figures 17 and 18 show that L4linux has higher cache and TLB miss rates compared to Linux. Since the L4 kernel considers L4Linux's applications as user process, each L4Linux system call leads to at least two context switches, that is, 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. Therefore, we showed that the main cause of context switches resulted from system calls.

## 7. Conclusions

In this paper, we presented our profiling framework, L4oprof, a system-wide statistical profiling tool for the L4 microkernel environment. Our profiling framework leverages the PMU supported by conventional CPUs to enable profiling of a wide variety of hardware events such as clock cycles and cache and TLB misses. As far as we know, it is the first performance monitoring tool which uses the PMU in the L4 microkernel environment. For easy and fast development, we adapted Linux OProfile and extended its capabilities to be used in the L4 environment. We believe that L4oprof can help to develop L4 applications and reduce the optimization time due to its good performance.

Currently, L4oprof shows less than 3 % overhead below 15000 interrupts per second compared to the Linux OProfile, depending on the sampling frequency. As we presented in the case study section, the major overhead of the L4oprof results from running Linux in user mode on top of the microkernel. Moreover, using our profiling framework, we investigated why L4Linux applications show lower performance than normal Linux, and found that this behavior results from frequent context switches from system calls.

Currently, L4oprof supports only Intel Pentium 4 CPUs. We will port it to additional CPUs such as ARM/Xscale and AMD64. Investigating the performance of multiple L4Linux servers and supporting dynamic sampling frequency also remains as our future work. Furthermore, our profiling framework needs performance optimization such as moving the operations in the current OProfile server layer into the L4 microkernel layer to eliminate overhead of running in the user mode.

## 8. ACKNOWLEDGMENTS

The authors would like to thank the Ministry of Education of Korea for its support toward the Electrical and Computer Engineering Division at POSTECH through its BK21 program. This research was also 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).

## 9. 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, Technische Universität 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 VTune™ 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. Großer Beleg. Technische Universitat Dresden, Lehrstuhl für Betriebssysteme, 2003.

[12] T. Riegel. A generalized approach to runtime monitoring for real-time systems. Diploma thesis, Technische Universitat Dresden, Lehrstuhl für Betriebssysteme, 2005.

[13] M. Pohlack, B. Döbel, and A. Lackorzynski. 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