Performance Monitoring and Visualization of Object-Oriented Operating Systems

by

Adam Czajkowski

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Performance Monitoring and Visualization of Object-Oriented Operating Systems

Adam Czajkowski
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2008

Computer systems are becoming more complex every year, making it exceedingly difficult to understand how well a running system is performing. Various tools have been developed to aid in systems analysis. Some of them can be used to monitor performance and help identify performance bottlenecks. However, they all have limitations, and the scope of their monitoring capabilities is usually limited.

This dissertation presents a new monitoring system called the Kernel Object Viewer (KOV) that makes three primary contributions. First, it can dynamically track important performance metrics using Hardware Performance Counters:

1. at object instance-level granularity,
2. requiring no changes to the code,
3. adding no overhead when monitoring is not required, and
4. allowing monitoring overhead to be varied by dynamically changing the sampling frequency.

Second, it implements a system-wide scanning facility which extracts all live processes and their objects. Finally, it implements a novel mechanism to dynamically track lock acquisitions and lock contention.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor, Professor Michael Stumm. I am deeply grateful for the guidance, support, and encouragement he has provided me with over the years. Throughout the course of my research his advice has always been invaluable, and his extensive knowledge and expertise have made this research possible.

I would like to thank my family, Dr. Gregory Czajkowski, Tatiana Czajkowska and Tomasz Czajkowski, for their constant support and inspiration.

I would also like to acknowledge The Edward S. Rogers Sr. Department of Electrical & Computer Engineering for its financial support.

Contents

Chapter 1

Introduction

Computer systems, consisting of both hardware and software, are becoming more complex every year. For example, processor architectures have added deeper pipelines, multi-level caches, branch predictors, multiple cores, multi-threading, and other features in recent years. Often, multiple processing units are combined to form a single multiprocessor. Operating systems, which provide a layer of isolation between applications and the raw hardware, have added support for multi-processor architectures, enhanced security, numerous optimizations within the file system and memory management subsystems, as well as support for many new devices.

This increased complexity makes it exceedingly difficult to qualitatively and quantitatively understand how well a running system is performing. Moreover, for performance-critical applications it is equally challenging to deduce the effects of code (or design) modifications on performance (i.e., whether the changes significantly affect branch mispredictions or cache miss rates, or whether the changes significantly affect the page miss rate or load balancing). Similarly, a programmer might want to identify performance bottlenecks and their precise causes with the goal of optimizing the code, yet this is becoming more difficult as the complexity of the system substrate increases. Likewise, operating system designers will want to identify performance bottlenecks under different workloads.

Various tools have been developed to aid in systems analysis. Some of them can be used to monitor performance and help identify performance bottlenecks. However, they all have limitations, and the scope of their monitoring capabilities is usually limited to a particular layer. For example, the VTune monitoring tool monitors solely at the hardware level by presenting information gathered from hardware performance counters. Another performance monitoring tool, called GNU gprof, requires recompilation in order to monitor a target application, and thus is not fit for production environments. Similarly, neither of these tools simultaneously monitors a target application's performance as well as kernel-level structures (e.g. memory manager) which serve this application. In cases where an application performs a significant amount of I/O or network operations, a combined user-level and kernel-level look at performance is more comprehensive and thus more useful at accurate and precise identification of performance bottlenecks.

In this dissertation we present the design and implementation of the Kernel Object Viewer (KOV), an object-oriented monitoring system which

1. combines hardware-level and software-level monitoring capabilities,
2. simultaneously monitors user-level and kernel-level software components,
3. can attribute performance bottlenecks to specific object-level code segments, and
4. provides sophisticated visualization support to aid in system performance debugging.

We used KOV to analyze the performance of different applications on the K42 operating system. We demonstrate how KOV identifies hot objects in the operating system, and how KOV provides useful visual information by showing the rate of hardware events on a per-object basis.

1.1 Existing Monitoring Software

Existing performance monitoring tools can be classified into two categories, called static and dynamic instrumentation, based on the type of instrumentation used to quantify performance. Static instrumentation tools augment the original binary with extra instrumentation off-line. In contrast, dynamic instrumentation tools are capable of plugging in and removing monitoring functionality seamlessly at runtime. There exists a plethora of commercial performance monitoring tools available for multiple operating system platforms, as well as open-source alternatives available for several operating systems such as Linux and Solaris. One can acquire proprietary tools, such as Intel VTune, Windows System Performance Monitor, and LivePerf, or download open-source alternatives such as GNU gprof and InfraRED.

We provide a brief overview of some of these tools and highlight their current limitations.

Intel VTune

VTune is a sampling profiler developed by Intel and designed for generic performance monitoring of applications running on Intel hardware [36]. VTune uses hardware performance counters to monitor processor events (and thus the limitation to Intel hardware), or simply to provide a CPU utilization breakdown on a per-function basis. VTune uses the program counter (PC) value obtained automatically from each hardware performance counter event to trace back the aforementioned hardware events to specific segments of code.

Windows Performance Monitor

The Windows Performance Monitor tool is freely available for Windows 2000, Windows Server 2003, and Windows XP [28, 33, 44]. It enables the user to gather operating system level data such as number of memory pages swapped to disk, time spent executing each process, or disk usage. Each performance metric is attributed by this tool to a physical resource at coarse granularity. For example, execution time is attributed to "processor", and disk usage to "physical disk". Data are presented at process granularity versus time for each of the characteristics being monitored. All performance monitoring data gathered with this tool are also displayed in a graphical user interface (GUI) provided with the tool. This tool exclusively relies on performance gathering capabilities incorporated into the Windows operating system.

LivePerf

The LivePerf performance monitoring tool is a performance data gathering and displaying facility [27]. A GUI collects and displays data gathered from applications using their built-in performance gathering capabilities. LivePerf does not contribute any of its own monitoring capability, but rather relies on the targeted application's API to interface with and gather data from it. It currently supports gathering data from Websphere, SQL Server, DB2, Oracle, Unix and Windows operating system, Apache and IIS webservers, .NET and J2EE applications. In addition, LivePerf enables logging and replaying of different statistics.

GNU gprof

GNU gprof is a tool that allows a user to learn where a program spends its time executing, which functions were called and how many times [21, 31]. With this information it is possible to deduce which portions of a program are slower than expected. However, gprof does not provide the reason for the slowdown. The type of information generated by gprof can single out "hot" functions, and thus allows the user to focus their efforts. Although somewhat useful for simple programs, gprof is not able to instrument multi-threaded applications. As such, the utility of this tool is limited to simple programs, and only provides course-grained timing measurements at function-level granularity.

InfraRED

InfraRED is a tool for monitoring performance of J2EE applications and diagnosing performance problems [24]. It collects metrics on various aspects of an application's performance and makes it available for quantitative analysis of the application. InfraRED uses Aspect-Oriented Programming (AOP) to inject performance monitoring code between distinct parts of the application, as defined by the AOP paradigm [25]. InfraRED is a comprehensive monitoring tool for programs written in Java, however it only utilizes software data gathering techniques (as provided by the instrumentation inserted between the AOP blocks).

Generic Tools

It is not always necessary to buy or download a performance monitoring tool. Operating systems typically provide useful utilities with the original distribution. For example, the Linux or Solaris operating systems contain vmstat, iostat, netstat, and other simple tools. Although these tools can provide statistics on virtual memory, disk access, processor activity (in the case of vmstat), or I/O (iostat), the results are aggregated for the system as a whole. These tools do not generally identify the source of a particular bottleneck. Nevertheless, such tools can be helpful at identifying the existence of a performance problem, but they are not specific enough for performance debugging.

Limitations of existing monitoring tools

The performance monitoring tools presented above gather performance data either at the hardware level or at the software level, but not both. Only a few of the tools map potential performance issues back to the code segments that incur them. Intel's VTune is one of the few, as it is capable of tracing hardware events back to the lines of code that caused them. However, none of the tools directly support object-oriented code by mapping potential performance bottlenecks back to specific object instances. Yet this information could be useful. By identifying which object instance causes a performance bottleneck, it becomes easier to perform accurate and precise performance debugging.

Not all of the above tools have a Graphical User Interface (GUI). The advantage of having a GUI in a performance monitoring tool is clear. Clarity in displaying the data makes it easier to interpret, and well designed graphs can highlight important trends.

1.2 KOV Overview

This dissertation describes the design and implementation of a performance monitoring tool called the Kernel Object Viewer (KOV) that addresses the limitations described above. KOV is comprised of two main parts connected by a common interface.

1. An ensemble of static and dynamic instrumentation together with a performance data gathering module located in the operating system kernel, and
2. A graphical user interface which displays the performance data at process and object-level granularity.

KOV was implemented for K42, an object-oriented operating system for shared memory multiprocessors. KOV uses both hardware-level and software-level monitoring capabilities. Hardware performance counters (HPCs) are used to obtain information on hardware-level events (e.g. TLB misses, IPC rates), and software modifications enable system analysis (e.g. by obtaining a list of live objects), as well as provide information on software-level metrics (e.g. sleep lock queue lengths, object invocation counts).

HPCs can be configured to generate events periodically at a user-defined frequency. As such, the KOV monitoring system allows for fine-grained, variable-overhead performance readings of any hardware-level event being monitored. Using HPCs allows this performance monitoring technique to be fully dynamic with respect to a target application or OS subsystem, because monitored application or system-level code is not modified.

The benefit of this form of dynamic instrumentation is that it incurs no overhead when turned off, and can be enabled or disabled at any time without restarting the system or target application.

Software-level performance gathering capabilities have been implemented with direct source-code modifications. Object classes were extended to allow for a scan of live objects, as well as to provide a hierarchical relationship with other objects. When an object is scanned, it responds with a list of other objects it refers to, and thereby positions itself with respect to other objects within the system's hierarchy. Although this static instrumentation cannot be removed without recompiling the system, it is outside of the critical path, and is in fact not invoked outside the scope of the KOV performance monitoring system. Other software-level instrumentation code enables logging of object invocations, as well as measurement of sleep lock queue lengths. This code introduces overhead to the system, even when disabled, and is quantified in Section 4.3.

KOV collects performance data from both kernel-level and user-level components. HPCs can be configured to count hardware events generated by the execution of either kernel-level code, user-level code, or both. Likewise, the object scan initiated by KOV is on a per process basis, and thus the scan can include the kernel process and arbitrarily many user processes. Software-level monitoring of sleep lock queue lengths is also available for both user-level and kernel-level code. Since the sleep lock instrumentation is part of the class definition used to implement sleep locks (this class is provided as a part of libc), any code which instantiates this object class will enable itself to be monitored by KOV.

For object-oriented software, it is particularly important to be able to track performance at object instance granularity. Because an object instance encapsulates data specific to the instance, two objects of the same class may experience observably different performance behaviour even though they share the same code base. As a result, optimizations on an entire class may improve performance for some instances but worsen performance for other instances. Some object-oriented systems support dynamic optimization at object instance granularity, such as K42's hot-swapping facility [8], where individual object instances may be replaced dynamically at run-time by an instance of another (related) class providing the same functionality but with a different implementation.

The live representation of the system, as well as all performance data gathered, are shown in a graphical user interface. The aforementioned object hierarchy is shown in the form of a tree, and performance bottlenecks are shown either directly on the tree (by highlighting the problem objects), or in one of many types of graphs. By displaying performance data in a KOV graph, the user can depict behaviour over time for a specific object instance, or the system as a whole. It is also possible to display all the objects with their attributed percentage of total data.

We have implemented KOV on the K42 operating system running on PowerPC hardware. KOV obtains data from the architecture's Performance Monitoring Unit (PMU) using the statistical PMU sampling system we developed as an extension of Azimi et al [11]. Currently, we use a GUI to display the performance statistics gathered and updated in real-time. The GUI displays a hierarchical tree of object instances and performance statistics in two ways: (*i*) object instances are displayed ordered by severity of performance impact, and (*ii*) performance statistics for any object instance are displayed over time. Using the GUI, it is possible to observe correlations between different performance metrics (such as correlation between lock contention and TLB misses). Ultimately, we envision a scenario where dynamic run-time optimizers will obtain the statistics from the monitoring infrastructure and use object hot-swapping [8] to improve system performance.

1.3 Contributions

In this dissertation we make three primary contributions. First, we describe a performance monitoring infrastructure we have developed, called the Kernel Object Viewer (KOV), that dynamically tracks important performance metrics using Hardware Performance Counters:

- at object instance-level granularity,
- requiring no changes to the code,
- adding no overhead when monitoring is not required, and
- allowing monitoring overhead to be varied by dynamically changing the sampling frequency.

The second contribution of this dissertation is a comprehensive, system-wide scanning facility which extracts all live processes and their objects. In addition, the scanning capabilities of KOV encompass both the kernel process and user processes, and cause no overhead when not in use.

The third contribution of this dissertation is a novel mechanism to track lock acquisitions and lock contention in a way that requires no changes to the code. To gather statistics on locks, we use PMU features of IBM PowerPC processors to count occurrences of load-linked and store-conditional instructions and then walk the stack to tie lock events to specific object instances.

1.4 Outline

The rest of this dissertation presents background information relevant to KOV in Chapter 2, the design and implementation of our tool in Chapter 3, followed by an experimental evaluation of KOV in Chapter 4, and concluding remarks in Chapter 5.

Chapter 2

Background

This chapter presents background material to allow a better understanding of the remainder of this dissertation. Specifically, we present a brief overview of the Performance Monitoring Unit, provide a more in-depth explanation of static and dynamic software instrumentation, and conclude by describing the structure of the K42 operating system, the platform upon which the Kernel Object Viewer (KOV) prototype was evaluated.

2.1 Performance Monitoring Unit

Today's complex, speculative, out-of-order execution cores usually come with sophisticated Performance Monitoring Units (PMUs). A modern processor will have a PMU that supports memory hierarchy profiling, instruction sampling, functional unit event sampling, and potentially other implementation dependent functionalities. An example architecture of a performance monitor is shown in Figure 2.1. A Hardware Performance Counter (HPC), accumulates occurrences of a distinct hardware event, chosen by a series of multiplexers which lead to the counters block in which the actual HPCs reside.

A typical PMU will have many HPCs (e.g. PowerPC 970FX has eight), where a single HPC can be configured to count only one type of event from a subset of all hardware events at any given time. The size of this subset of events available for monitoring by each HPC is determined by multiplexing logic leading up to the counters block, as shown in Figure 2.1. Although actual implementation details vary depending on the specific processor architecture, it is sufficient to point out that events are chosen by a network of multiplexers comprised of several stages.

In the platform used to develop the prototype described in this dissertation, specifically the PowerPC 970FX, three stages of multiplexers choose which events end up being counted by each HPC. Given that the total number of events that could potentially be counted on this architecture is roughly 500, from a hardware implementation point of view, it is too costly to have eight, 500-way multiplexers to allow each and every HPC to count any event. Consequently, the cheaper three-stage network of multiplexers allows each HPC to count only a specific subset of all hardware events. By assigning each HPC to a different set of hardware events, it is then possible to span all events by using all available counters. On the PowerPC 970FX there is some overlap between the sets of events each counter can monitor, consequently giving the programmer some flexibility in terms of configuration parameters.

Functionally, a Hardware Performance Counter is a register which accumulates the occurrences of hardware events. A software program, running in kernel-space, can directly read from, and write to, all HPC registers.

A simple usage model for performance monitoring involving the use of HPCs could be realized by periodically polling (reading) all counters to obtain an update of all event counts being monitored. Since polling is considered an inefficient programming technique, due to the constant overhead associated with periodic reads, the PMU also allows counters to generate interrupts.

The use of PMU interrupts for performance monitoring is a two step process. The first step involves configuring a set of Special Purpose Registers (SPRs) on the PMU (registers referred to as MMCRx). A software program, running in kernel-space, will write to these SPRs values that will determine when a PMU interrupt is to be generated,

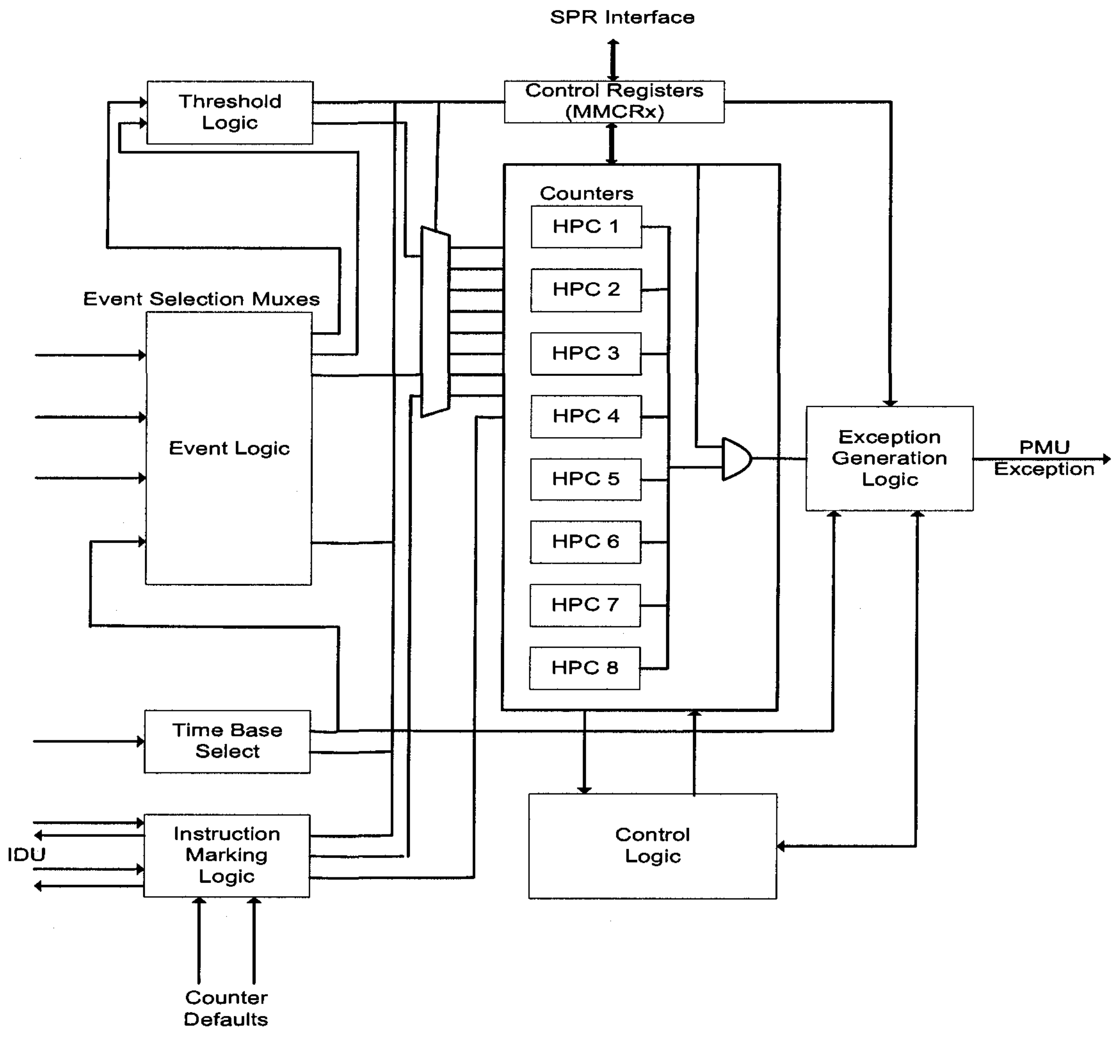


Figure 2.1: Performance Monitor Architecture for the PowerPC 970FX RISC Microprocessor. Hardware Performance Counters (HPCs) receive signals from a variety of sources, ranging from a thresholder, event generator, time base selector, instruction marker, processor functional units (i.e. arithmetic-logic unit), and others. The crucial link between the hardware and software layer is the Exception Generation Logic. Based on a set of flags set in Special Purpose Registers (SPR), an interrupt will be generated once any of the counters meets a pre-defined condition set in performance monitor specific control registers (MMCRx).

and which HPCs are to generate interrupts. PMU interrupts can be generated when a specific HPC overflows, or periodically after executing a specified number of processor cycles.

The second step of using PMU interrupts for performance monitoring involves the Interrupt Service Routine (ISR), which is called whenever a PMU interrupt occurs. The ISR is a software program which runs at exception-level and is capable of performing low-level tasks, such as reading from, and writing to, SPRs and HPCs, or saving data to software buffers for later analysis. The programmer must be careful when writing an ISR, because an exception (such as a page-fault) when executing an ISR will cause the system to crash. As such, an ISR should only perform simple tasks.

A simple usage model for performance monitoring involving the use of interrupts could consist of setting one HPC to generate an interrupt every 10,000 branch misprediction events, and the aforementioned ISR to read the HPC value and save it in a dedicated software buffer. Another program could subsequently read this data from the buffer and use it for performance analysis.

In conclusion, HPCs allow counting of detailed micro-architectural events in the processor [4, 17, 18, 40], enabling new ways to monitor and analyze performance. There has been a considerable amount of work done using HPCs to explore the behaviour of applications, as well as to identify performance bottlenecks resulting from excessively stressed micro-architectural components [3, 19, 42].

2.1.1 PMU Limitations

Most microprocessor PMUs offer a limited number of HPCs. For instance, the IBM POWER4 and PowerPC970 provide eight HPCs, the POWER5 has six per SMT (two of which are hard-wired), Intel Itanium II has four generic HPCs and five registers for holding instruction and data address samples, and the AMD Athlon has four generic HPCs. In addition to the limited number of HPCs, there are often restrictions on the combinations of hardware events that the HPCs can count, as previously mentioned for the PowerPC970. Other architectures have similar limitations. For instance, although Intel P4 and Xeon have 18 HPCs, they are divided into nine groups, each of which can only count events from a specific subset of 48 available hardware events.

In many performance monitoring scenarios, several low-level hardware events must be counted simultaneously to obtain information of interest. For instance, to obtain the L1 data cache miss rate on an IBM POWER4 processor, at least four separate events must be counted (L1 load misses, L1 store misses, L1 loads, and L1 stores). Also, usually two HPCs have to be dedicated to counting processor cycles and instructions retired to provide context for other data being gathered. For example, one thousand branch mispredictions per one million retired instructions is less significant than one thousand branch mispredictions per fifty thousand retired instructions (a 20x difference). On a processor such as the AMD Athlon, that leaves only two unused HPCs to gather other hardware events. These two remaining HPCs are not sufficient to count L1 load misses, L1 store misses, L1 loads, and L1 stores.

Even if one had eight counters available, such as on the PowerPC 970FX, and could fit all the L1 events as well as processor cycles and retired instructions in a single PMU configuration, there is other information available which, if monitored, could aid in accurate and precise performance debugging. It might be desirable to also monitor L2 cache misses, L3 cache misses, branch mispredictions, TLB misses, instruction mix (integer/floating point), instruction cache misses, or other events that are important for obtaining a complete picture of application and system performance depending on the workload being explored.

2.1.2 Software Multiplexing of Hardware Performance Counters

To address the need for a large number of hardware counters to enable comprehensive hardware event monitoring, Azimi et al proposed and implemented a system which dynamically multiplexes the set of hardware counters by using fine-grained time slices [11]. The programming interface component takes a set of events to be counted as input and automatically assigns them to a number of HPC groups such that in each group there are no conflicts due to PMU constraints. The sampling engine assigns each group a fraction of cycles out of the total sampling period. At the end of each HPC group's time slice, the sampling engine automatically assigns another HPC group to the PMU. The value that is read from an HPC is scaled up linearly as if that group was active during the entire period. As a result, the user's program (e.g. a run-time optimizer) is presented with more logical counters than actually exist in the underlying processor architecture.

Some types of events exhibit more volatile behaviour than others. For example, L1 miss rates change dramatically depending on the current workload, whereas the number of completed processor cycles does not. To capture greater variations in system performance it becomes necessary to increase the sampling frequency for those events which change more often. As defined by the Nyquist rate, the HPC sampling rate must be at least twice the frequency of changes within the data being monitored (e.g. if L1 miss rate changes every 100ms, HPCs must sample at least every 50ms to accurately measure event transitions).

The multiplexing system introduced by Azimi et al can easily be programmed to favour certain HPC groups by using their configuration for longer periods of time. This is accomplished by allocating multiple time slices to the group, rather than just one in the default case. This PMU multiplexing scheme is analogous to time-sharing a CPU amongst various processes. Moreover, the accuracy may differ for different hardware events with the same share size. A default share assignment scheme might be overridden by explicit requests from a user interested in closely monitoring a specific hardware event, such as L1 miss rates.

With multiplexing, time is usually measured in terms of CPU cycles. Therefore, one counter in each HPC group is reserved to count CPU cycles. The use of cycle counters as timers allows the user to define arbitrarily fine time-slices down to a few thousands of cycles. Another metric that can be used to define HPC group share sizes is the number of instructions retired. The main advantage of instruction-based multiplexing is that the HPC group share sizes are aligned more closely with the progress of the application. Actual share sizes will differ in terms of real time depending on the amount of available instruction level parallelism (ILP) in the application currently running.

Sampling introduces inaccuracies in measurement. A pathological case for the multiplexing engine is the existence of a large number of short-lived bursts of a particular hardware event. If the burst time is shorter than a particular number of cycles, the HPC that counts that hardware event might be inaccurate because the PMU actually counts the event only in a fraction of the total time slice, and thus it may miss short-lived bursts. However, given that most applications will go through several execution phases, each longer than a time slice, the occurrence rate of hardware events is stable in the common case. Experimental results presented by Azimi et al demonstrate that the statistical distance between the sampled and real rates of hardware events is small in most cases.

KOV makes extensive use of Azimi's HPC multiplexing infrastructure. KOV provides a Graphical User Interface (GUI) for configuring the PMU on the PowerPC 970FX. Each set of configuration parameters can be saved in a config file, such as "TLB misses" or "Branch mispredictions". The GUI logic prevents the user from choosing conflicting events within a single configuration. The user can subsequently send both the "TLB misses" and "Branch mispredictions" configurations to the multiplexing infrastructure and specify the percentage of time each configuration is to be used within the PMU. The multiplexing infrastructure automatically switches between the two different configuration parameters.

2.2 Instrumentation Techniques

To address the growing need for comprehensive performance monitoring systems, a variety of tools have emerged that can loosely be classified into one of two categories depending on the type of instrumentation being used to quantify performance. Static instrumentation systems can be characterized by the fact that they augment the original binary with extra instrumentation off-line. In contrast, dynamic instrumentation is defined by the capacity to plug in and remove monitoring functionality seamlessly while the system is running. Examples of static instrumentation systems include: KLogger[20], K42's event log[45], Linux Trace Toolkit[46], and Lockmeter[15], and examples of dynamic instrumentation systems include: KernInst[43], Kprobes[32], DTrace[16], SystemTap[34], JIFL[30], and PAPI[14].

2.2.1 Static Instrumentation

Static instrumentation can be the done via direct source code modifications such as with Paragon[37] and K42's event log[45] or by instrumenting the binary off-line as is done with Atom[41]. Although not requiring source code in the latter case does increase the system's utility, this type of approach still suffers from the need to restart the target application when new instrumentation is added or removed. Similarly if the OS is being instrumented, a full system reboot is necessary. An example of how static instrumentation could be done is shown in Figure 2.2. When adding static instrumentation, the user has unlimited freedom in where and how to instrument. However, static instrumentation incurs constant overhead and requires recompilation to remove it completely.

```
long int invocation_count;

void foo()
{
    /* Source code instrumentation counts number of function calls */
    Invocation_count++;

    /* Start of Original Function */
    ...
}
```

Figure 2.2: An example implementation of static instrumentation. Since the system is not running when adding this type of instrumentation, the user has unlimited freedom in where and how to instrument. Unfortunately, this approach incurs constant overhead and requires recompilation to remove it completely.

As such, static instrumentation is inappropriate for production environments in which restarts represent an unacceptable lapse in service. Moreover, static instrumentation does not lend itself well to the analysis of systemic problems or emergent misbehaviour [29], which only appears after prolonged and continuous system operation, because it implies having all the necessary functionality already in place. This is difficult to achieve in practice because the nature of an emergent problem is not know during development.

2.2.2 Dynamic Instrumentation

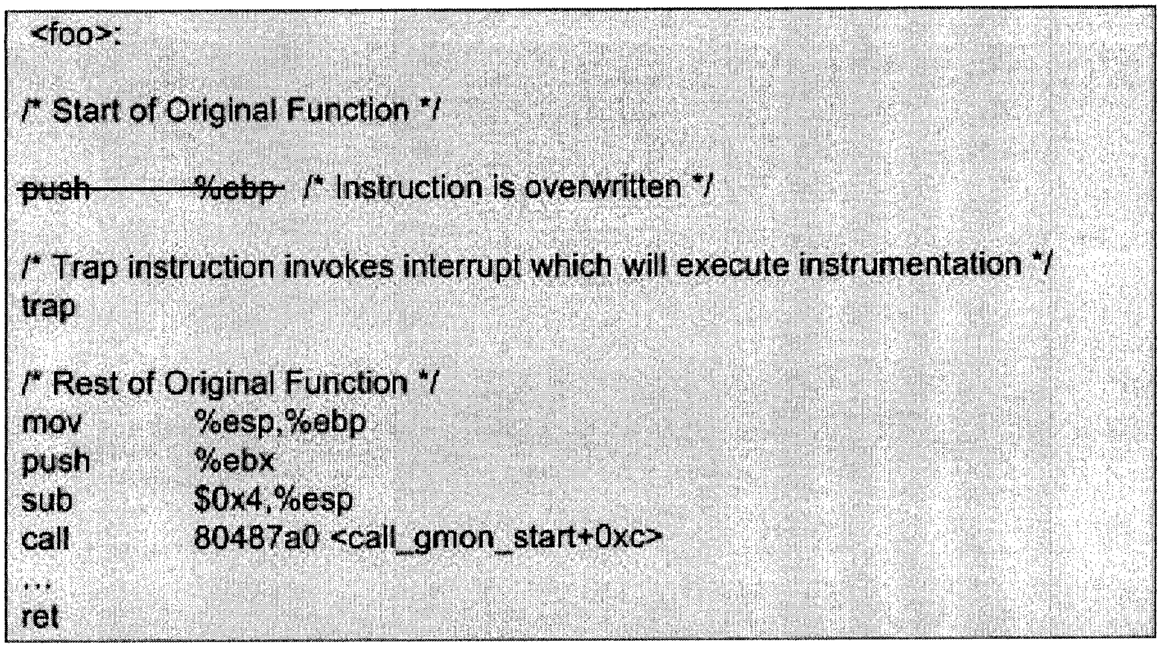
Dynamic instrumentation on the other hand entails no off-line modifications to the target application or OS subsystem being monitored. Instead, the monitoring tool inserts dynamic instrumentation by modifying a target application's executable code in memory. Alternatively, hardware generated events can be used to interrupt a currently executing program to take snapshots of its current state. These are only a few techniques of dynamic instrumentation. Because dynamic instrumentation is not as direct as static instrumentation, it is also much richer in terms of the assortment of existing approaches, and currently remains an active field of research.

Within the scope of dynamic instrumentation, one can consider software and hardware approaches. The former consists of two main groups, probe and just-in-time (JIT) based paradigms, whereas the latter utilizes hardware performance counters (HPCs). Due to the inherent level of abstraction at the software and hardware level, some information lends itself better to either one of the two approaches. For example, when interested in the behaviour of a scheduling algorithm, hardware has no notion of processes and thus is not a good choice. Alternatively, when monitoring cache misses, a program does not experience them directly, and thus HPCs are a better solution.

The first dynamic instrumentation technique to have emerged is the probe-based approach. It works by overwriting instructions in the original program with trampolines to instrumentation code. Effectively, only one instruction is overwritten whereas the actual instrumentation function, which resides in a different location, can be arbitrarily large. Since the overwritten instruction isn't executed once it is overwritten, its effects have to be duplicated by the instrumentation function, to ensure correct execution of the original program.

Probes can be implemented in a straight-forward way on fixed-length instruction set architectures (ISAs), such as Sun's UltraSparc, by inserting jump instructions to the relevant code at each instrumentation point; however, on variable-length ISAs, such as the popular Intel x86 and AMD x86-64, probes have to be implemented with trap instructions [43]. An example of what such an instrumented program would look like is shown in Figure 2.3. At each instrumentation point, execution of the trap instruction causes an exception handler to be dispatched. The handler must then determine what type of instrumentation is needed at that point. The overhead of such traps can be substantial, and it can make comprehensive and fine-grained instrumentation unfeasible [30].

Just-In-Time (JIT) based dynamic instrumentation was developed to address some

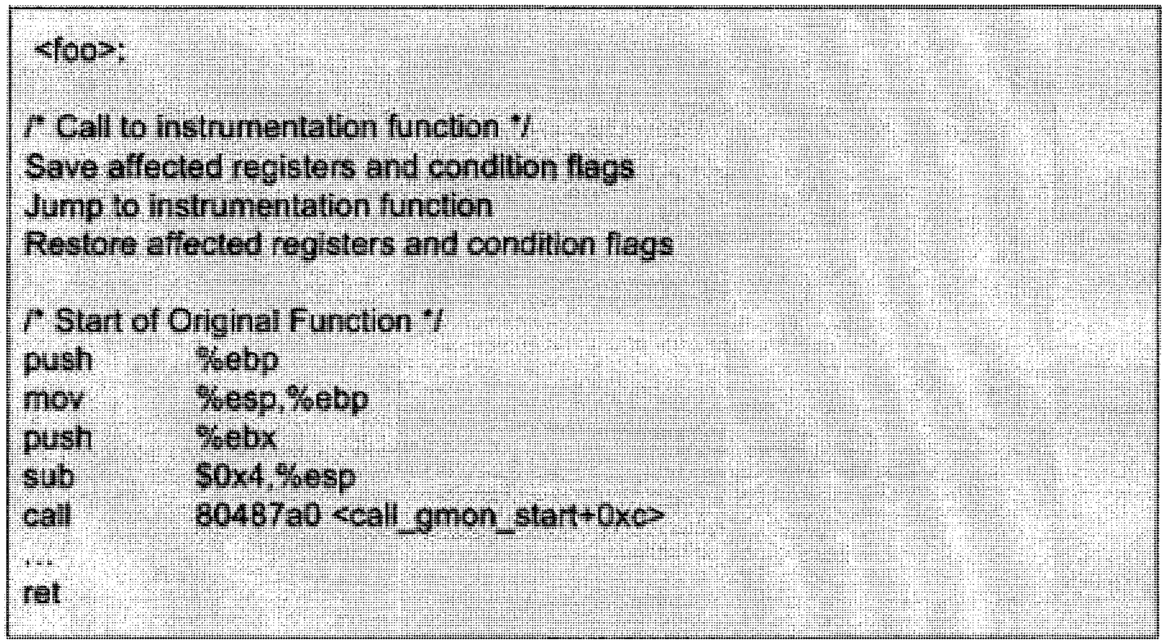


```
<foo>:

/* Start of Original Function */

~~push        %ebp~~ /* Instruction is overwritten */

/* Trap instruction invokes interrupt which will execute instrumentation */
trap

/* Rest of Original Function */
mov         %esp,%ebp
push        %ebx
sub         $0x4,%esp
call        80487a0 <call_gmon_start+0xc>
...
ret
```

Figure 2.3: An example implementation of dynamic instrumentation using the probe-based approach. In order to keep the binary the same size, and thus preserve the correctness of static branch targets, an existing instruction is overwritten with a trap. The resulting interrupt will call the required instrumentation function based on the trap's address and compensate for the overwritten instruction with extra code.

of the most prevalent shortcomings of the probe-based techniques. With this approach, execution is redirected to a runtime system at the entry of a section of code that is to be instrumented. A JIT compiler creates a duplicate copy of each basic block of the original code immediately before it is executed, embedding calls to instrumentation routines within it, much as if the instrumentation had been added to the source code and the source recompiled. An example of what this type of instrumentation would look like is shown in Figure 2.4. The resulting instrumented basic blocks are stored in a code cache, from where they are dispatched instead of the original code.

JIT instrumentation can provide better performance and a better usage model than probe-based techniques for large amounts of fine-grained instrumentation. The primary



```
<foo>:

/* Call to instrumentation function */
Save affected registers and condition flags
Jump to instrumentation function
Restore affected registers and condition flags

/* Start of Original Function */
push        %ebp
mov         %esp,%ebp
push        %ebx
sub         $0x4,%esp
call        80487a0 <call_gmon_start+0xc>
...
ret
```

Figure 2.4: An example implementation of dynamic instrumentation using the JIT-based approach. Since the code being instrumented is in the code cache, and not in the original binary, no existing instructions need to be overwritten. A call to the desired instrumentation function is inserted, as well as necessary instructions required to maintain the original program's state which could be corrupted by instrumentation.

performance advantage stems from the fact that instrumentation can be inserted between instructions. This eliminates the need for expensive trap instructions to redirect execution to instrumentation code on variable-length ISAs. In addition, when using the JIT technique, instrumentation is only inserted into code after it is known that it will execute, thereby eliminating any cost of instrumenting instructions that might not be executed. Furthermore, if the instrumentation code is small enough, it can be inlined directly into the copied basic blocks to eliminate the cost of executing function calls.

From a usability point of view, probe-based instrumentation requires a user to specify the exact locations in the code where instrumentation should be inserted. When instrumentation of a large amount of code is desired, the user is required to manually specify the location of a large amount of probes. Manually inserting hundreds of probes can become erroneous. In contrast, JIT instrumentation requires only entry, and possibly exit, points of the entire code being considered for instrumentation to be identified. Because instrumentation is added as code blocks are discovered, there is no need for a priori identification of possible instrumentation points. For example, when instrumenting a system call with the JIT technique, only the appropriate entry in the system call table needs to be identified. Conversely, the probe-based technique would require the user to go through the entire system call code and specify the addresses for all the probes to be inserted.

2.3 Instrumentation Tools

We have described the various techniques available to monitor performance. We now describe several tools that implement these techniques in greater detail. Examples of static instrumentation systems include: KLogger[20], K42's event log[45], Linux Trace Toolkit[46], and Lockmeter[15], and examples of dynamic instrumentation systems include: KernInst[43], Kprobes[32], DTrace[16], SystemTap[34], JIFL[30], and PAPI[14].

2.3.1 KLogger

KLogger monitors performance by recording occurrences of kernel events and using hardware performance counters. The logging code is integrated into the kernel and activated at runtime by a special *sysctl* call using the *proc* file system. KLogger logs events defined at compile-time, including events such as context switching, recalculation of priorities, forks, execs, changing the state of processes, and others. Logs are stored in a memory buffer (typically 4MB). Every five seconds a daemon stores the contents of the memory buffer to disk, allowing the user to analyze the data at a later time. Because the events monitored by KLogger can only be specified at compile time, and results are not analyzed in real-time, this tool is not usable for performance debugging production systems, where a full system reboot presents an unacceptable lapse in service. In contrast, KOV can change the metrics it monitors while the system is running, and analyzes results in real-time. This flexibility is important to detect bottlenecks immediately as they emerge, and to ease the analysis of a newly identified bottleneck by choosing a more specific set of metrics (e.g. when the memory subsystem becomes a bottleneck, KOV can start monitoring L1 cache, L2 cache, and TLB miss rates immediately upon a user's request).

2.3.2 K42's event log

K42's event tracing infrastructure provides for correctness debugging, performance debugging, and performance monitoring of the system [7]. This infrastructure allows for inexpensive and concurrent logging of events by applications, libraries, servers, and the kernel. All events are stored in a special event log which can be read in several ways. This event log may be examined while the system is running, written out to disk, or streamed over the network. Post-processing tools allow the event log to be converted to a human readable form or to be displayed graphically. The types of events that can be monitored by this tracing facility are divided into 64 major classes. Events that are related to a common purpose are placed in the same major class (e.g. memory management events are grouped into *traceMem*). All classes of events represent actions or positions in the code deemed important by the developer.

Although useful, this infrastructure only monitors components of K42 explicitly identified at compile-time. In addition, when enabled, the tracing facility will log all events from a selected major class of events, regardless of how many events are of interest to the user. This can be problematic if the user is only interested in a small subset of events from a particular class, because there can be a significant amount of data obscuring the results the user is actually interested in seeing. In contrast, KOV only monitors occurrences of events the user has explicitly chosen, avoiding overhead resulting from gathering superfluous data.

2.3.3 Linux Trace Toolkit

The Linux Trace Toolkit (LTT) [46] is a static instrumentation system that allows a user to record and analyze system behaviour. The toolkit is capable of recording information such as CPU time per process, instruction count, fraction of time spent in each function, as well as disk and network I/O statistics. The main contribution of this work is its data collection facility that records and stores pertinent information.

The data collection facility consists of three main components: a trace facility, a trace module, and a daemon. The trace facility functions as a unique entry point to all other kernel facilities. Once an event occurs, the trace facility forwards the event to the trace module. The trace module can then determine the type of event that has occurred. If the event is one identified to be monitored, then the trace module will keep track of it. However, it is possible to configure the trace module to ignore some of the events it was designed to track by the means of an *ioctl* system call. This permits a finer control of the monitoring system.

One drawback of the Linux Trace Toolkit is the fact that it does not attribute performance bottlenecks to specific code segments. Although kernel events caused by the execution of a program can be logged, the LTT does not identify which parts of the program caused these events to occur. In contrast, KOV attributes performance bottlenecks to specific code segments as well as individual object instances.

2.3.4 Lockmeter

Bryant and Hawkes designed and implemented a spin lock monitoring infrastructure for the Linux kernel called Lockmeter [15]. Similarly to our approach, Lockmeter can identify problematic spin lock instances with low overhead; however, Lockmeter requires source code annotation of Linux lock macros. In contrast, our approach does not require any direct source code instrumentation of locking structures and is based on exploiting hardware performance counters which enable fine-grained overhead control by varying the sampling frequency.

2.3.5 KernInst

KernInst [43] is a dynamic instrumentation framework designed for debugging, profiling, and application tuning. KernInst was the first to implement probe-based dynamic instrumentation in the kernel. Because it targeted the UltraSparc RISC architecture, which is a fixed-length ISA, KernInst was able to safely implement probes with branch instructions. Although this tool was only evaluated on the UltraSparc, Tamches and Miller proposed using trap instructions for redirecting control on x86. The current code release includes an x86 implementation which uses this trap-based strategy. Due to the high overhead associated with using traps, this tool is only usable with a small set of probes, thus limiting the scope of code being monitored. In contrast, KOV can simultaneously monitor all user and kernel level code with minimal overhead (see Section 4.3).

2.3.6 Kprobes

Kprobes [32] uses dynamic instrumentation to insert probes in the form of trap instructions. Using probes, this tool can instrument arbitrary code with extra code to record information. Because execution is redirected to instrumentation routines by means of a trap and hash table lookup, instrumentation is heavyweight. To alleviate this, a patch called Djprobes is currently under development, which allows overwriting some addresses with a 5-byte jump instruction, enabling direct jumps to instrumentation code. The most prevalent shortcoming of this tool is its overhead. In contrast, KOV experiences minimal runtime overhead even when simultaneously monitoring various metrics (see Section 4.3).

2.3.7 DTrace

DTrace [16] is an instrumentation framework for the Solaris operating system, designed for use with production systems. DTrace instrumentation works by inserting jump-based trampolines on fixed-length RISC architectures, but uses the same trap mechanism as KernInst or Kprobes on variable-length ISAs. Anecdotally, DTrace runs quite fast on Sparc architectures, however we expect it would suffer similar overheads as Kprobes on x86 because of the need to use trap instructions. DTrace is also able to dynamically instrument both user and kernel-level code. Because DTrace is intended for use in production systems, it guarantees that user instrumentation cannot cause additional system failures. User-supplied instrumentation code is expressed in a C-like high-level control language which enforces safety. DTrace makes it easy to monitor system resources, allowing system administrators to quickly identify the causes of system sluggishness, or to examine the otherwise unattainable system resources used by software (e.g., the number of I/O requests per second).

This tool suffers from the same drawbacks as KernInst and Kprobes when considering the scope and overhead trade-off associated with performance monitoring. As the amount of code being monitored increases, so does the number of probes. Consequently, monitoring overhead is directly proportional to the amount of code being monitored. In contrast, KOV can simultaneously monitor all user and kernel-level code with minimal overhead (Section 4.3). KOV controls overhead by varying the accuracy of results when adjusting the PMU sampling frequency, and not by specifying the scope of performance monitoring. Consequently, KOV makes no a priori judgements about which segments of code are expected to perform worse than others, and therefore is more comprehensive in its measurements.

2.3.8 SystemTap

The SystemTap project [34] is a joint effort by Red Hat, IBM, Intel, and Hitachi to add an easy to use front-end to Kprobes with functionality similar to DTrace, including the use of a scripting language. Instrumentation scripts can make symbolic references to the kernel, user programs, or included libraries (called 'tapsets'). Scripts are compiled into a kernel module and loaded to start the probes and handlers. Although a stable version of SystemTap is not yet released, some early adopters have found it useful. SystemTap currently uses Kprobes for low-level instrumentation. Consequently, the performance trade-offs when compared to KOV are similar to Kprobes.

2.3.9 JIFL

The JIT Instrumentation Framework for Linux (JIFL) [30] was the first instrumentation technique to instrument kernel code. JIFL was designed to alleviate the high performance cost associated with probe-based instrumentation techniques, such as Kprobes. JIFL shows the feasibility and desirability of kernel-based JIT instrumentation for the Linux kernel on an SMP machine. JIFL works by inserting instrumentation code directly into a duplicate copy of the original code. By embedding calls to instrumentation routines within the duplicate copy of the original code, the usual overhead associated with traps in the probe-based approach is eliminated. Using JIT instrumentation, JIFL outperformed Kprobes, at both micro and macro levels, by orders of magnitude when applying medium- and fine-grained instrumentation.

Although superior to other dynamic instrumentation tools in terms of performance, JIFL incurs considerable memory overhead associated with duplicating the original code. As such, the scope of performance monitoring with JIFL is limited. In contrast, KOV can monitor all user and kernel-level code simultaneously with minimal overhead (see Section 4.3).

2.3.10 PAPI

PAPI [14] is a public domain tool that is implemented on many platforms. Its main emphasis is on platform-independence rather than efficiency. The portable interface is implemented in software, and as a result it may incur significant overhead in some scenarios. PAPI also implements Hardware Performance Counter (HPC) multiplexing at user-level. A fine-grained timer is used as a means for a HPC group switch. The timer sends a signal to the process that has requested a multiplexed set of hardware events. A major limitation of this approach is that the sampling granularity must be large due to the large overhead of an HPC group switch, which requires a system call. As a result, the sampling error may become high for some applications, where a high sampling frequency is required to obtain an accurate measure of performance. In contrast, KOV has been shown to achieve accurate results while incurring minimal overhead (see Section 4.3.1). This advantage is largely the consequence of the fact that KOV uses a kernel module to interact with the PMU, and thus avoids the associated system call overhead.

2.4 K42

K42 is a high performance, open source, general-purpose research operating system (OS) kernel designed for cache-coherent multiprocessors. This OS was designed to address scalability across server systems aimed at utilizing small to very large-scale multiprocessors. Towards this end, primary focus was placed on achieving a high degree of spatial and temporal locality in code and data. K42 features a modular, object-oriented structure where each resource or entity is managed by a separate object instance [10]. Currently K42 runs only on the PowerPC architecture, and supports both the Linux Application Programming Interface (API) and Application Binary Interface (ABI).

2.4.1 Design Features

The K42 design team's initial goal was to start with a clean slate and examine what system structures were needed to achieve excellent performance in a scalable, maintainable, and extensible system [26]. In order to gain traction with the community, developers aimed at fully supporting existing applications. To this end, K42 was made fully Linux API and ABI compatible [6].

A large part of K42's design was oriented around providing an easily extensible infrastructure to tailor to emerging application requirements while at the same time providing an attractive systems research platform [38]. Various research groups have, using K42, explored new approaches in memory management, scheduling, inter-process and intra-process communication, event management, file systems, performance monitoring, scalable data structures, and dynamic adaptation [8, 13, 26, 39].

The entire operating system was designed using an object-oriented structure. Each virtual resource (e.g., virtual memory region, network connection, file, process) and physical resource (e.g., memory bank, network card, processor, disk) is managed by its own object instance. Each object encapsulates the meta-data necessary to manage the resource as well as the locks necessary to manipulate the meta-data. Therefore, global locks, global data structures, and global policies were entirely avoided. K42's modular design enables developers to confine the impact of their changes within a fixed set of modified components, thereby greatly simplifying the debugging process as well as decreasing the inherent complexity of the system as a whole.

K42 is structured around a client-server model, and much of the functionality traditionally implemented in the kernel or servers is moved to libraries in the application's address space. For example, all thread scheduling is done by a user-level scheduler library that is linked into each process. This design supports flexibility on a per-application basis. The specialization of services for a class of applications (e.g., games, scientific applications, databases, JVMs) is achieved by choosing the objects that are appropriate for the requirements of the target application and packaging them into a library. Overhead is reduced in many cases because crossing address space boundaries to invoke system services can be avoided. Also, space and time are consumed in the application rather than in the kernel or servers. For example, an application can have a large number of threads or file descriptors without consuming any additional kernel memory.

As multi-core chips become more prevalent, the scalability of the operating system becomes an important issue. K42 has been designed to achieve good multiprocessor performance through its object-oriented structure by maintaining the following characteristics. (1) Independent requests to different resources are serviced independently because there are no shared data structures to be traversed and no shared locks to be accessed, (2) locality is maintained for resources accessed by a small number of processors, and (3) the use of clustered-object technology allows widely accessed objects to be implemented in a distributed fashion.

Clustered objects are an enhanced object-oriented model supported by K42 [5, 9, 22]. Clustered objects, described in more detail in Section 2.4.2, improve access locality by enabling selective partitioning, replication, and distribution of object implementations.

The systematic integration of support for flexible data distribution on a per-object basis has yielded a simpler, incremental approach to scalable system design and implementation. The clustered object based infrastructure eases the addition of new scalable services by allowing the developer to focus initially on functional aspects through a non-distributed version, and then extend the implementation incrementally (for example, on a per-method basis) to a distributed version.

Distributed implementations can offer better scalability, but they also typically suffer greater overheads when scalability is not required. Distributed implementations also tend to optimize certain operations, improving their scalability, while increasing costs of other operations. In order to provide a means for coping with the tradeoffs of using distributed implementations, K42 enables hot-swapping, a technique for dynamically replacing a live clustered object instance with a different, but compatible, instance. This mechanism can be used to switch between shared and distributed implementations and additionally enable other forms of dynamic adaptation.

The design of K42, and the aforementioned hot-swapping capabilities, are centered around the concept of clustered objects. The next subsection explains clustered objects in greater detail.

2.4.2 Clustered Objects

Clustered objects are the building blocks of K42, and are integral to realizing scalability and customizability within the K42 operating system. Each clustered object is identified by a unique interface to which every implementation conforms [5]. K42 uses a C++ pure virtual base class to express a clustered object interface (Clustered Object interface class). An implementation of a clustered object consists of two definitions: a Root definition and a Representative definition expressed as separate C++ classes. The Root class defines the global portions of an instance of the clustered object. Every instance of a clustered object has exactly one instance of its Root class that serves as the internal central anchor or 'root' of the instance. The Representative (Rep) definition of a clustered object defines the per-processor portion of the clustered object. The Representative class implements the interface of the clustered object, inheriting from the clustered object's interface class. Each Rep has a pointer to the Root of the clustered object instance. The methods of a Rep can access the shared data and methods of the clustered object via its root pointer.

Clustered objects support distributed designs while preserving the benefits of a component-based approach [5]. A clustered object can be internally decomposed into a group of cooperating subparts, called Representatives, that implement a uniform interface, but use distributed structures and algorithms to avoid shared memory and synchronization on its frequent and critical operations. Clustered objects provide an infrastructure to implement both shared and distributed implementations of objects, and transparently to the client, permit the use of the implementation appropriate for the access pattern of the object. Collections of C++ classes are used to define a clustered object, and run-time mechanisms are used to support the dynamic aspects of the model.

Clustered objects allow each object instance to be decomposed into per-processor Representatives [1], and therefore provide a vehicle for distributed implementations of objects. Figure 2.5 illustrates a clustered object of a simple distributed integer counter. Externally, a single instance of the counter is visible, but internally, the implementation of the counter is distributed across a number of Representatives, each local to a processor. An invocation of a method of the clustered object's interface on a processor is automatically and transparently directed to the Representative local to the invoking processor. The internal distributed structure of a clustered object is encapsulated behind its interface and transparent to clients of the object. In Figure 2.5, a single instance of the counter, represented by the outer ring labeled with the counter's interface (**inc, val and dec**), is accessed by code executing on the processors at the bottom of the diagram. All

[1]A Representative can be associated with a cluster of processors of an arbitrary size, from 1 to n, and not necessarily per processor

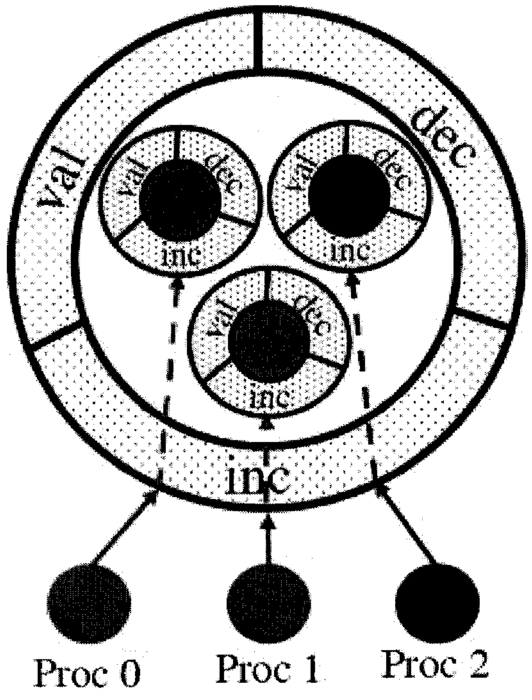


Figure 2.5: Abstract Clustered Object Distributed Counter

processors invoke the **inc** method of the instance. Transparently to the invoking code, the invocations are directed to internal per-processor Representatives, illustrated by the three inner rings in the diagram. Each Representative supports the same interface, but encapsulates its own data members. This ensures that the invocation of the **inc** method on each processor results in the update of an independent per-processor counter, thereby avoiding sharing and ensuring better performance.

At run-time, an instance of a given clustered object is created by instantiating an instance of the desired Root class. Instantiating the Root establishes a unique Clustered Object Identifier (COID also referred to as a Clustered Object ref) that is used by clients to access the newly created instance. To the client code, a COID appears to be a pointer to an instance of the Rep Class [5]. To provide better code isolation, this fact is hidden from the client code with the macro: *#define DREF(coid) (*(coid))*. For example, if c is a variable holding the COID of an instance of a clustered performance counter that has

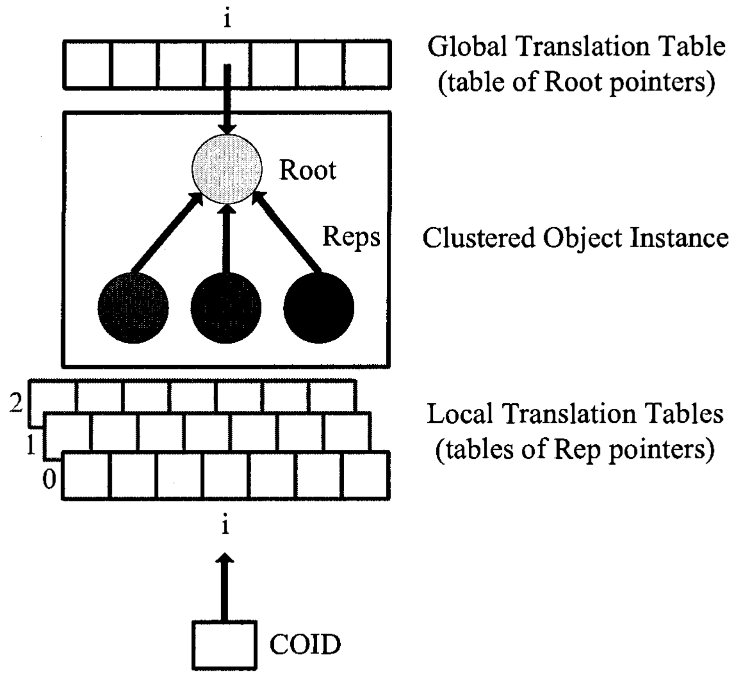


Figure 2.6: A Clustered Object Instance and Translation Tables

a method **inc**, a call would look like: *DREF(c)->inc()*.

A set of tables and protocols are used to translate calls on a COID in order to achieve the unique run-time features of clustered objects. There is a single shared table of Root pointers called the Global Translation Table (GTT) and a set of Rep pointer tables called Local Translation Tables (LTTs), one per processor. An illustration of how a clustered object's Root and Reps relate to each other is shown in Figure 2.6.

In order to avoid overhead caused by redundant creation of Reps on processors that do not use them, a Rep is not immediately created or installed into the LTTs when the clustered object is instantiated. Instead, empty entries of the LTT are initialized to refer to a special hand-crafted object called the Default Object. The first time a clustered object is accessed on a processor, the same global Default Object is invoked. The Default Object leverages the fact that every call to a clustered object goes through a virtual function table. (Remember that a virtual base class is used to define the interface for a clustered object.) The Default Object overloads the method pointers in its virtual function table to point at a single trampoline method. The trampoline code saves the current register state on the stack, looks up the Root installed in the GTT entry corresponding to the COID that was accessed, and invokes a well-known method that all Roots must implement, called *handleMiss*. This method is responsible for installing a Rep on the processor into the LTT entry corresponding to the COID that was accessed. This is done either by instantiating a new Rep or by identifying a preexisting Rep and storing its address into the address pointed to by the COID in the LTT. On return from the handleMiss method, the trampoline code restarts the call on the correct method of the newly installed Rep. The above process is called a Miss and its resolution Miss-Handling. Note that after the first Miss on a clustered object instance, on a given processor, all subsequent calls on that processor will proceed as standard C++ method invocations via two pointer dereferences. Thus, in the common case, methods of the installed Rep will be called directly with no involvement of the Default Object.

2.4.3 Memory Management

An overview of K42's overall memory management structure can be seen in Figure 2.7. Each of the objects in the diagrams has the following functionality:

1. Process - root of the object tree representing a Process in the kernel. The Process maintains a list of Regions that exist in its address space, and maintains a reference to a virtual to physical memory address translator.

2. Region - represents the mapping from a range of virtual addresses to a range of file offsets (all memory in K42 is accessed by using files).

3. File Representative (FR) - the kernel realization of a file. Facilitates communication with the external implementation of the file to do I/O and for other file system purposes.

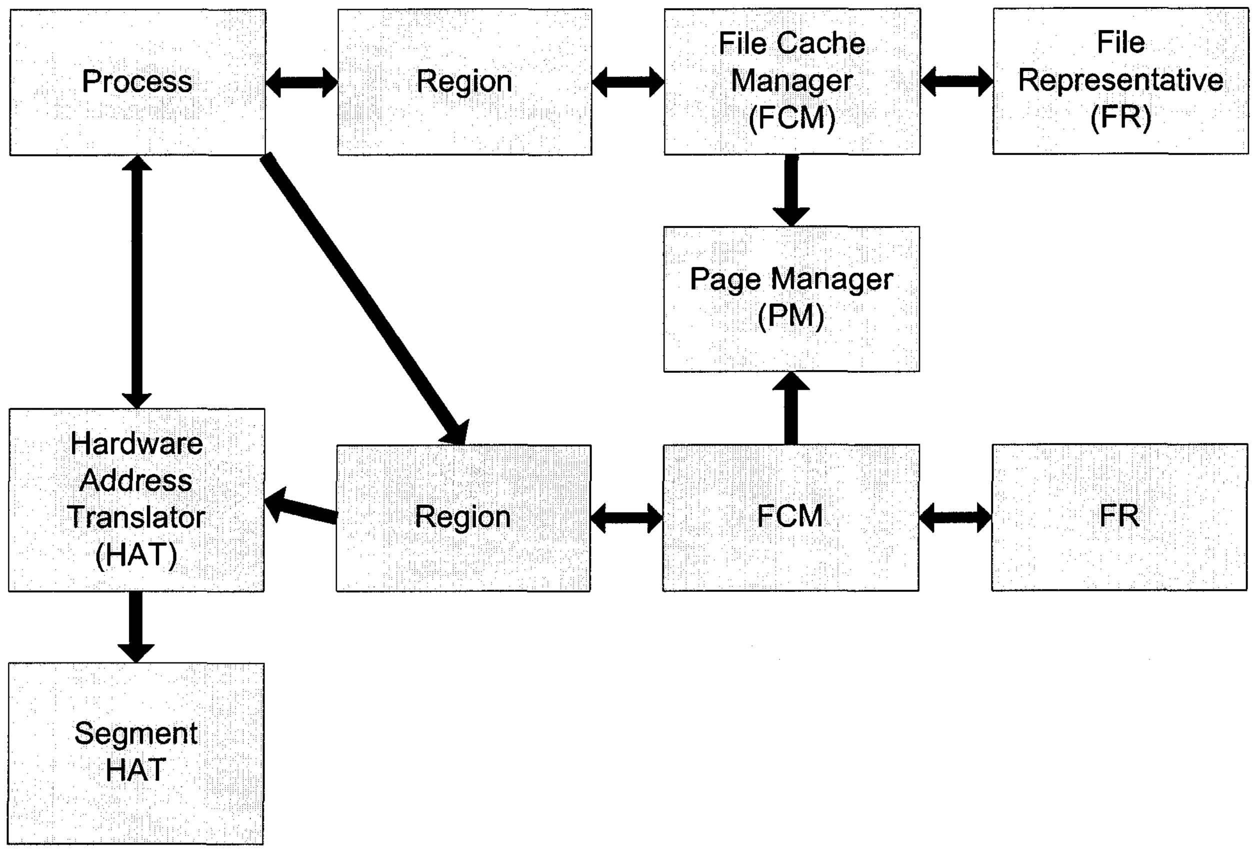


Figure 2.7: An overview of K42 memory management structure. Each K42 Process contains a single address space. The address space is made up of Regions, each of which spans a range of virtual addresses in the Process's address space. A Region maps its range of addresses onto a range of offsets in a file, handled by the FR. The FR communicates with an external file system of choice, which is a service provided by a user-level implementation. Although at first glance it seems that there is a duplicate version of the Region object in this hierarchy, the different connectivity is used to represent a specialized implementation. A Region can be used to only manage virtual addresses (top), or to implement processor-specific memory by using the processor's number in its mapping of virtual addresses to file offsets, in which case it needs the HAT (bottom).

4. File Cache Manager (FCM) - *controls the page frames currently assigned to contain* file contents in memory. It also implements the local paging policy for the file and supports Region requests to make file offsets addressable in virtual memory.

5. Page Manager (PM) - controls the allocation of page frames to FCMs.

6. Hardware Address Translator (HAT) - manages the hardware representation of an address space.

7. Segment HAT - manages the representation of a hardware segment. Segments are of hardware dependent size and hold several virtual memory pages. Their size and amount of pages they can store depends on both the underlying architecture and page granularity.

We have presented background material which relates to the work described in this dissertation, including HPCs, various instrumentation techniques, and the K42 operating system. The next chapter will describe KOV's design and implementation details.

Chapter 3

System Design and Implementation

In this chapter we present the design and implementation of the Kernel Object Viewer (KOV), a tool designed to aid programmers and operating systems designers in analyzing performance. As previously mentioned, computer systems are becoming more complex every year. This increased complexity makes it exceedingly difficult to qualitatively and quantitatively understand how well a running system is performing. Moreover, for performance-critical applications it is equally challenging to deduce the effects of code (or design) modifications on performance. We have designed KOV to aid programmers in the task of identifying performance bottlenecks and their precise causes, as well as to aid operating system designers in identifying performance bottlenecks within the operating system under different workloads.

3.1 Overview

Conceptually, the KOV performance monitoring tool is composed of two distinct parts: a monitoring infrastructure and a graphical user interface (GUI). The monitoring infrastructure obtains a list of all live processes and their objects and gathers performance data, whereas the GUI displays this data to the user.

A user interacts with KOV via the GUI. A user can issue commands in the GUI to

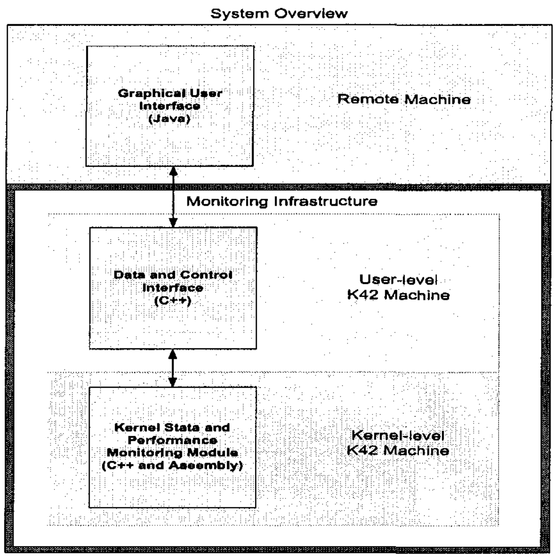

System Overview
Graphical User Interface (Java)
Remote Machine
Monitoring Infrastructure
Data and Control Interface (C++)
User-level K42 Machine
Kernel Stats and Performance Monitoring Module (C++ and Assembly)
Kernel-level K42 Machine

Figure 3.1: A high-level view of the system's architecture. Two of the bottom elements have to be on the same machine, whereas the GUI can run remotely.

obtain a list of all live processes and their objects and monitor performance. The GUI sends these commands to the monitoring infrastructure to perform the desired tasks. The monitoring infrastructure receives these commands and subsequently starts gathering data from the system. This data is stored internally in the monitoring infrastructure. The GUI periodically polls the monitoring infrastructure for this data and subsequently displays it to the user.

An overview of KOV's design is shown in Figure 3.1. The figure shows the GUI, and the two components that implement the monitoring infrastructure. The following sections describe these two components and the GUI in more detail.

3.2 Monitoring Infrastructure

The monitoring infrastructure of KOV is comprised of two components, a data gathering component which resides in the kernel, and a communication interface component which resides in user-space. Both of these components must run on the machine being monitored. The aforementioned components will be referred to as:

1. the Kernel State and Performance Monitoring Module (KM), and

2. the Data and Control Interface (DCI),

respectively, in the remainder of this dissertation.

The KM is a kernel module which maintains a list of all objects in the system and monitors the performance of the system. Objects are identified by obtaining a list of all live processes from the kernel and scanning the list of all instantiated objects within each process. The KM monitors performance by using the processor's Performance Monitoring Unit (PMU) (please refer to Section 3.2.1) and by using software instrumentation. A kernel module is necessary to perform these tasks because only the kernel process, which is privileged, is able to access the PMU. All the information gathered by the KM is stored internally in a set of data buffers. These buffers are subsequently read by the DCI.

The DCI is the link between the KM and the outside world. The DCI communicates with the KM by issuing system calls. Communication with the KM is comprised of passing control parameters to the KM, and retrieving the list of all live processes and their objects, and performance data from the KM's data buffers. Communication with the outside world starts when a GUI connects to the DCI.

The DCI uses the network for communication with the outside world by listening for connections on a specific network port. This allows the DCI to facilitate performance monitoring from remote workstations. It is preferable to have the ability to monitor a system's performance remotely because such a setup enables the user to monitor many machines, located at varying geographical locations, from a single machine.

The following subsections describe the KM and DCI in greater detail.

3.2.1 Kernel State and Performance Monitoring Module

The KM is solely responsible for obtaining object information and performance data from the system being monitored. It is designed to work with K42, an object-oriented operating system. Therefore information about the system can be extracted by obtaining a list of all live processes, and scanning all instantiated objects within each process. Performance monitoring data is gathered with the use of the architecture's PMU and software instrumentation. Consequently, the KM can perform the following tasks:

1. perform a system scan by obtaining a list of all live processes and scanning their objects,

2. configure and extract data from the PMU, and

3. obtain information through software instrumentation.

Object Scan

The list of objects is constructed by obtaining a list of all live processes and subsequently scanning all instantiated objects within each process. The KM obtains a list of all live processes by accessing K42's ProcessAnnex object. The ProcessAnnex object resides in the kernel and keeps track of all live processes by maintaining a list of Process objects. A Process object is the kernel's representation of a process and contains information ranging from the process' identification number (PID), memory regions assigned to the process, and other information.

Using the Processes' PIDs, the KM is able to scan for all instantiated objects within each process. The object scan obtains two pieces of information. The first is a list of all instantiated objects within a particular process, and the second is the object's state (specifically, references to other objects). The first piece of information lets the user know what objects a particular process has instantiated (for example, one Process object, five Region objects, and five File objects), whereas the second piece of information lets the user know how these objects relate to each other (for example, the Process object has a list of references to all five Region objects, and each Region object has a reference to one File object). These two pieces of information allow an object hierarchy of the system to be constructed. Before we explain in detail how the object scan is performed, a brief overview of K42's design is necessary.

The K42 operating system is implemented with the use of clustered objects. All clustered objects within a process are accessed through the use of one Global Translation Table (GTT), and potentially multiple Local Translation Tables (LTTs) (please refer to Section 2.4.2). A clustered object will have a reference to the Root object in the GTT and each instantiated Representative object (Rep) will have a reference in a LTT. The Reps handle requests from local processors and store local data, and the Root object stores all shared data for that clustered object. A part of a Rep's local data are references to other clustered objects.

The aforementioned object scan proceeds as follows:

1. references to the Root object of each clustered object are obtained by traversing the entire GTT,

2. for each clustered object a corresponding local Rep is found for the current processor by using the LTT,

3. The Rep object is called to obtain its state.

Obtaining all kernel-level clustered objects is straight-forward since the KM is running in kernel-space. User-level clustered objects are obtained by following the same set of steps described above by using inter-process communication (IPC). The IPC call invokes the clustered object manager for the targeted user-level process and performs the three object scan steps described above.

The Translation Tables for each process are accessible through a clustered object manager. The clustered object manager maintains the GTT and LTTs for a particular process. There is an instance of a clustered object manager for each process. The clustered object manager class was extended to allow the KM access to the Translation Tables.

Each clustered object Rep class has been extended with new methods to provide information about its state. The methods *getNumChildren()* and *getChildren()* are invoked by the KM for each Rep. The returned information is a list of clustered object references this Rep contains.

The only difference between a kernel-level and a user-level object scan is the way in which scan results are saved. The KM stores object scan results performed at kernel-level in a dedicated buffer which resides in kernel-space. Because the KM uses IPC to obtain scan results from user-level processes, scan results are first saved into a shared region [1], and then copied into a dedicated buffer in the kernel. The user-level process writes the results of its object scan in the shared region, and the KM subsequently reads this data, and saves it in its kernel-level buffer. Information from these dedicated buffers is subsequently read by the DCI.

We have discussed how the list of all live processes and their objects is obtained; in the following sections we discuss how the KM monitors performance using the PMU and software instrumentation.

Performance Monitoring Unit

Today's execution cores come with sophisticated Performance Monitoring Units (PMUs). A typical PMU will have several Hardware Performance Counters (HPCs) (e.g., PowerPC 970FX has eight), where a single HPC can be configured to count one type of event from a subset of all hardware events at any given time. For example, an HPC can accumulate occurrences of branch mispredictions or TLB misses.

The use of PMU interrupts for performance monitoring is a two step process. The first

[1]A shared region is a segment of memory accessible by both the user- and kernel-level processes

step involves configuring the PMU to increment a HPC when a desired hardware event occurs. The KM configures the PMU by writing to a set of Special Purpose Registers (SPRs). Values written to these SPRs determine which events are counted by each HPC, and which HPCs can generate interrupts. PMU interrupts can be generated when a specific HPC overflows, or periodically after executing a specified number of processor cycles.

The second step of using PMU interrupts for performance monitoring involves the Interrupt Service Routine (ISR), which is invoked whenever a PMU interrupt occurs. The ISR is a kernel routine which runs at exception-level and is capable of performing low-level tasks, such as reading from, and writing to, SPRs and HPCs, or saving data to buffers for later analysis. We wrote our own ISR to allow for performance monitoring of the system at object-level granularity. Our ISR performs three tasks:

1. Extracts an object reference from the interrupted process' stack,

2. Saves the object reference in a dedicated buffer, and

3. Saves the address of the instruction at which the interrupt occurred in a dedicated buffer for later analysis.

The PMU generates interrupts at the hardware level when a software program is executing code within a kernel or user level process. When a PMU interrupt occurs, execution of this program is halted, and the ISR starts executing. If the interrupted program was written in an object-oriented programming language, such as C++, then the program being executed has a reference to an object which stores the program's data. By C++ convention, this object reference (which we call the 'context object reference') is stored in a specific location on the program's stack. Figure 3.2 shows the data stored on a function's stack for the PowerPC architecture. By C++ convention, the 'context object reference' is stored in location **sp + 0xb0**. The ISR can therefore extract the context object reference by performing a stack walk in the interrupted program [2].

The context object reference extracted by the ISR by performing a stack walk is then saved in a dedicated buffer that was instantiated when the PMU was first initialized. In addition to the object pointer, the ISR also uses the dedicated buffer to store the address of the instruction at which the interrupt occurred. The address of this instruction will coincide with a segment of code within this object's class. Consequently, given the HPC threshold used to generate the interrupt and the extracted object pointer, performance measurements can be attributed to specific object instances within the system. Using the instruction address, performance measurements can also be attributed to exact code segments within the object's class, thereby specifying any performance bottlenecks with greater accuracy. All data from this buffer is subsequently read by the DCI.

The dedicated buffer used to store PMU information is implemented as a simple wrap-around buffer class. This buffer class has been designed such that one writer and one reader can use it concurrently. Hence, a single HPC can write to this buffer at any time, and the DCI can read from it at any time. There is a single dedicated buffer instance per HPC to simplify the task of identifying which information in the buffer was written by which HPC, which is important when matching PMU events to specific object instances. Consequently, since on the PowerPC 970FX there are eight HPCs per processor, there are also eight dedicated buffers per processor. The size of this dedicated buffer is set during initialization. The size is approximated to allow the buffer to hold at least one second worth of HPC information, because the DCI reads information from these buffers once every second. The default buffer size is 8KB, where each sample is composed of one 64 bit object pointer, and one 64 bit instruction address. Therefore, a default buffer can hold 512 samples (8192 bytes / 16 bytes = 512). This size was determined experimentally to be sufficient for most PMU configurations. If an HPC generates more than 512 samples

[2]Non object-oriented programs are monitored using the same technique; however, the value extracted from location **sp + 0xb0** will not correspond to a valid COID. All values extracted from the ISR are checked against COIDs extracted from the GTT by the GUI to verify that the object pointer is valid.

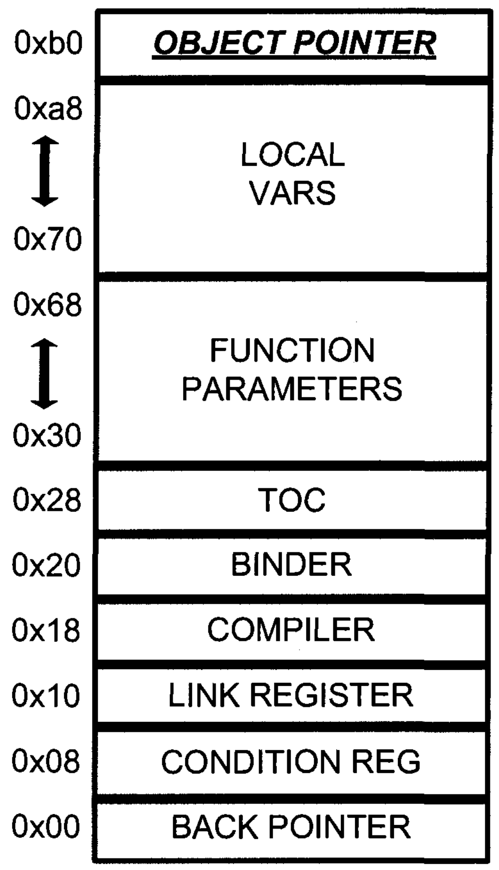


Figure 3.2: This figure shows the data stored on a function's stack. A stack pointer (**sp**) holds the address of the location **0x00**. Since the stack grows down, i.e. the stack address decreases for each new function call, data for the current function is stored above the current **sp**. The 'context object reference' is stored at address **0xb0** above the current **sp**.

within a second, only the first 512 samples are stored, whereas the rest are lost. When a sample is lost, the number of lost samples is counted. When the DCI reads information from each of the dedicated buffers, it checks if any samples were lost. If samples were lost a new, larger, buffer is created to replace it. Given the old buffer's capacity, and the number of lost samples, the new buffer size is determined by incrementing the old buffer's size by 4KB until the new size can hold all the samples [3].

We have discussed how the PMU is used to monitor performance. The following section will describe how the PMU is used to specifically monitor spin lock contention.

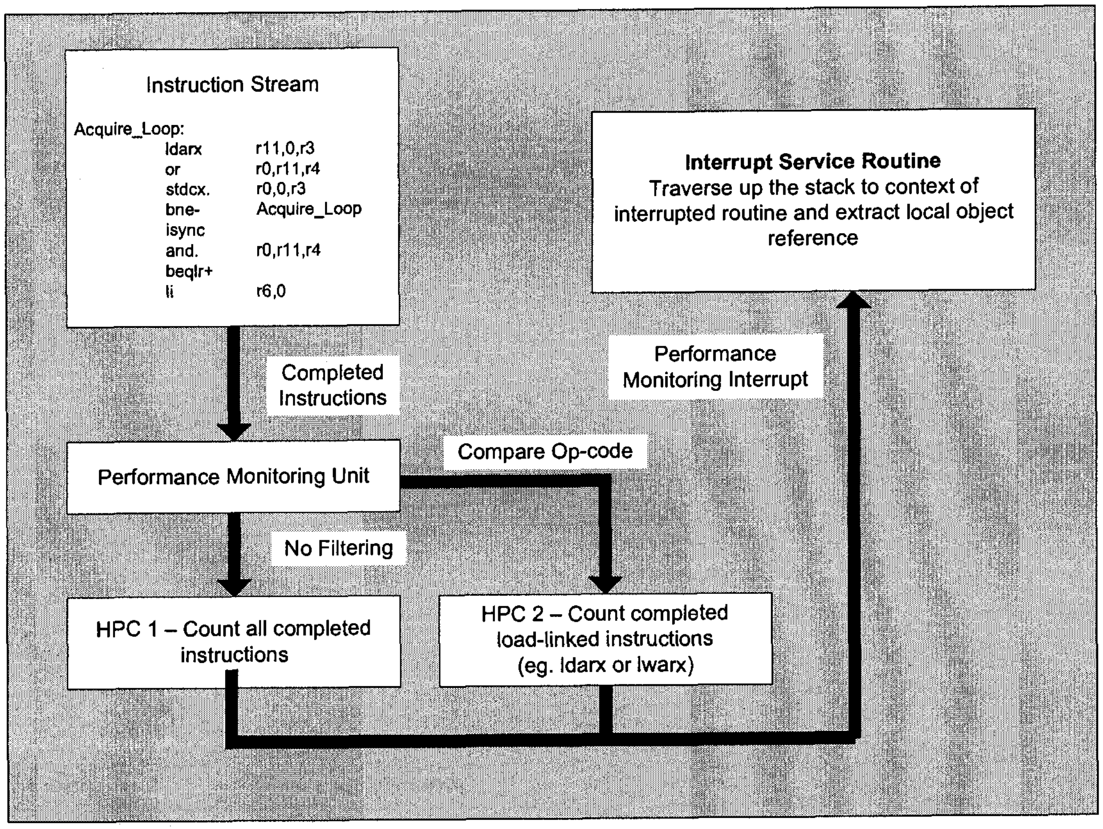
Spin locks

A novel contribution of this dissertation is the monitoring of spin lock contention without requiring any changes to the target code. Spin lock contention is measured with the use of the same ISR as mentioned above. The key lies in the PMU event that is used to trigger the ISR. The PMU enables instruction op-code comparisons of up to six partial op-codes and one full op-code entry. A match results in an increment to an HPC and upon overflow causes an interrupt to occur. For the monitoring of spin locks, instructions of interest are the load-linked (`ldarx/lwarx` in PowerPC) and store-conditional (`stdcx/stwcx` in PowerPC) instructions, which are used to implement spin locks. Figure 3.3 shows the control flow in the PMU when monitoring load-linked instructions and all completed instructions.

To estimate the amount of lock contention as a percentage of total execution time, it is necessary to count the time spent executing spin locks. To estimate the time spent executing spin locks, one must consider how spin locks work.

Spin locks operate by first reading a specific memory location into a register using a load-linked instruction. Secondly, the value in the register is tested, and set to another value if the tested condition holds true. Thirdly, the modified value in the reg-

[3]4KB is the size of a virtual memory page


Instruction Stream
Completed Instructions
Performance Monitoring Unit
No Filtering
Compare Op-code
HPC 1 – Count all completed instructions
HPC 2 – Count completed load-linked instructions (eg. ldarx or lwarx)
Performance Monitoring Interrupt
Interrupt Service Routine
Traverse up the stack to context of interrupted routine and extract local object reference

```
Acquire_Loop:
        ldarx     r11,0,r3
        or        r0,r11,r4
        stdcx.    r0,0,r3
        bne-      Acquire_Loop
        isync
        and.      r0,r11,r4
        beqlr+
        li        r6,0
```

Figure 3.3: An illustration of the control flow within the PMU when configured to monitor spin lock contention. HPC 1 has been configured to count all instructions. HPC 2 has been configured to count only load-linked instructions (specified by op-code). Once an HPC threshold is exceeded, an interrupt will be generated. The interrupt service routine will extract the context object reference from the program's stack.

ister is written back to the same memory location using a store-conditional instruction. The store-conditional instruction will succeed only if the memory location has not been changed by other threads running concurrently. Thus, if successful, we are guaranteed that the above three steps were done atomically with respect to other threads. If unsuccessful (*stdcx.* in Figure 3.3 is not executed), the code will need to 'spin' back and start with the first step again. This loop is executed indefinitely until the store-conditional succeeds, hence the term 'Spin Lock'.

The KM monitors spin lock contention by configuring the PMU to count the number of load-linked instructions executed. Each load-linked instruction executed represents four instructions being executed. These instructions are load, test-and-set, store, and branch. Hence, when estimating the percentage of total execution time spent in spin locks versus other code, the number of counted load-linked instructions is multiplied by four and compared to the total number of executed instructions (as counted in a second HPC). Once a HPC counts a sufficient number of load-linked instructions (as defined by a HPC threshold), an interrupt will be generated. The interrupt service routine will extract the context object reference from the program's stack. Subsequently, this object reference is used to attribute lock contention to specific object instances.

A manual inspection of K42's binaries and user libraries has confirmed that the load-linked instruction is not used for any other purpose than to implement locking structures. However, if any user-level applications running on K42 were to use load-linked instructions for purposes other than to implement spin locks, it is possible that the execution of these load-linked instructions might introduce errors into measurements performed using our technique. However, if such load-linked instructions are not used within a loop as described above, then it is not likely that they would significantly affect the accuracy of our results. Nevertheless, to guarantee a greater accurary of results, the user can mark any process with the PMU to exclude it from monitoring. Consequently, only the processes of interest and the operating system will be monitored for spin lock contention.

Software Instrumentation

In addition to monitoring performance with the PMU, the KM also uses software instrumentation to monitor other events. For example, we have added software instrumentation to monitor the use of sleep locks, as well as to monitor the number of object invocations. The rest of this subsection describes each of these software instrumentation techniques in greater detail.

A sleep lock is a technique of providing access to shared data where all threads except the owner of the lock are prevented from accessing the data. Consider an example implementation of a sleep lock where only one thread can hold the lock at any given time. Assume that a thread acquires the lock successfully and subsequently starts performing its operations. When a second thread tries to acquire the lock while it is being held by the first thread, the second thread will add itself to a wait queue and stop executing by going to sleep. As more threads try to acquire the lock while it is held, they will similarly add themselves to the wait queue and stop executing. When the first thread completes its work and relinquishes the sleep lock, it will notify the wait queue that the lock is free. The notification wakes up the thread at the head of the queue, which then removes itself from the queue, and acquires the lock. This process continues until there are no more threads on the wait queue.

K42 provides an implementation of the aforementioned queue operations in the BlockedThreadQueues class. The BlockedThreadQueues class provides a method for the enqueue, dequeue and wake-up operations.

Performance monitoring of sleep locks has been implemented by adding static instrumentation to the BlockedThreadQueues class that logs all enqueue and dequeue operations, by saving the queue pointer and the thread identification number in a buffer. The buffer used by this instrumentation is the same type of buffer as used by the PMU. Given this information, the length and members (threads) of each queue can be deduced.

There is a dedicated log buffer for each BlockedThreadQueues object instance, created when monitoring of sleep locks is enabled. Since performance monitoring of sleep locks is implemented using static instrumentation, the instrumentation code is always present. However, events are logged only when the KM sets the enable flag. Due to the object-oriented nature of this implementation, the enable flag can be set only in specifically targeted lock objects, thus limiting overhead.

We have described how the KM monitors sleep locks; in the rest of this section we will describe the software instrumentation used to monitor object invocations.

An object invocation is an access to an object's data or use of an object's method. For example, if object A accesses the data of object B, or if object A executes a method of object B, then object A has invoked object B. By monitoring the number of invocations of all the objects in the system, it is possible to identify which object's data and methods are used most frequently. If a particular object is accessed very frequently, then optimization on that object might potentially yield greater performance improvements.

As previously mentioned, K42 is implemented using clustered objects. Key operating system subsystems, such as the memory manager, scheduler, and I/O handlers, as well as user-level objects provided with libc are implemented using clustered objects. Clustered objects in K42 are always accessed using the dereference macro (DREF) (please refer to Section 2.4.2). As a result, we have added static instrumentation to the DREF macro to count object invocations. Consequently, the monitoring of object invocations is restricted to clustered object invocations.

The instrumentation consists of two additional function calls on top of the normal operations performed by DREF. The first function call retrieves the enable flag which determines whether logging of object invocations is turned on. The second function call increments an object invocation counter associated with the target object. A hashtable is used for this purpose, where the object reference is the key, and the value for each key is the total number of times this particular object reference (key) has been passed to the DREF macro.

A hashtable was used to monitor object invocations, rather than a dedicated buffer, due to the great volume of data that is generated by this instrumentation. A particular object can be invoked millions of times during the execution of a program. Using the buffer approach, this instrumentation would have generated millions of entries. Using the hashtable, there is one entry per object with the key corresponding to the object reference, and the value corresponding to the cumulative invocation count (e.g., one million). The drawback of using a hashtable rather than a buffer, is that there is no resolution over time, since samples are not time-stamped. It is possible to overcome this limitation by periodically reading the hashtable. A time-stamp is created by each read of the hashtable, creating an artificial measure of progress over time. For example, the first read after one second returns a key and value pair stating that object A has been referenced 10 times. A second read one second after the first read returns a key and value pair stating that object A has been referenced 100 times. Thus, during a period of one second, object A has been referenced 90 times. Using such a technique, a coarse-grained time-stamp can be added to data read from the hashtable.

We have discussed how KM obtains the list of live processes and their objects, and monitors performance. The following section will describe how this information is retrieved from the KM.

3.2.2 Data and Control Interface

The DCI is the link between the KM and the outside world. The DCI communicates with the KM by issuing system calls, and with the outside world through network connections. Communication with the KM is comprised of passing control parameters to the KM, and retrieving of the list of all live processes and their objects, and performance data from the KM's data buffers.

The DCI runs as a stand-alone user-level process on the machine being monitored. Before the DCI is able to perform any actions, it must first accept a connection from the

GUI. Once a GUI has successfully connected to the DCI, from either the local machine or remotely, the DCI provides the following functionality:

1. Passes control parameters to the KM,

2. Retrieves object information and performance data from the KM.

Control parameters that can be sent to the KM include PMU configuration parameters, flag settings to enable or disable software instrumentation, and requests to perform an object scan. The DCI receives these control parameters directly from the GUI.

The DCI retrieves information from the KM by issuing system calls. Separate system calls fetch performance data and object information. The DCI stores the data it has fetched from the KM in its own set of user-level buffers, which are subsequently read by the GUI.

The DCI essentially forwards information from the GUI to the KM, and retrieves data from the KM to be read by the GUI. As such, the DCI only performs rudimentary operations. Nevertheless, The DCI allows the GUI to monitor performance from a remote workstation. It is preferable to have the ability to monitor a system's performance remotely because such a setup enables the user to monitor many machines, located at varying geographical locations, from a single machine.

We have described the entire monitoring infrastructure used by KOV, including the KM and DCI. The following section will describe how data gathered by the monitoring infrastructure is displayed to the user in the GUI.

3.3 Graphical User Interface

The objective of the GUI is to retrieve object information and performance data and present it in a concise fashion. The GUI retrieves all its information from the DCI by issuing requests for data over the network.

The GUI can present information to the user in many ways. Figure 3.4 shows a sample of what the user can expect to see with the GUI when monitoring the system under the ApacheBench workload. The Kernel Object Tree on the left hand side of the screen shows a tree of objects for the process handling the Process File System, labeled '/kbin/procfsServer', with a process identification (pid) number of 13. Underneath this process label is a tree of clustered objects (sorted by name). These objects belong to this process either explicitly (i.e. created in user-space by this process for a specific purpose) or implicitly (i.e. created by the kernel on behalf of the process to keep track of memory, handle I/O, and perform other core functions for this process). Effectively, the sub-tree for each process contains all user- and kernel-level clustered objects related to a process. All objects shown in the tree are also differentiated by a suffix indicating their specific identification number, since there can be many instances of the same object class (e.g. COList).

The five windows to the right of the Kernel Object Tree show performance monitoring results for five metrics:

1. successful lock acquisition rate,
2. spin lock contention,
3. branch misprediction rate,
4. Translation Look-aside Buffer (TLB) miss rate, and
5. Instructions Per Cycle (IPC).

This particular snapshot shows the aforementioned data aggregated over all the listed objects belonging to all the listed processes, and shows how they progress over time.

Another way the user can view information with the GUI is by looking at performance results for one specific metric in greater detail. One can display a list of the most

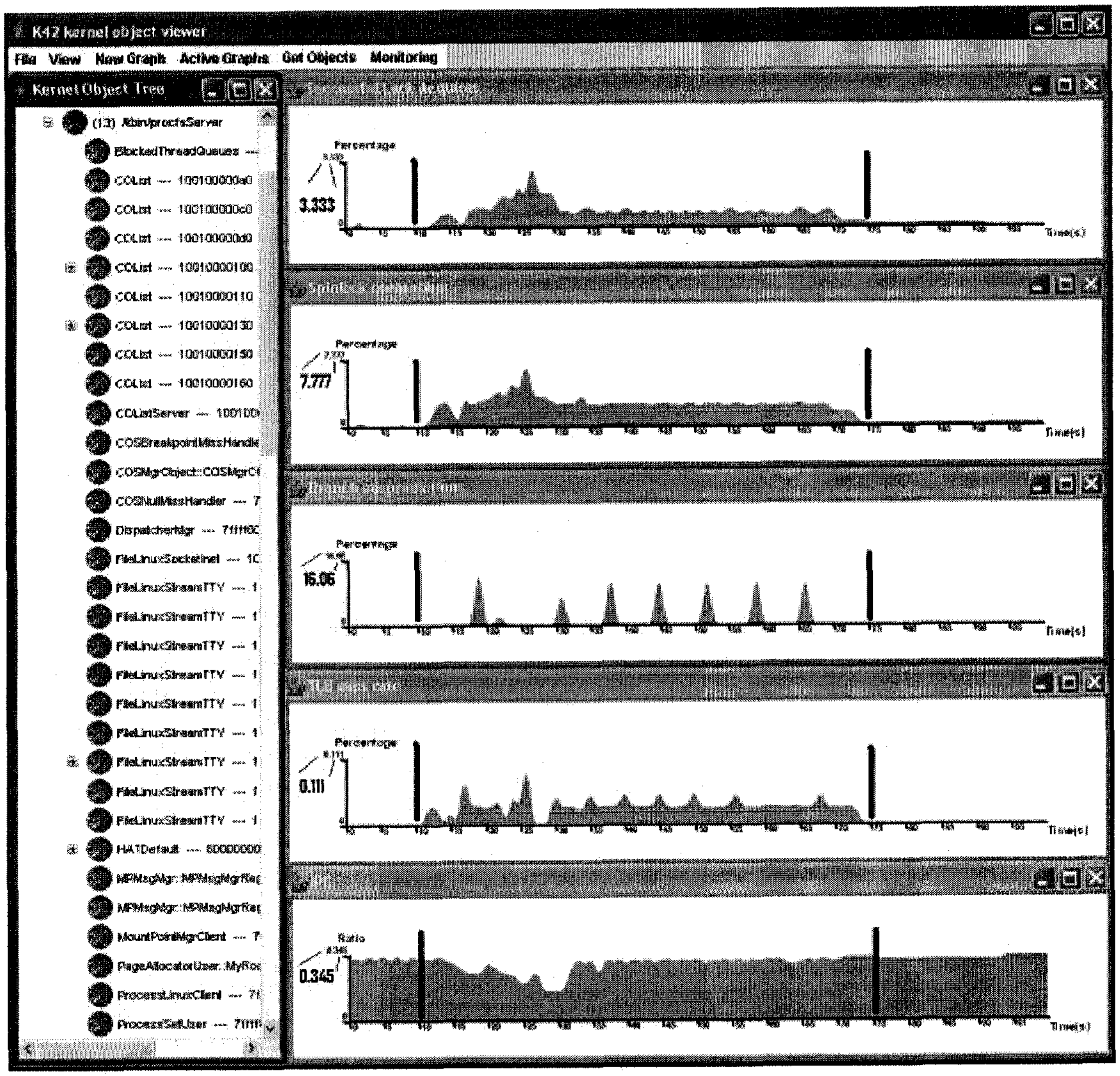


Figure 3.4: A snapshot of the complete first run of ApacheBench on the Apache web-server with five metrics from time index 10s to 75s (indicated by the vertical black lines) being measured. From top to bottom, the windows show the measured successful lock acquisition rate, spin lock contention, branch misprediction rate, TLB miss rate, and IPC respectively. ApacheBench was configured to generate a request concurrency of 70, and a total of 5000 requests.

significant contributors to a specific metric, as is shown in Figure 3.5, or the contributions of a single, specific object over time to the metric of choice as is shown in Figure 3.6.

The intended purpose of displaying information in this fashion is to use multiple forms of data representation to identify bottlenecks in the system as a whole, and also on a per-object basis. For example, Figure 3.4 referenced in the above example can be used to identify the most severe problems. Subsequently, Figure 3.5 can show the list of highest contributing objects to the results seen in Figure 3.4. Finally, Figure 3.6 breaks down an individual object's contributions to a metric of choice over time as it is tracked throughout the execution of a benchmark. One could use such a step-by-step process to determine which object causes the most lock-contention in the system, and roughly how far into the program's execution this contribution is most significant. This is just an example use of this GUI, and one can certainly do other things with the collected data such as simple logging or displaying it in an activity indicator.

The GUI and the monitoring system can be configured using the configuration utility shown in Figure 3.7. The configuration utility allows the user to configure the hardware's PMU and to toggle performance monitoring for specific objects. The prototype described in this dissertation is specific to the PowerPC 970 PMU, and the PMU configuration utility is therefore specific to this architecture's hardware events. The user can configure the PMU to monitor any and all of the events across the eight available hardware counters. The user can also enable the monitoring of object invocations and sleep lock contention, which are not tied to the underlying hardware.

The GUI is the only part of KOV that connects to the DCI. However, the communication protocol between the GUI and the DCI consists of a simple exchange of commands and data. Therefore, it would be potentially feasible to use other programs to connect to the DCI directly and obtain monitoring information on their own.

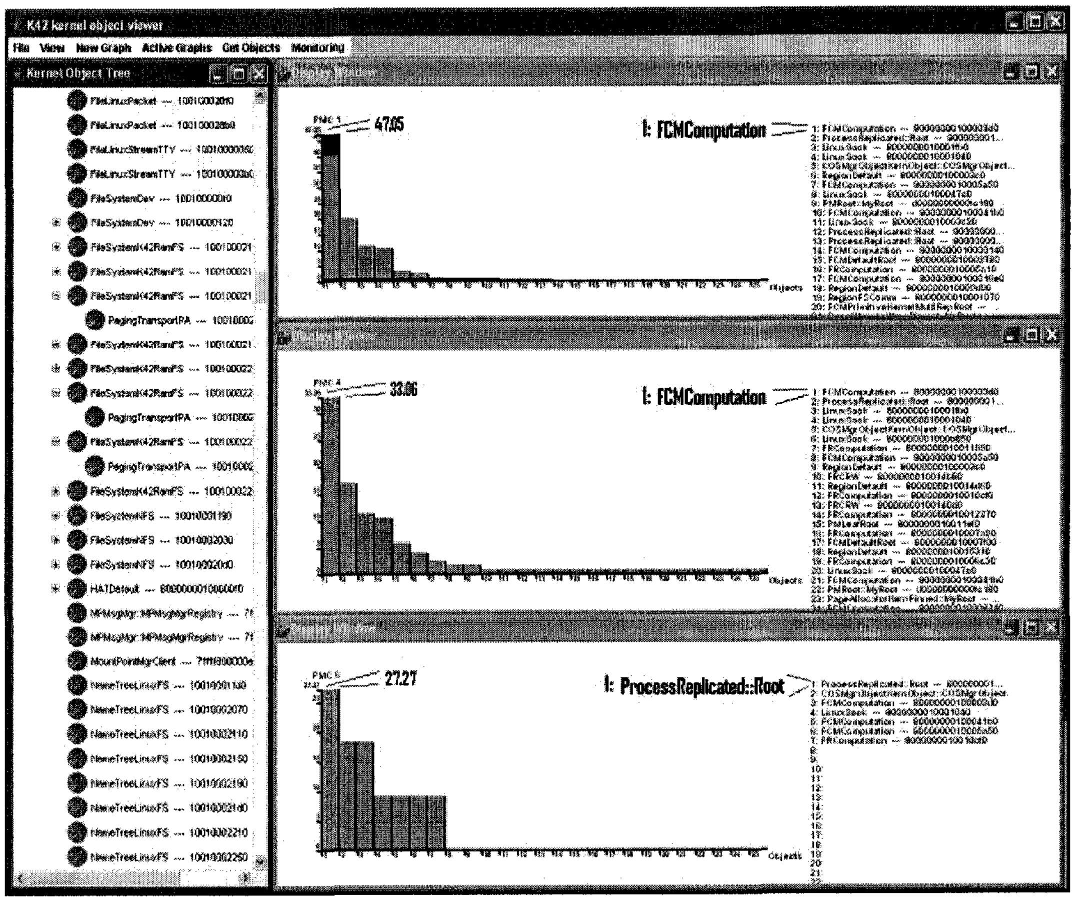
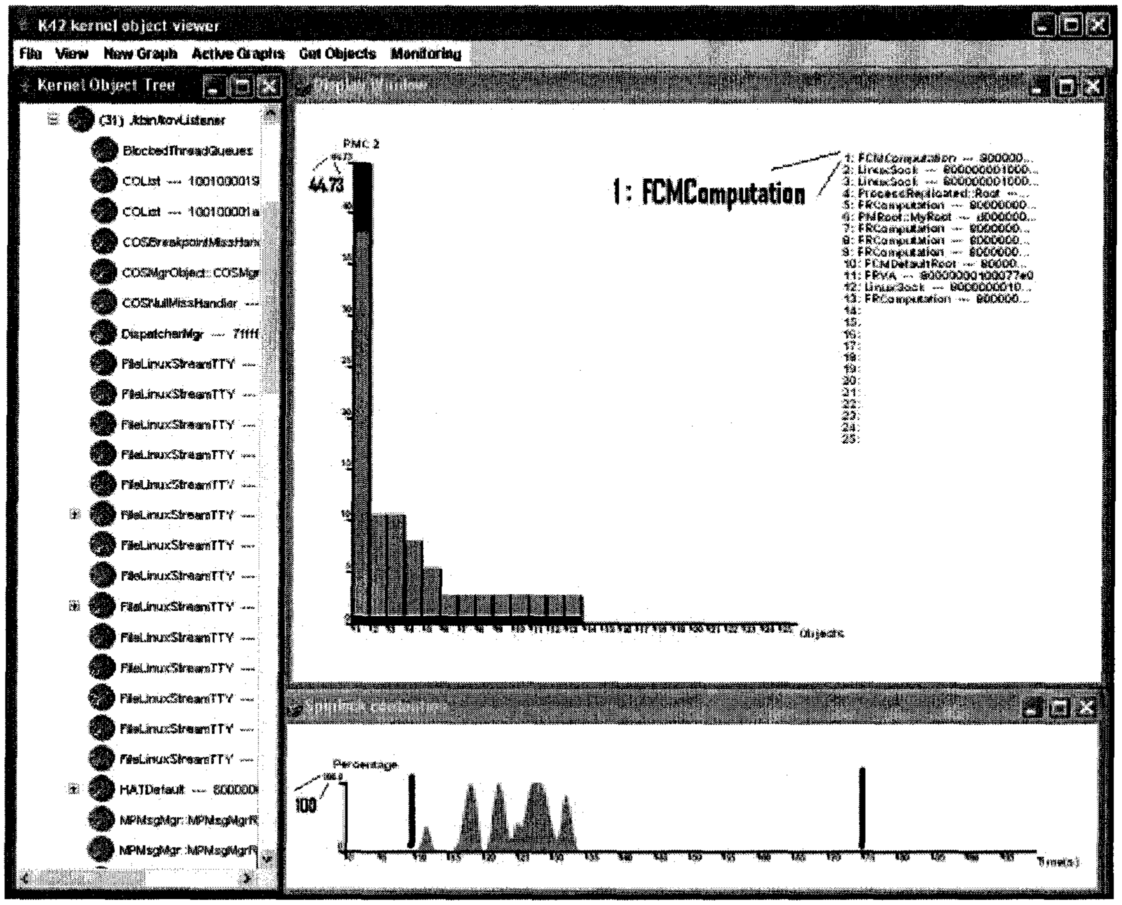
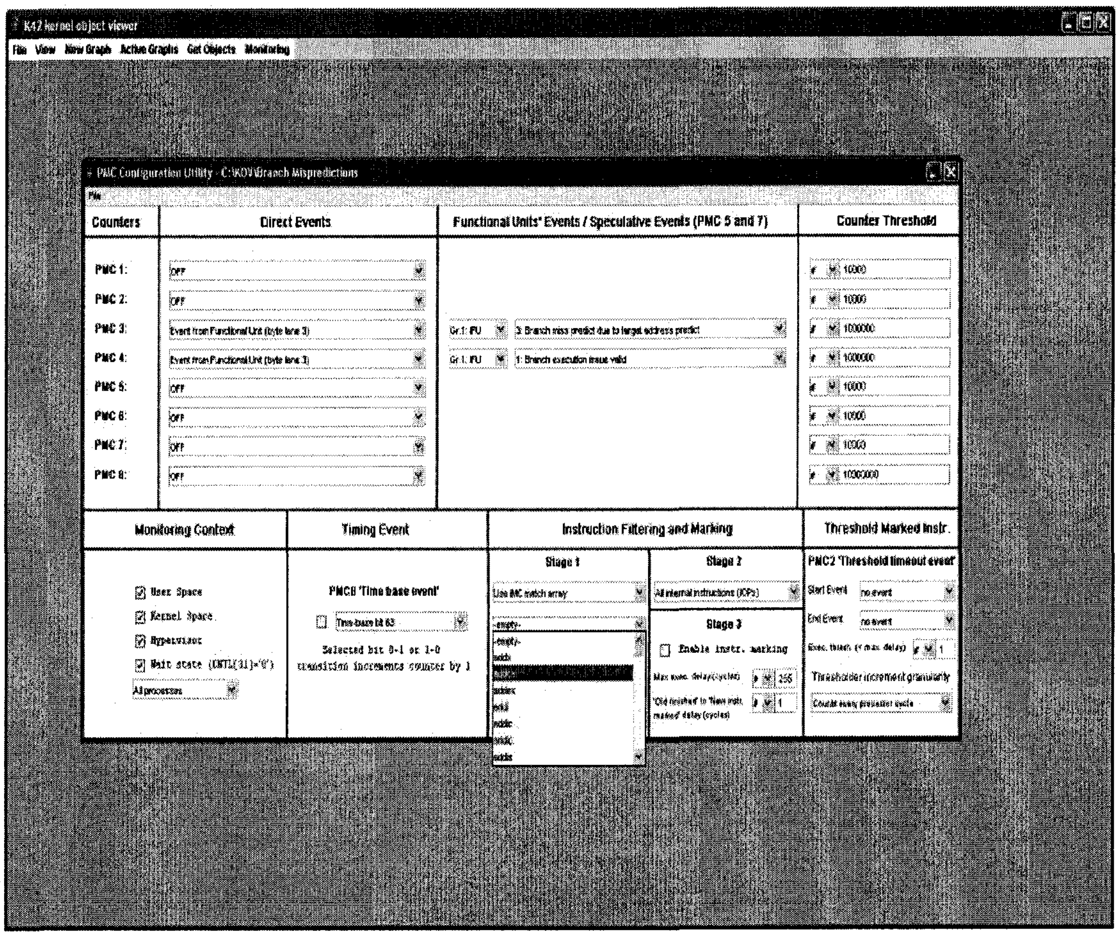


Figure 3.5: Lock contention on a per-object basis using different PMU counter thresholds when measured for the first run of ApacheBench on the Apache webserver. From top to bottom: 10k, 100k, and 1M per one sample. ApacheBench was configured to generate a request concurrency of 50, and a total of 5000 requests.



1: FCMComputation --- 900000...
2: LinuxSock --- 80000000D1000...
3: LinuxSock --- 80000000D1000...
4: ProcessReplicated::Root --- ...
5: FRComputation --- 80000000...
6: PMRoot::MyRoot --- d000000...
7: FRComputation --- 8000000...
8: FRComputation --- 8000000...
9: FRComputation --- 8000000...
10: FCMDefaultRoot --- 80000...
11: FRVA --- 800000000100077e0
12: LinuxSock --- 800000010...
13: FRComputation --- 800000...
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:

Figure 3.6: Lock contention on a per-object basis when measured for the first run of ApacheBench on the Apache webserver. The top window shows the contribution of each object to the global amount of lock contention, experienced throughout the entire run of the program. 44.73 % of all lock contention is caused by the FCMComputation object. The bottom window shows the contribution of the FCMComputation object to the global lock contention total on a per second basis. During the first 20 seconds of the run, there are periods during which 100 % of contention was attributed solely to the FCMComputation object. ApacheBench was configured to generate a request concurrency of 70, and a total of 5000 requests.


K42 kernel object viewer
File View New Graph Active Graphs Get Objects Monitoring
PMC Configuration Utility - C:\KOV\Branch Mispredictions
File

Counters	Direct Events	Functional Units' Events / Speculative Events (PMC 5 and 7)		Counter Threshold
PMC 1:	OFF			10000
PMC 2:	OFF			10000
PMC 3:	Event from Functional Unit (byte lane 3)	Gr.1: IFU	3: Branch miss predict due to target address predict	1000000
PMC 4:	Event from Functional Unit (byte lane 3)	Gr.1: IFU	1: Branch execution issue valid	1000000
PMC 5:	OFF			10000
PMC 6:	OFF			10000
PMC 7:	OFF			10000
PMC 8:	OFF			10000000

Monitoring Context

- [✓] User Space
- [✓] Kernel Space
- [✓] Hypervisor
- [✓] Wait state (CNTL[31]='0')
- All processes

Timing Event

PMC8 'Time base event'

- [ ] Time-base bit 63

Selected bit 0-1 or 1-0 transition increments counter by 1

Instruction Filtering and Marking

Stage 1

- Use IMC match array
- -empty-
- -empty-, addx, [illegible], addex, addi, addic, addic., addis

Stage 2

- All internal instructions (IOPs)

Stage 3

- [ ] Enable instr. marking
- Max exec. delay (cycles): 255
- 'Old finished' to 'New instr. marked' delay (cycles): 1

Threshold Marked Instr.

PMC2 'Threshold timeout event'

- Start Event: no event
- End Event: no event
- Exec. thresh. (< max. delay): 1
- Thresholder increment granularity: Counts every processor cycle

Figure 3.7: A snapshot of the PMC configuration utility that allows the user to configure the Performance Monitoring Unit (PMU) on the PowerPC 970FX. The setup shown here is used to configure Branch Mispredictions monitoring. Monitoring of executed instructions is also possible, and is supported by the PMU (currently being pointed to by the cursor).

3.4 Limitations

There are three limitations to the KOV performance monitoring tool. The first two are design limitations resulting from KOV's reliance on hardware performance monitoring capabilities and the use of K42. The third limitation is the result of KOV's current prototype implementation.

The monitoring infrastructure used by KOV is dependent on the processor's Performance Monitoring Unit (PMU). The PMU monitors micro-architectural events specific to the processor it is implemented on (in our case the PowerPC 970FX). Consequently, if KOV were to use the PMU on another processor, the PMU configuration utility and multiplexing logic would have to be ported to use the new architecture.

KOV has been designed for K42, an object-oriented operating system. The object scan capabilities of the monitoring infrastructure are specific to K42's design. If one were to port KOV to another operating system, the whole scan portion of our tool would have to be redesigned (and if the target operating system is not object-oriented, then the object capabilities of KOV become moot).

The current implementation of the KOV performance monitoring tool allows for the DCI to accept only one GUI connection, and similarly the GUI can only connect to one DCI at a time. This is only a limitation of the present implementation of KOV. In theory both the DCI and GUI can be extended to allow for multiple connections. The DCI can be extended to allow multiple GUIs to read the same list of all live processes and their objects, and performance data. In addition, the GUI can be extended to monitor multiple machines by maintaining separate data structures for each machine being monitored.

We have described the KOV performance monitoring tool and its limitations. We will conclude this chapter by providing a brief summary of all the components of KOV in the following section.

3.5 Chapter Summary

Figure 3.8 shows an overview of all the components of KOV and how they interact with one another. The top box shows the GUI, the only part of KOV the user interacts with directly. Via the GUI, the user can request information about the system by issuing commands to start an object scan, use the PMU, monitoring sleep locks, or monitor object invocations. The DCI receives these commands from the GUI and forwards them to the KM in the form of system calls. Subsequently, the KM starts gathering data requested by the user. All data gathered by the KM is read by the DCI. The GUI periodically polls the DCI for any data it has retrieved from the KM, and once the GUI has obtained information from the DCI, it is displayed to the user.

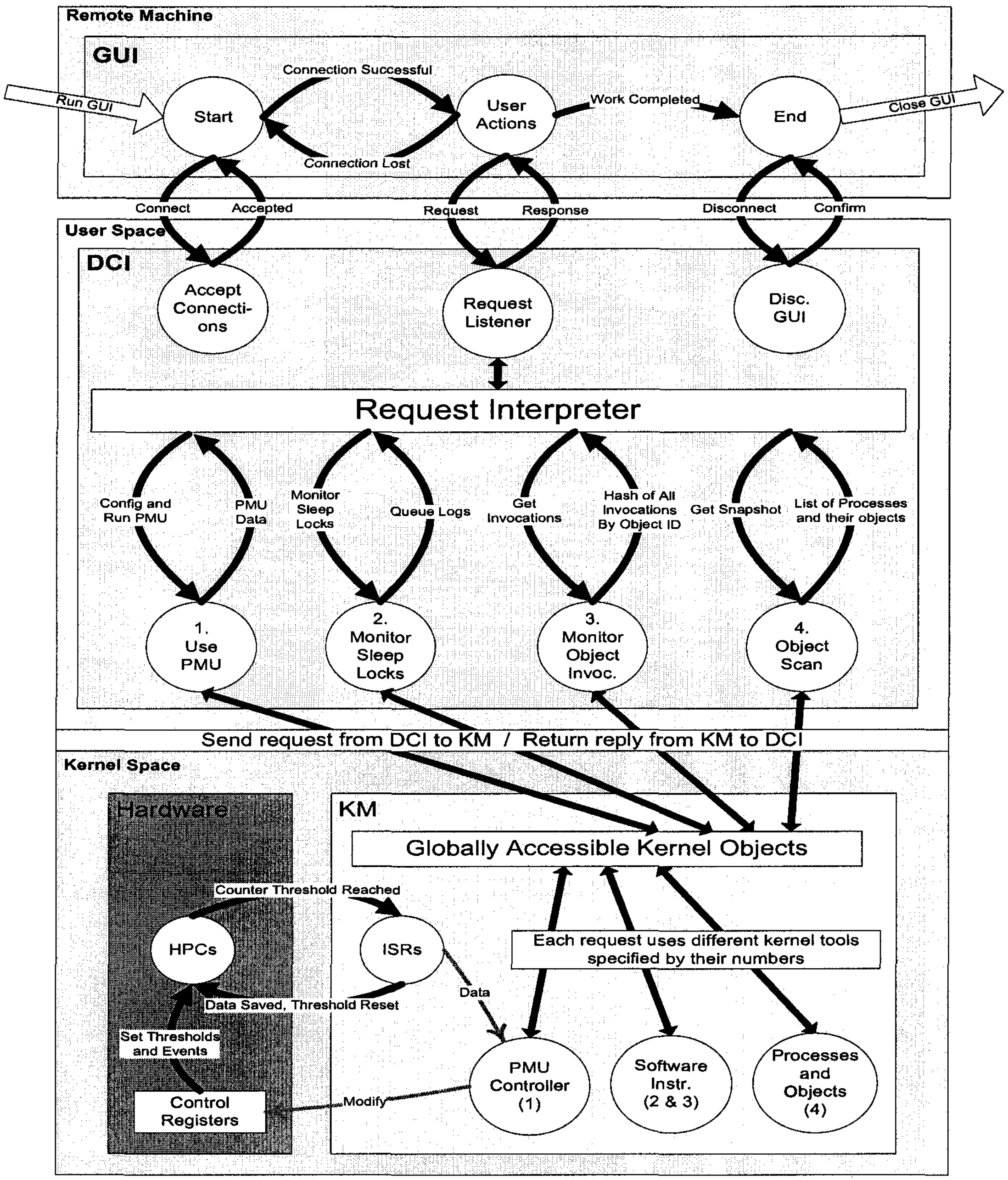


Figure 3.8: This figure shows an overview of all the components of KOV, and how they interact with one another. The top box shows the GUI, the middle box shows the DCI, and the bottom box shows the KM. The arrows represent the flow of information between different components.

Chapter 4

Experimental Evaluation

In this chapter we present the experimental evaluation of the Kernel Object Viewer (KOV), a tool designed to aid programmers and operating systems designers in performance analysis. As previously mentioned, computer systems are becoming more complex every year. This increased complexity makes it exceedingly difficult to qualitatively and quantitatively understand how well a running system is performing. In Section 4.2 we evaluate and analyze the performance of several workloads using KOV to demonstrate how KOV can be used to accurately identify a performance bottleneck, and subsequently identify the cause of the bottleneck. Ultimately, we aim to show how KOV can be used to ease qualitative and quantitative analysis of a running system. We then analyze the overheads introduced by KOV in Section 4.3.

4.1 Experimental Workloads

We use two workloads to demonstrate the feasibility of our tool. The first workload is Apache 1.3 when running under the ApacheBench 0.5 and SURGE 1.0 generated workloads. The second workload is SPECjbb 2000 running on the IBM J2SE 5.0 Java virtual machine. The two workloads are described in more detail in the following subsections.

4.1.1 Apache Workloads

Apache HTTP webserver version 1.3 is an open-source HTTP server which can run on a multitude of operating systems including Linux, Solaris, and Windows 2000 [35]. Apache is capable of handling multiple client requests simultaneously.

This version of Apache handles multiple client requests by maintaining a pool of child processes. The parent process distributes incoming requests amongst its child processes for servicing, and does not directly handle any requests itself. An initial set of child processes is created when Apache starts (by default five), and more child processes are created lazily once the number of outstanding requests exceeds the number of active child processes. The maximum number of child process that Apache creates is limited by parameters read during the webserver's initialization. Once created, child processes are not destroyed until the server shuts down. The end result of this design choice is that during periods of low server load, child processes that are not handling requests are put to sleep. As more connections are made to the webserver, and server load increases once again, the parent process will wake up existing child processes to handle outstanding requests.

Apache configuration parameters also allow the user to set the maximum number of keep-alive requests. A keep-alive request is a technique of implementing HTTP persistent connections. A persistent connection allows multiple HTTP requests and responses to be exchanged between the client and server during one connection. Normal HTTP connections allow for only one request/response pair to be exchanged. When the client first initiates a connection with the server, the client will specify that it wishes to establish a persistent connection using keep-alive requests. The server will then periodically send a keep-alive request to the client (every x seconds). The client has to respond to each keep-alive request to maintain the persistent connection. The connection remains open until the client or server explicitly terminate the connection, or the client stops responding to the server's keep-alive requests and a period of time passes (the timeout period).

All performance measurements made on Apache used the setting for a maximum of 100 child processes and 100 keep-alive requests. By setting the maximum number of keep-alive requests and child processes equal to each other, every child process is able to complete all client requests during one connection. If a workload also uses the keep-alive option when issuing its requests, then each child process will only accept one connection throughout the entire run of the workload.

To measure performance of the Apache HTTP webserver, we use two programs that generate workloads for the server to handle: ApacheBench and SURGE.

ApacheBench

ApacheBench is a command line computer program designed to measure performance of HTTP webservers [1]. ApacheBench generates a steady stream of requests, putting a constant load on the server. The user can specify the number of requests generated concurrently, and the total number of requests sent. The number of concurrently generated requests tests the amount of server parallelism available, whereas the total number of requests determines the total running time of ApacheBench.

ApacheBench was run multiple times by varying request concurrency from 10 to 150 in increments of 10, and keeping the total number of requests steady at 5000. All requests use the keep-alive option. Each particular configuration of ApacheBench was measured three times using three consecutive runs. The purpose of measuring multiple consecutive runs is to show the difference in performance resulting from initialization (e.g. process creation). The first run is expected to perform worse than the remaining two, whereas run two and three are expected to perform similarly to one another.

SURGE

The Scalable URL Reference Generator (SURGE) is a tool designed to generate a Web workload which mimics a set of real users accessing a webserver [12]. SURGE generates file references matching empirical measurements of 1) server file size distribution; 2) request size distribution 3) relative file popularity; 4) embedded file references; 5) temporal locality of references; and 6) idle periods of individual users.

SURGE can be configured in two ways. An initial set of parameters is fixed before compile time. They include:

1. Total number of requests for the most popular file, which we set to 20,000, and

2. Total number of documents to be used in the test, which we set to 20,000.

The total number of requests for the most popular file will affect the magnitude of server load at peak time. This is because the majority of requests for the most popular file are timed to coincide with peak load.

In addition to the aforementioned pre-compile parameters, the user can also vary several options dynamically, such as the number of client processes used and the number of threads per client. The user also specifies benchmark duration in seconds. Finally, SURGE requires the user to specify the directory path of the files created on the web server.

All of our SURGE tests used two clients and a runtime of 60 seconds, whereas the number of threads per client was varied from 5 to 60 in increments of 5. Each particular configuration of SURGE was measured three times using three consecutive runs.

4.1.2 SPECjbb2000 Workloads

This version of the Standard Performance Evaluation Corporation (SPEC) java business benchmark (jbb) was released in 2000 to evaluate the performance of multi-tier server-side

Java applications [2]. The benchmark runs on several versions of UNIX, Windows/NT, Linux and other operating systems.

SPECjbb2000 represents an order processing application for a wholesale supplier. SPECjbb2000 models a wholesale company, with warehouses that serve a number of districts. Customers initiate a mix of operations, such as placing new orders or requesting the status of an existing order. Additional operations are initiated by the company, such as processing orders for delivery, entering customer payments, and checking stock levels.

SPECjbb2000 assigns one active customer per warehouse. A warehouse is implemented as a unit of about 25MB of data stored in binary trees. Warehouses map directly to Java threads. As the number of warehouses increases during the full benchmark run, so does the number of threads. SPECjbb2000 measures the throughput of the underlying Java platform, which is the rate at which business operations are performed per second. A typical benchmark run takes about three minutes per warehouse. SPECjbb2000 measures throughput in a fixed amount of time, so faster machines do more work in the allotted time.

A complete run of SPECjbb2000 consists of two phases: a warm-up phase and a measurement phase. The warm-up phase creates data for all the warehouses and typically lasts 30 seconds. After the warm-up phase is complete, the measurement phase will emulate customer accesses to the warehouse as described previously.

We have configured SPECjbb2000 to use a maximum of four warehouses. We used the standard 30 second warm-up phase followed by a 120 second application-level measurement phase.

4.2 Experimental Results

This section presents results for the Apache and SPECjbb2000 workloads gathered by KOV. KOV was configured to measure:

1. successful lock acquisition rate,
2. spin lock contention,
3. branch misprediction rate,
4. Translation Look-aside Buffer (TLB) miss rate, and
5. Instructions Per Cycle (IPC).

The successful lock acquisition rate is measured by counting the number of successfully executed store-conditional instructions multiplied by the number of instructions in a spin lock (four) as a percentage of all retired instructions.

Spin lock contention is measured by utilizing the technique described in Section 3.2.1. The results for this metric show the amount of lock contention as a percentage of all retired instructions.

The branch misprediction rate is measured by counting the number of correctly and incorrectly predicted branches (also referred to as predicted and mispredicted branches, respectively). Results for this metric show the total number of mispredicted branches as a percentage of the total number of branches (predicted plus mispredicted branches).

The TLB miss rate is measured by counting the number of TLB misses caused by fetching instructions as well as reading and storing data. The results for this metric show the cumulative number of TLB misses as a percentage of all retired instructions.

IPC is measured by counting the number of retired instructions and completed processor cycles. Performance results for this metric show the total number of retired instructions as a percentage of the total number of completed processor cycles.

4.2.1 Apache loaded with ApacheBench

Figure 4.1 shows the results for Apache loaded with ApacheBench. The figure shows results for the first (left) and third (right) run of ApacheBench. The x-axis on each graph

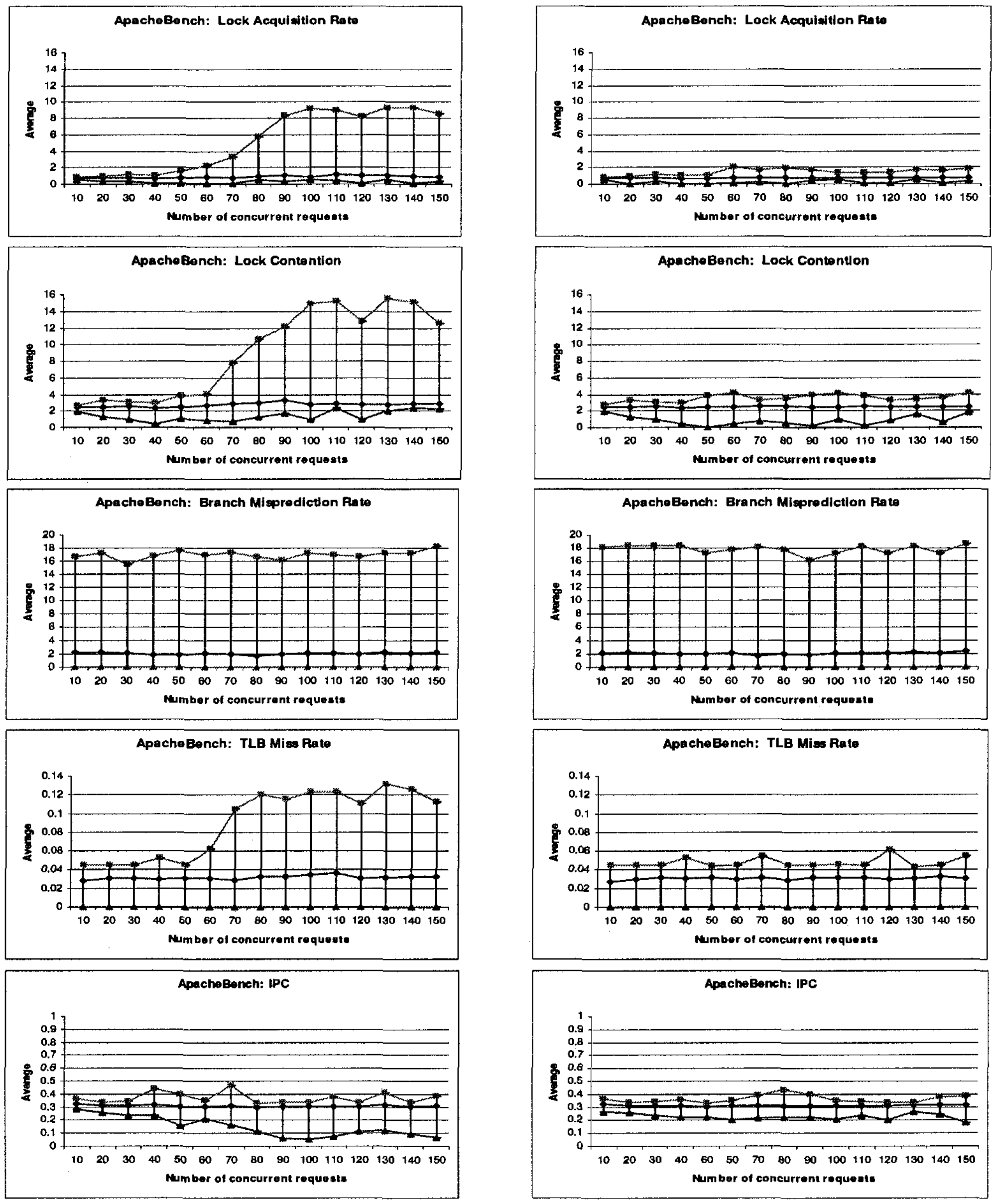


Figure 4.1: This figure depicts the monitoring results for ApacheBench for five metrics: successful lock acquisition rate, spin lock contention, branch misprediction rate, TLB miss rate, and IPC. The figure shows results for the first (left) and third (right) run of ApacheBench. The x-axis on each graph shows request concurrency, which is varied from 10 to 150 in increments of 10. The y-axis shows the percentage composition for each metric. Each individual graph shows three plots, from top to bottom, the maximum, average, and minimum values for each run.

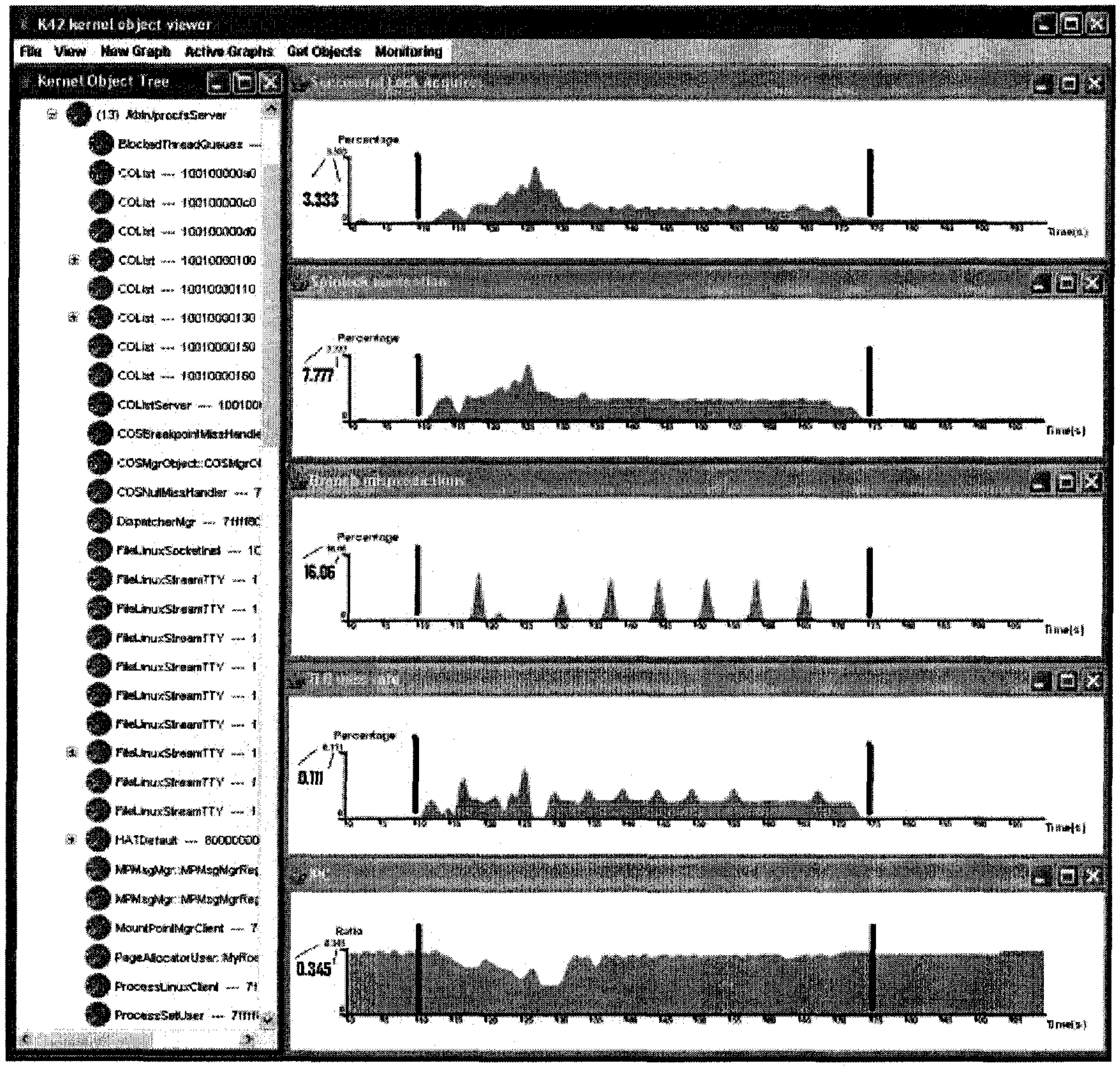


Figure 4.2: A snapshot of the complete first run of ApacheBench on the Apache webserver with five metrics from time index 10s to 75s (indicated by the vertical black lines) being measured. From top to bottom, the windows show the amount of measured successful lock acquisition rate, spin lock contention, branch misprediction rate, TLB miss rate, and IPC respectively. ApacheBench was configured to generate a request concurrency of 70 and a total of 5000 requests.

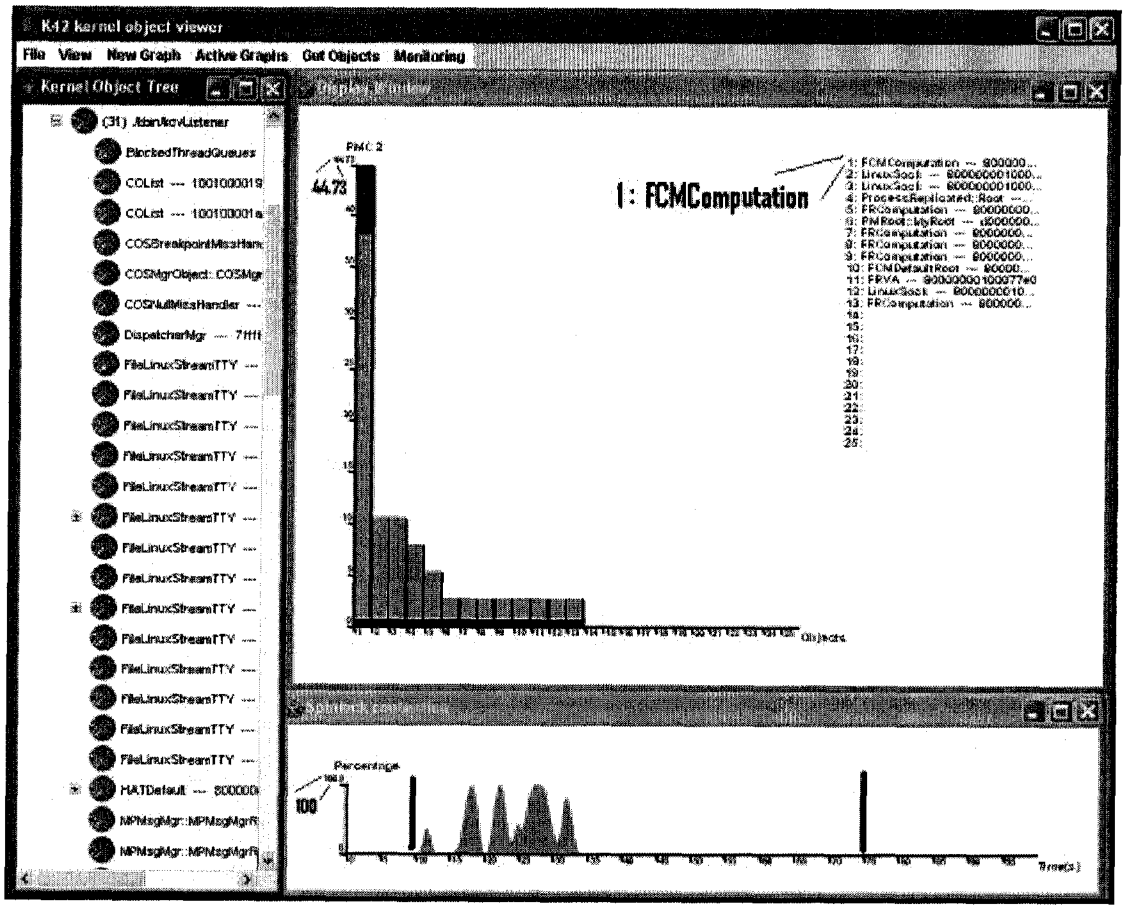


Figure 4.3: Lock contention on a per-object basis when measured for the first run of ApacheBench on the Apache webserver. The top window shows the contribution of each object to the global amount of lock contention, experienced throughout the entire run of the program. 44.73 % of all lock contention is caused by the FCMComputation object. The bottom window shows the contribution of the FCMComputation object to the global lock contention total on a per second basis. During the first 20 seconds of the run, there are periods during which 100 % of contention was attributed solely to the FCMComputation object. ApacheBench was configured to generate a request concurrency of 70, and a total of 5000 requests.

shows request concurrency as it is varied from 10 to 150 in increments of 10, whereas the y-axis shows the percentage composition for each metric (calculated as described above). The first run measures the metrics of Apache when loaded with ApacheBench as well as the overhead associated with creating child processes. Recall that Apache is initialized with only five child processes, where extra processes are created lazily once there are enough outstanding requests to warrant their creation. The third run measures the metrics of Apache when loaded with ApacheBench with no child process creation overhead (since child processes are not destroyed once created).

Before we discuss the results, we will elaborate on the significance of considering peak values, in addition to the overall average. A program, through the course of its execution, can go through several different phases of execution. Each phase can result in a different behaviour of the program and its interactions with the operating system, either because the code being executed is different, or because the actions performed by the code are different. In both cases, the average value for a metric will consider the aggregate performance results over the entire run of a program, and therefore hide the behaviour of individual execution phases.

The rest of this section discusses the results for Apache loaded with ApacheBench in greater detail.

Using performance results obtained with KOV, we can see a gradual rise in peak lock contention during the first run of ApacheBench, shown in Figure 4.1 top curve, as request concurrency increases. There is a significant difference in peak lock contention between the first run (left), where the value increases to almost 16 % (which is roughly five times the average) and the third run (right), where this increase is absent. Using KOV's GUI, we investigated the performance of the first run of ApacheBench when configured with a request concurrency of 70 to gain further insight.

The complete first run of ApacheBench configured for 70 concurrent requests is shown in Figure 4.2. Each window in Figure 4.2 shows the five metrics as obtained by KOV over the course of the entire run of ApacheBench (isolated by vertical black lines from other monitoring information).

The peak values of interest are represented by the first several seconds of the ApacheBench run (10-30s). Since the difference between the first run and the third run of ApacheBench is only the creation of child processes by the Apache webserver, the process creation phase is a likely cause of the increased lock contention. Consider the fact that ApacheBench sends 70 concurrent requests to be handled by the webserver. However, Apache was only initialized with five child processes. The Apache webserver will create an additional 65 child processes, in order to handle all remaining 65 requests.

Process creation requires a significant amount of work to be performed by the operating system. This includes allocating memory pages for the process and initializing the process' data structures. In addition to increased lock contention during the initialization period, Figure 4.2 also shows a corresponding increase in TLB misses for ApacheBench. This information suggests that the likely cause of the contention is the memory subsystem, since TLB misses are directly related to accessing memory pages. Further use of KOV's performance monitoring and visualization capabilities validates this hypothesis.

A per-object breakdown of lock contention, shown in Figure 4.3, has attributed the majority of lock contention to one object instance, namely the FCMComputation object. We identified the purpose of the FCMComputation class by investigating K42's source code. The FCMComputation class is responsible for fetching and allocating memory pages. In addition, Figure 4.3, in the bottom window, shows the amount of lock contention contributed by the FCMComputation object to the global lock contention total shown in Figure 4.2. The contribution of the FCMComputation object shown in the bottom window of Figure 4.3 is calculated on a per second basis, and shows that the contention occurs primarily during the first 20 seconds. With the contribution equaling 100 %, this indicates that all lock contention during that time period was attributed to the FCMComputation object. Considering the findings discussed above, we can conclude that the most likely cause of the increased peak lock contention was the following scenario.

As Apache creates new child processes, each process' request for new pages is handled by the memory subsystem. Since Apache's child processes are all created when the initial requests arrive, there is contention on the memory subsystem to provide pages for 65 processes as quickly as possible. As the initialization phase finishes, Figure 4.2 shows that lock contention drops to the average level shown in Figure 4.1.

We have shown how KOV can be used to identify a performance bottleneck within the operating system by using performance results obtained by monitoring the execution of Apache when loaded with ApacheBench. A key aspect of this is the unique capability of KOV to obtain metrics from specific object instances. In the following section we will show how KOV can be used to understand the execution of a program by comparing performance results of Apache loaded with SURGE, against those we obtained in this section using ApacheBench.

4.2.2 Apache loaded with SURGE

Figure 4.4 shows results for Apache loaded with SURGE for the aforementioned five metrics. The figure shows results for the first (left) and third (right) run of SURGE. The x-axis on the each graph shows the number of threads per client ranging from 5 to 60 in increments of 5, whereas the y-axis shows the percentage composition for each metric (calculated as described in a previous section). Figure 4.5 shows a snapshot of the complete execution of the first run of SURGE, when configured to generate 35 requests per client, using two clients.

Figure 4.4 and Figure 4.5 show performance results for various configurations of two runs of SURGE, and one detailed run displayed in KOV's GUI respectively, similarly to what was shown in Figure 4.1 and Figure 4.2 for ApacheBench.

By looking at Figure 4.4 and Figure 4.1 we can see that there is a similarity between

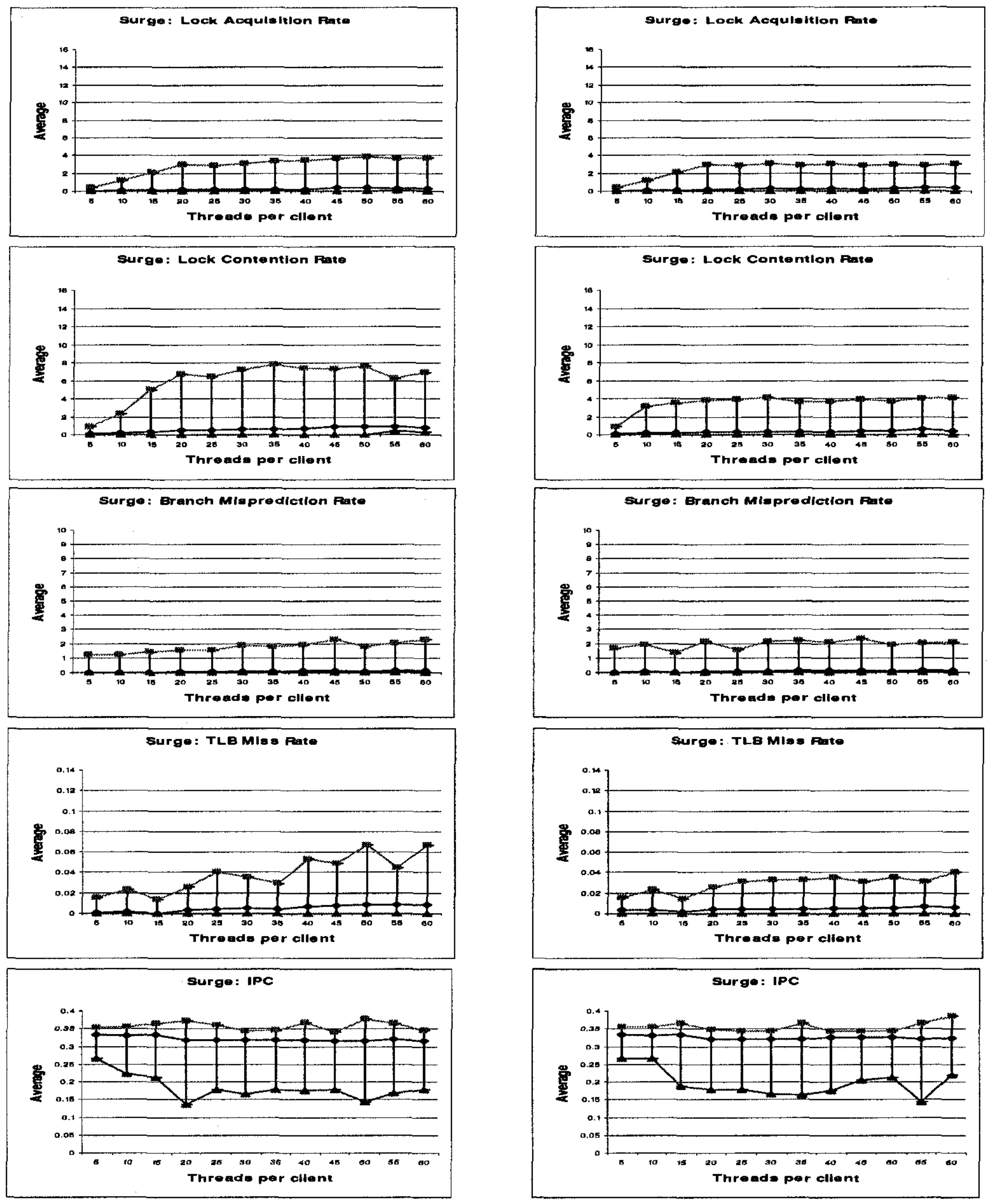


Figure 4.4: This figure depicts the monitoring results for SURGE for five metrics: successful lock acquisition rate, spin lock contention, branch misprediction rate, TLB miss rate, and IPC. The figure shows results for the first (left) and third (right) run of Apache loaded with SURGE. The x-axis on each graph shows the number of concurrent requests generated by each of the two clients, ranging from 5 to 60 in increments of 5. The y-axis shows the percentage composition for each metric. Each individual graph shows three plots, from top to bottom, the maximum, average, and minimum values for each run.

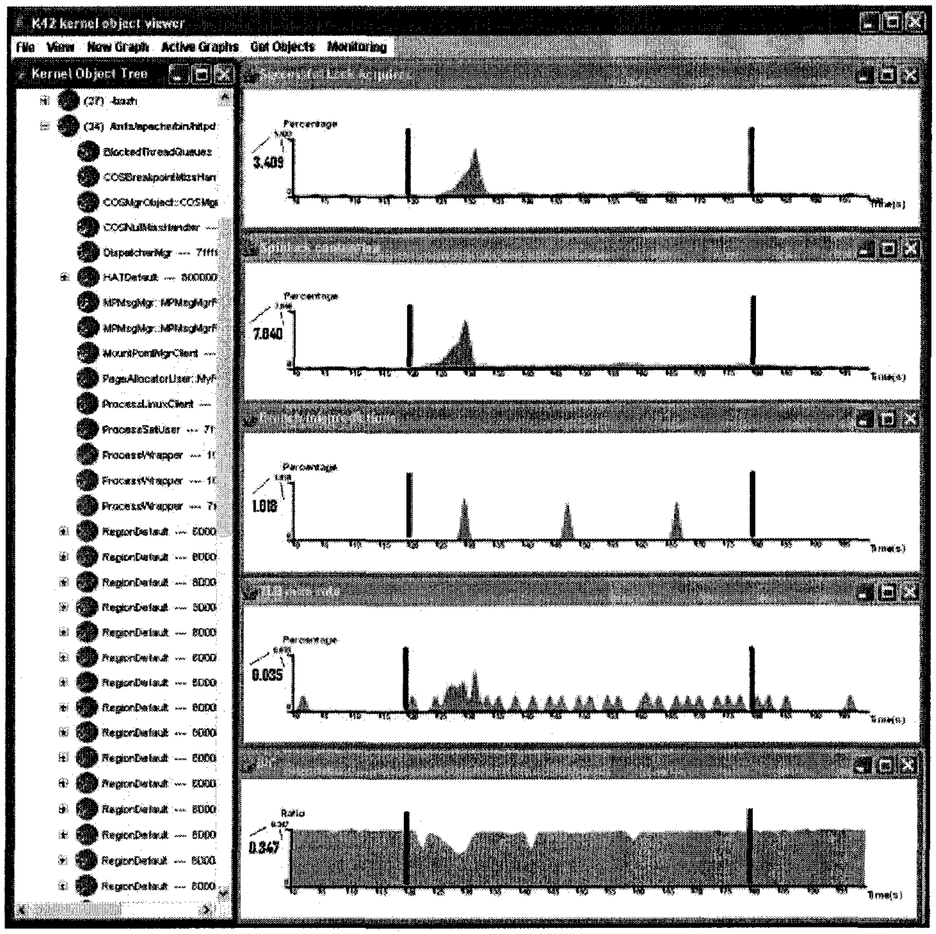


Figure 4.5: A snapshot of the complete first run of SURGE on the Apache webserver measured using five metrics from time index 20s to 80s (indicated by the vertical black lines). From top to bottom, the windows show the amount of measured lock acquisition rate, lock contention, branch misprediction rate, TLB miss rate, and IPC respectively. SURGE was configured to generate a 35 requests per client, use two clients, and execute for a total of 60 seconds.

the rise in peak lock contention for Apache loaded with SURGE and ApacheBench. Figure 4.5 also indicates that Apache loaded with SURGE experiences a peak in lock contention during the initial seconds of the run. From analysis of Apache loaded with ApacheBench, we learned that a workload's initialization phase can create high lock contention, corresponding to the child process creation phase in Apache. However, Figure 4.4 does not indicate that values for peak lock contention continue increasing for higher concurrency with SURGE, as they did with ApacheBench. For a possible cause of this difference, we will examine performance results gathered using other metrics.

Let us consider the behaviour of TLB misses with SURGE and ApacheBench. With ApacheBench, the TLB miss rate is constantly rising; however, with SURGE this is not the case. With SURGE, the TLB miss rate goes up until about 15 requests per client and does not increase much further for a greater number of requests. With SURGE, 15 requests per client corresponds to a request concurrency setting of 30 for ApacheBench (SURGE uses two clients).

The analysis for ApacheBench concluded that the memory subsystem was responsible for the increased level of lock contention. The TLB miss rate is directly proportional to the rate at which new pages are being fetched by the system. Consequently, the fact that the TLB miss rate does not increase for SURGE with a corresponding increase in the number of requests per client, indicates that the rate at which new pages are fetched by Apache during a SURGE workload does not increase either.

We validated our hypothesis by examining the documentation on the SURGE workload. Contrary to ApacheBench, a SURGE generated workload does not immediately open all of its connections. Rather the number of connections steadily increases until finally reaching the maximum value, specified by the number of clients and requests per client. This period of maximum load is the period of the so called 'surge'. The effect this steady increase in the number of requests has on the system is that only several new child processes are created by Apache at any given time, rather than all at the same time, which is the case when running ApacheBench. Consequently, under SURGE there is a lower demand for new memory pages at any given time. A lower demand for new memory pages manifests itself in the form of a lower TLB miss rate.

We have shown how KOV can be used to identify a performance bottleneck within the operating system using ApacheBench, and how KOV can help the user understand how a program's behaviour can manifest itself on the system using SURGE. The next section shows a similar analysis using KOV for SPECjbb2000.

4.2.3 SPECjbb2000

The performance of SPECjbb2000 was monitored during a complete run of the benchmark which consists of a 30 second warm-up phase and a 120 second measurement phase. Figure 4.6 and Figure 4.7 show the results obtained for SPECjbb2000 for the aforementioned five metrics. Figure 4.6 shows the first 82 seconds of execution. The 30 second warm-up phase can be identified on the figure as the portion of execution between the two black lines (time indices 18s to 48s), whereas the measurement phase starts after the second black line (time index 48s) in Figure 4.6 and continues up until its conclusion, which is shown in Figure 4.7 (at time index 70s).

We will proceed by analyzing each phase of execution in detail using to the results gathered for our five metrics. We will then compare these results against the ones obtained for Apache workloads.

Looking at the warm-up phase in Figure 4.6 (the region between the two vertical black lines) we can see that the results for three metrics vary substantially from the average: spin lock contention, TLB miss rate, and IPC. From the analysis of Apache we have learned that high lock contention can be caused by the memory subsystem, but also that memory pressure is usually accompanied by a high rate of TLB misses. Although Figure 4.6 shows high lock contention, the TLB miss rate during that period is insignificant. Consequently, by looking at the results gathered with KOV, we can conclude that the memory subsystem is not a bottleneck in the warm-up phase because the TLB miss rate is not correlated with lock contention, as was the case with ApacheBench.

Figure 4.6 shows that there is a significant increase in IPC (up to 0.545 from an average of roughly 0.400) during the period of low TLB miss rate and high lock contention. Two reasons for a high IPC rate could be the execution of highly optimized code, or the execution of very few memory operations. Code optimizations such as prefetching, loop unrolling, or branch hinting, reduce the number of processor stalls, and thus increase IPC. Code which contains few memory operations has a lower chance of accessing data not present in the cache, or TLB, and therefore causes fewer processor stalls, which translates into increased IPC.

We consulted the documentation of SPECjbb2000 for reasons that could explain the higher IPC, higher lock contention, and lower TLB rates in the warm-up phase. The documentation indicates that the warm-up phase for SPECjbb2000 is composed of two tasks. The first task creates data structures, whereas the second task executes several threads which populate the aforementioned data structures. We will proceed by analyzing each of these tasks in greater detail.

The first task of the warm-up phase creates binary trees for each warehouse (please refer to description of SPECjbb2000 in Section 4.1.2) which are accessed during the measurement phase. This task is computational in nature and therefore puts more stress on the CPU rather than the memory subsystem. Thus, one possible reason for a low TLB miss rate and increased IPC is that this task exhibits good cache locality, which would manifest itself by exhibiting fewer TLB misses and fewer processor stalls. A second possible reason is the presence of the JIT compiler in the JVM. It is possible that the highly computational task was optimized by the JIT compiler, in which case extra prefetching instructions, loop unrolling, or branch hints might have been inserted to increase performance. Finally, the higher incidence of lock contention during the warm-up phase is likely the result of concurrent accesses to the binary trees created at this time,

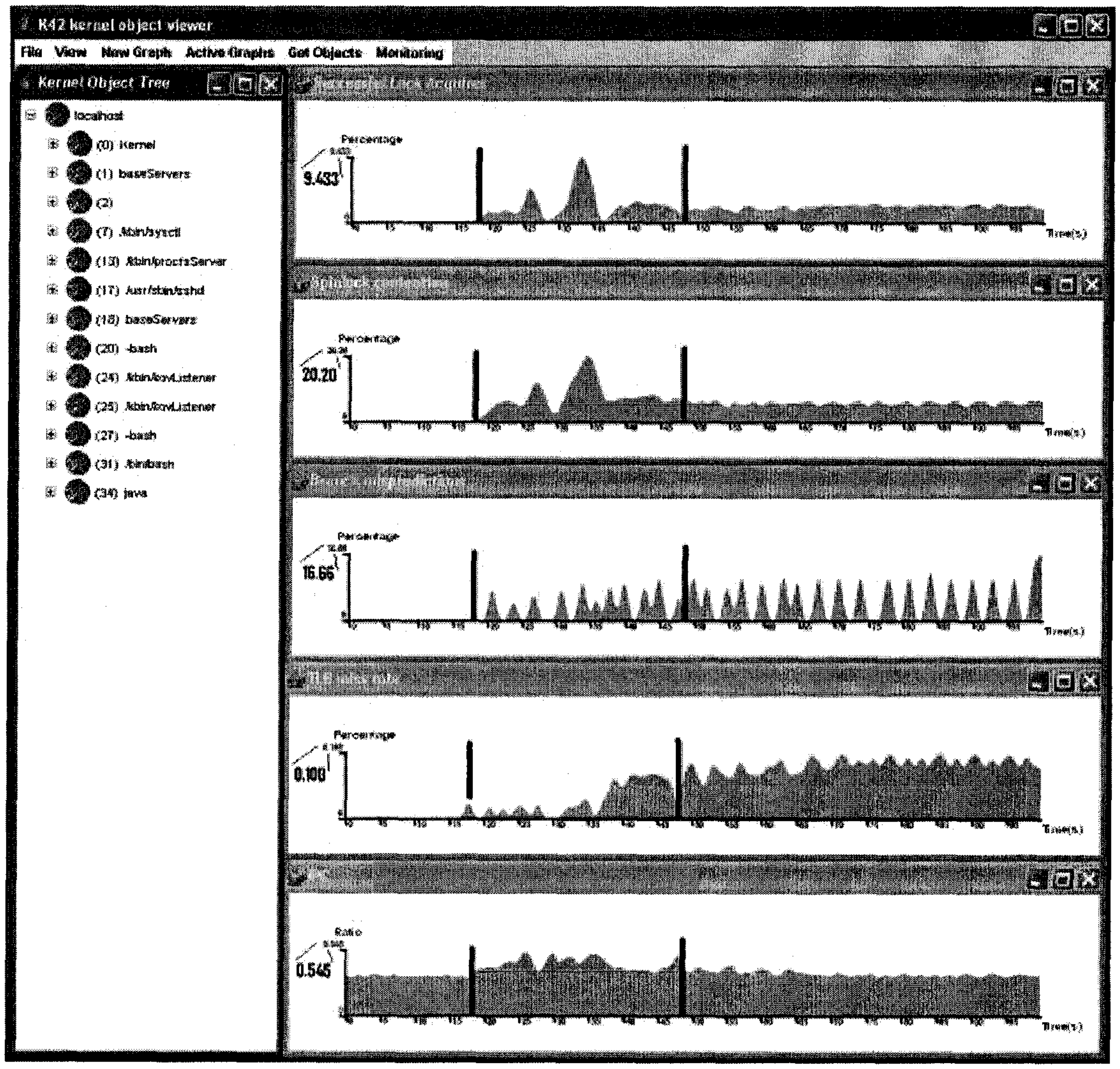


Figure 4.6: A snapshot of the first 82 seconds of execution of SPECjbb2000. The results for the 30 second warm-up phase of the SPECjbb2000 run are between time index 18s and 48s (indicated by the vertical black lines). From top to bottom, the windows show the successful lock acquisition rate, spin lock contention, branch misprediction rate, TLB miss rate, and IPC, respectively. A complete run of SPECjbb contains a 30 second warm-up phase, and a 120 second measurement phase. The first 52 seconds of the measurement phase shown start after time index 48s (right vertical black line).

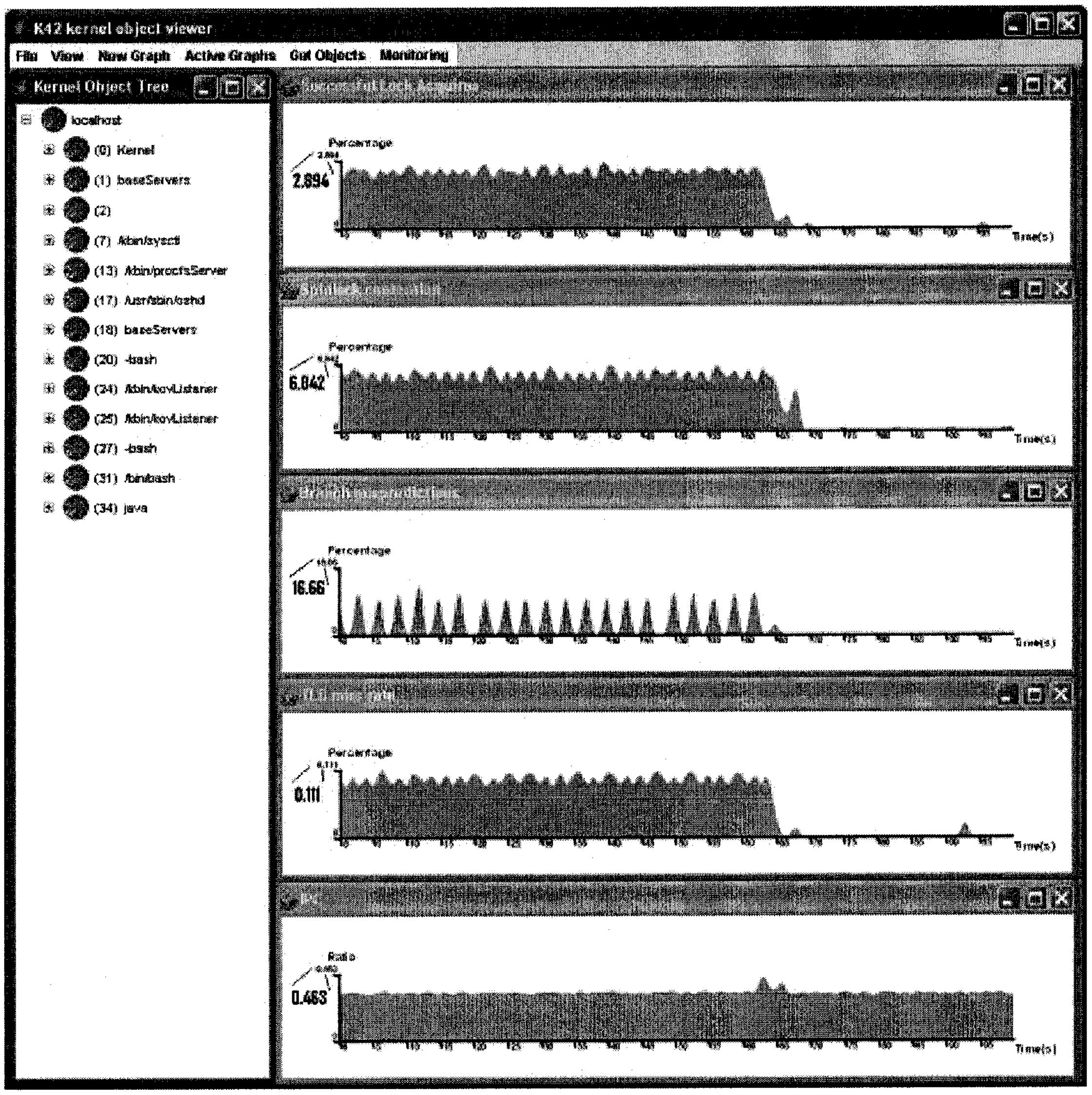


Figure 4.7: A snapshot of the last 65 seconds of the execution of SPECjbb2000. From top to bottom, the windows show the successful acquisition rate, spin lock contention, branch misprediction rate, TLB miss rate, and IPC, respectively. A complete run of SPECjbb contains a 30 second warm-up phase, and a 120 second measurement phase. This snapshot only shows the last 65 seconds of the measurement phase

although we don't have access to further information that could prove this hypothesis.

The second part of the warm-up phase executes several threads which populate the aforementioned binary trees. This task stresses memory, rather than the CPU, since a large amount of data is accessed within a short amount of time. Figure 4.6 and Figure 4.7 show that there is a correlation in the results between the second part of the warm-up phase and the 120 second long application measurement phase.

For the rest of this section, we will describe the measurement phase in greater detail based on the documentation of SPECjbb2000. We conclude by analyzing the results of the second part of the warm-up phase and the measurement phase together.

Documentation for SPECjbb2000 indicates that the benchmark simulates an order processing application for a wholesale company, where customers initiate a set of operations such as placing new orders or requesting the status of an existing order. By this definition, one can expect a mix of memory and CPU load. Performance results for the measurement phase, shown both in Figure 4.6 and Figure 4.7, indicate that the load on the memory subsystem is more significant than on the CPU. The higher rate of TLB misses, combined with a lower IPC rate support this claim. Since the second part of the warm-up phase consists of several threads, primarily issuing requests for data, the similarity in performance results between this portion of the benchmark and the measurement phase can be attributed to load being primarily put on the memory subsystem in both cases.

In conclusion, we have shown using KOV that there is a significant difference in the way in which Apache workloads, and the SPECjbb2000 benchmark affect the underlying system. In addition, we show KOV's utility by displaying results for a variety of metrics. The visualization of multiple metrics, rather than just one, such as TLB miss rate, or spin lock contention, was crucial in understanding the operating system and application performance behaviour.

We have described the results obtained with KOV by monitoring Apache and

Table 4.1: Overhead of monitoring per benchmark

Workload	PMU Overhead (%)	Soft. Instrum. (%)	Data Traffic and Scan(%)	Total(%)
ApacheBench	1.25	2.28	2.05	5.58
SURGE	1.03	2.11	2.06	5.20
SPECjbb2000	2.97	2.17	2.31	7.45
Average	1.75	2.19	2.14	6.08

SPECjbb2000 workloads. The next section will provide a detailed breakdown of overhead introduced by KOV.

4.3 Overhead Considerations

The results presented in Section 4.2 incorporate the execution of a program as well as the overhead introduced by our tool. The KOV performance monitoring tool incurs overhead from:

1. PMU interrupt handling,
2. the presence of software instrumentation, even if disabled, and
3. transfer of monitoring data and from periodically performing object scans.

Table 4.1 shows the overhead introduced by KOV when running the Apache webserver under ApacheBench and SURGE, and when running SPECjbb2000, distributed amongst the aforementioned three categories with all software instrumentation disabled.

Overhead for each workload was measured by comparing results with and without KOV. The overhead for ApacheBench was calculated by measuring the percentage increase in execution time. For SURGE, the overhead was calculated by measuring the percentage increase in transfer delay because the workload's execution time is fixed.

Similarly, since the runtime of a SPECjbb2000 workload is constant, the overhead for SPECjbb2000 was calculated by measuring the drop in throughput with KOV present. Each overhead measurement is the average overhead incurred over three consecutive runs of each workload calculated using arithmetic mean.

Overhead measurements presented in Table 4.1 indicate that the cost of performing an object scan is relatively constant, as is the cost of disabled software instrumentation. However, the PMU overhead varies slightly between different Apache workloads, and more significantly when compared against SPECjbb2000. The higher overhead generated by the PMU for SPECjbb2000 can be attributed to a higher rate of events for a majority of the five PMU-collected metrics measured in our evaluation.

We present a more detailed analysis of the overhead incurred by measuring each of the five metrics individually in Table 4.2. Because the overhead of monitoring each metric separately is very small, the ApacheBench workload was extended to run for 50,000 requests, instead of 5,000, and the SURGE workload was extended to run for 600 seconds, rather than 60. The SPECjbb2000 workload could not be extended to run for a longer period of time, however SPECjbb2000 measures its throughput very accurately (in tens of thousands of operations per second), and thus a noticeable difference was still present for different overhead measurements. Table 4.2 also shows the overhead of using the PMU to simultaneously measure all five metrics, similarly to the result shown in Table 4.1. This result is shown because it measures the overhead for the extended workloads.

Overhead generated by the PMU is the result of monitoring the following metrics:

1. successful lock acquisition rate,
2. spin lock contention,
3. branch misprediction rate,
4. TLB miss rate, and

Table 4.2: Overhead of using PMU to monitor five metrics for each workload

Workload	Successful lock acquisition rate (%)	Spin lock contention (%)	Branch mis- -prediction rate (%)	TLB miss rate (%)	IPC (%)	Simult. (%)
ApacheBench	0.35	0.45	0.29	0.35	0.19	1.20
SURGE	0.37	0.34	0.24	0.28	0.20	1.01
SPECjbb2000	0.68	0.86	0.28	1.20	0.24	2.97
Average	0.47	0.55	0.27	0.61	0.21	1.73

5. IPC.

Table 4.2 shows that on average, IPC, lock contention and TLB miss rates are higher for SPECjbb2000 than for any Apache workload. The higher rate of events generated by the PMU translates to more interrupts being handled by the system, which is directly proportional to overhead.

It should be noted that measuring each of the five metrics separately with the PMU incurs a total of 2.11 % overhead on average, whereas measuring all five metrics simultaneously incurs only 1.73 % overhead on average. The 0.38 % higher overhead for five metrics measured separately results from the need to count the number of retired instructions four times, when monitoring successful lock acquisition rate, spin lock contention, TLB miss rate, and IPC. When measuring all five metrics simultaneously, the number of retired instructions is only counted once, hence the lower combined overhead generated by the PMU.

In addition to a detailed breakdown of the overhead incurred by the PMU, we also show in Table 4.3 a breakdown of the overhead incurred by disabled software instrumentation and KOV. Overhead generated by the presence of disabled software instrumentation in our prototype consists of:

1. object invocation instrumentation, and

Table 4.3: Overhead of disabled software instrumentation per workload

Workload	Object Invocations (%)	Sleep locks (%)
ApacheBench	2.28	<0.01
SURGE	2.11	<0.01
SPECjbb2000	2.17	<0.01
Average	2.19	<0.01

2. sleep locks instrumentation.

The presence of software instrumentation within the system causes overhead, whether it is enabled or disabled. The results in Section 4.2 incorporate the overhead of disabled software instrumentation which can monitor object invocations and sleep locks when enabled. Overhead for each type of software instrumentation was measured by performing the aforementioned three consecutive runs on three different versions of the binary. The first binary contained no software instrumentation, the second binary only contained instrumentation to monitor object invocations, whereas the third binary only contained instrumentation to monitor sleep locks. The difference in performance runs for the first and second binary expose the overhead of object invocation instrumentation, and the difference in performance runs for the first and third binary expose the overhead of sleep locks instrumentation.

Table 4.3 shows the overhead incurred by each type of disabled software instrumentation when measured for Apache and SPECjbb2000 workloads. Recall that instrumentation to monitor object invocations modified the DREF macro, and instrumentation to monitor sleep locks modified the BlockedThreadQueues class (please refer to Section 3.2.1). The overhead measurements indicate that instrumentation designed to monitor sleep locks incurs negligible overhead (below 0.01 %). The reason for such a negligible contribution results from the implementation of sleep locks. The sleep locks instrumentation is only executed when a sleep lock is acquired, or released, not during normal execution

Table 4.4: Overhead of software instrumentation on normal execution

Code Instrumented	Overhead when disabled(%)	Overhead when enabled (%)
DREF macro	51.4	250.5
BlockedThreadQueues	2.3	10.5

in or outside the critical section. In addition, sleep locks instrumentation, when disabled, only executes a single *if-statement* on top of the original sequence of instructions to determine whether logging should be performed. Contrary to this, instrumentation designed to monitor object invocations is used frequently, and even when disabled executes at least one function call and *if-statement.* The function call is necessary to access the enable flag for this instrumentation, because the flag is not present in the object using the DREF macro. A more detailed breakdown of the overhead of software instrumentation on normal execution of the DREF macro and BlockedThreadQueues class is shown in Table 4.4.

The overhead shown in Table 4.4 shows the overhead of software instrumentation on execution time of the DREF macro and the BlockedThreadQueues class when ran repeatedly inside a loop. Each loop was iterated 1 million times to generate enough variation in execution time between the instrumented and uninstrumented runs. The significantly higher cost of software instrumentation in the DREF macro contributes to the higher overhead for object invocation instrumentation shown in Table 4.3. The notably higher cost of enabled instrumentation within the DREF macro than BlockedThreadQueues class is due to the simplicity of the DREF macro's original implementation. By default the DREF macro performs only one function call. When DREF instrumentation is enabled, there is an additional function call, for a total of two, being executed to save the object pointer, which constitute a significantly higher percentage of the original execution time than for the BlockedThreadQueues class (which originally performs a hash and calls several functions within each instrumented method).

Table 4.5: Overhead of performing actions and data traffic within KOV

Workload	Processes and Objects (%)		Object State (%)		PMU	
	Action	Data Traffic	Action	Data Traffic	Action	Data Traffic
ApacheBench	0.10	0.16	0.74	1.13	1.20	0.38
SURGE	0.10	0.13	0.77	1.22	1.01	0.33
SPECjbb2000	0.15	0.20	0.77	1.19	2.97	0.55
Average	0.12	0.16	0.76	1.18	1.73	0.42

Finally, KOV incurs overhead by passing data and performing periodic object scans of the system. Data passed include results gathered from the PMU, and the following two tasks performed during an object scan:

1. obtaining a list of live processes and extracting objects from the GTT, and
2. obtaining state information from all the objects.

Table 4.5 shows overhead for performing each task required for the object scan and using the PMU, separated into the overhead incurred by gathering data by the KM (Action column), and the transfer of data from the KM to the GUI (Data Traffic column). The values for the PMU in the Action column are the same as in Table 4.2 (Simult. column), but have been added to this table for completeness. The period with which the object scan was repeated is five seconds. These results for each overhead measurement were obtained by comparing the performance of extended workloads, similarly to the extended workloads used to measure the overhead of each PMU metric.

The overhead of performing each action separately, and moving the generated data, was measured by using different binaries. A different version of the binary was used to specify whether data should, or should not be passed back to the GUI from each of the actions shown in Table 4.5. Overhead shown in the 'Action' column of the PMU is identical to the overhead measurement for the PMU from Table 4.2.

4.3.1 Sampling Frequency

Many of the results presented in this Chapter were obtained by using the PMU. A critical parameter when configuring the PMU is the counter threshold value which determines the sampling frequency used. Sampling frequency affects the accuracy of statistical sampling, and it affects the overhead associated with using the PMU. In this section, we present an exploration of various PMU sampling frequencies, and study their affects on the accuracy of our results and the overhead associated with performance monitoring. Before we discuss the results, a short overview of the trade-off of sampling frequency and measurement overhead is given.

In essence, the trade-off between sampling frequency and measurement overhead can be summarized into one point. A lower sampling frequency generates results with lower accuracy, but also introduces less overhead. Conversely, a higher sampling frequency generates results with greater accuracy, but also introduces more overhead.

The link between sampling frequency and measurement accuracy can be explained by referring to the Nyquist frequency (please refer to Section 2.1.2). Effectively, as the sampling frequency of the PMU decreases, so does the range of frequencies of events the PMU is capable of detecting.

Ideally, one would configure the PMU to the maximum sampling frequency and therefore capture all available events; however, a higher sampling frequency translates to a greater number of PMU interrupts. Each PMU interrupt not only stops the execution of the currently running program, but also executes the instructions of an interrupt service routine. These actions delay the execution of the main program, and thus lengthen its total execution time. Consequently, a greater incidence of interrupts means a longer execution time for the main program, and thus greater measurement overhead.

Figure 4.8 shows a per-object breakdown of lock contention when running ApacheBench with 5000 total requests and a concurrency level of 50. The three windows show the results obtained using different PMU counter thresholds. The three window show, from

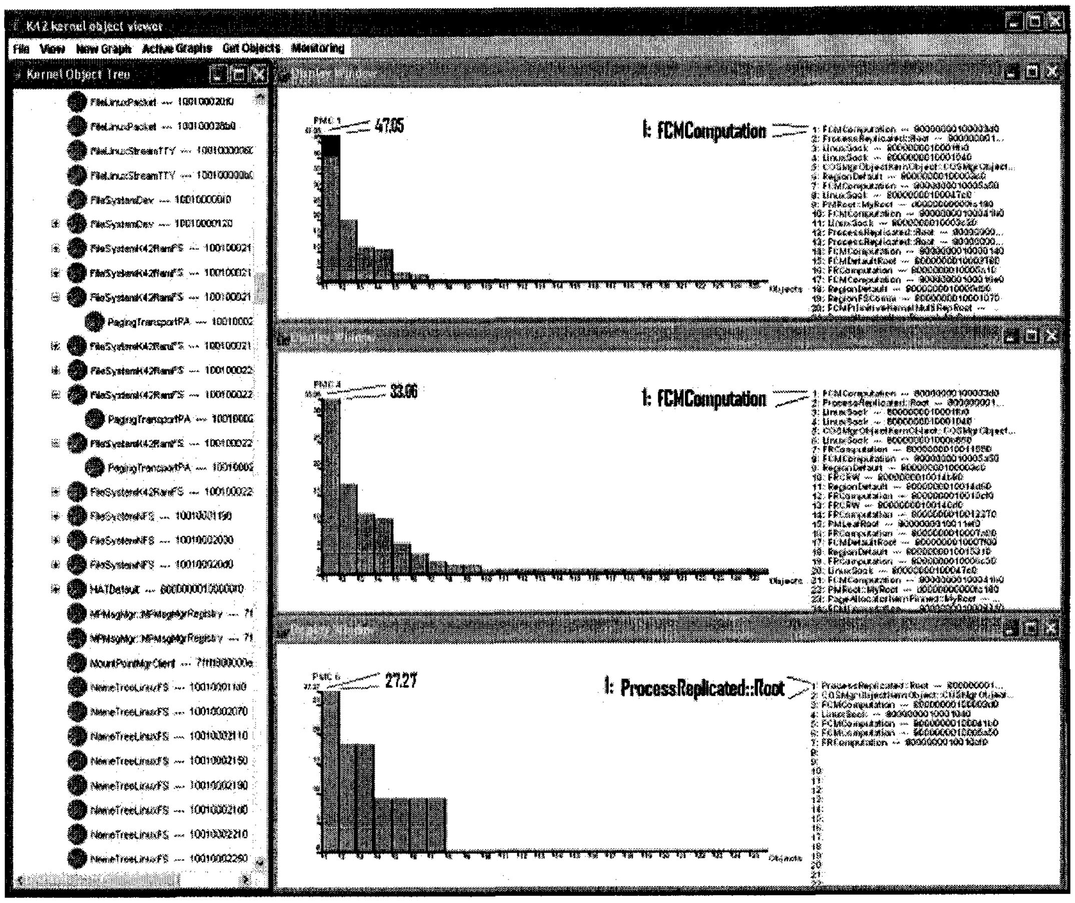


Figure 4.8: Lock contention on a per-object basis using different PMU counter thresholds when measured for the first run of ApacheBench on the Apache webserver. From top to bottom: 10k, 100k, and 1M per one sample. ApacheBench was configured to generate a request concurrency of 50 and a total of 5000 requests.

top to bottom, results obtained with the highest sampling frequency (lowest threshold), medium sampling frequency (medium threshold), and low sampling frequency (highest threshold). From the figure we can see that the highest lock contention is attributed to the same set of objects for the high and medium sampling frequencies. For example, in the former case FCMComputation is attributed with 47.05 % of total contention, whereas in the latter, FCMComputation is attributed with only 33.06 % of total contention. The relationship between the objects attributed with the highest lock contention also remains the same for the high and medium sampling frequencies. For example, if one were to choose the objects with the highest lock contention to optimize, the choice across both sampling frequencies remains the same. More specifically, FCMComputation, ProcessReplicated::Root, and two LinuxSockets (differentiated by their object ID) are the highest contributors to lock contention in both cases.

Under the smallest sampling frequency (bottom display window of Figure 4.8) the data is no longer representative of the system. Some of the objects with high lock contention detected using higher sampling frequencies are not shown (e.g. LinuxSock), and the relationship between the objects is not similar to that obtained using a higher sampling frequency. Consequently, using a low sampling frequency does not provide representative information on the system.

Since performance overhead is directly proportional to the interrupt overhead generated by individual HPCs, it is intuitive then that the smallest sampling frequency that still provides meaningful information on the system should be chosen for monitoring. As such, a sampling threshold of 100,000 was chosen to monitor spin lock contention.

A similar study was done for other PMU events, and the counter thresholds we decided to use, including the spin lock contention threshold, were set as follows:

1. successful lock acquisition rate - 100,000

2. spin lock contention - 100,000

3. branch misprediction rate - 10,000,000 for predicted and mispredicted branches
4. TLB miss rate - 100,000 for instruction and data TLB misses
5. IPC - 10,000,000 for instructions and 100,000,000 for cycles

We have shown the overhead KOV induces on the system during monitoring and the decision process we used to pick our PMU configuration parameters. We will conclude this chapter by discussing the micro-benchmarks we used to validate the correctness of our measurement results.

4.4 Correctness of Measurements

The validity of the results gathered using KOV's performance monitoring capabilities was tested using hand-written micro-benchmarks. Since monitoring information on the system is gathered by using the PMU and software instrumentation, we will discuss the micro-benchmarks used to validate each of these separately.

4.4.1 PMU

The validity of individual performance configurations of the PMU was tested using hand written micro-benchmarks for each of the five metrics used in the previous sections, which include:

1. successful lock acquisition rate,
2. spin lock contention,
3. branch misprediction rate,
4. TLB miss rate, and
5. IPC.

The lock acquisition rate is measured by counting the number of successfully executed store-conditional instructions. To test correct counting we inserted an assembly block with a loop containing all the instructions that are necessary to implement a spin lock, including the load-linked, test-and-set, and store-conditional instructions. The conditional branch that usually follows the above sequence and would execute the locking instructions indefinitely was replaced with a conditional branch based on a loop index.

Two types of tests were performed using the above code. The first test involved setting the test-and-set condition such that it is always false, and thus the store-conditional never executes. The second test involved setting the test-and-set condition such that it is always true so that the store-conditional always executes. No other threads were ran in parallel when testing to ensure that the store-conditional is not affected by other store instructions being executed in parallel By varying the loop limit as well as the sampling frequency, the number of obtained data samples was verified against the expected number of samples. In addition, since the interrupt routine extracts the local object pointer, the micro-benchmark used object-oriented programming. Subsequently, the interrupt extracted object pointer was compared against the benchmark's object pointer for correctness. The micro-benchmark used to validate lock acquisition rate measurements was very similar to the one used to validate lock contention measurements.

Lock contention is measured by counting the number of executed load-linked instructions. Similarly to measuring the lock acquisition rate, an assembly block was used to insert locking instructions within an indexed loop. Because the load-linked instruction always executes, whereas the store-conditional does not, only one test was necessary to validate the measurements. Thus, by varying the loop limit as well as the sampling frequency, the number of obtained data samples was verified against the expected number of samples.

The branch misprediction rate is measured by counting the number of predicted and mispredicted branches. We used manually inserted branch prediction hints to validate our measurements. Our micro-benchmark contains an indexed loop, where we hinted the backwards branch in the looping structure as not taken. Since a backwards branch is not taken only when exiting the loop, and the typical branch predictor will predict a backwards branch as taken, we can vary the number of mispredicted branches by setting the loop limit. Predicted branches were counted by setting the branch hint to indicate that the backwards branch is always taken, and similarly varying the number of predicted branches by setting the loop limit.

The TLB miss rate is measured by counting the amount of TLB misses caused by fetching instructions and reading and storing data. The PowerPC 970FX, the processor used to evaluate our tool, has a 1024-entry, 4-way set associative TLB, with a Least-Recently Used (LRU) page replacement policy and uses 4KB memory pages [23]. The easiest way to simplify the behaviour of the TLB is to generate a series of sequential accesses that has no page reuse, because then the page replacement policy and set associativity do not matter. To achieve this, we designed a micro-benchmark which performs a series of 4KB memory allocations, and subsequently performs a read and write operation to that region of memory. The above operations are contained within an indexed loop. Therefore, each iteration of the loop will cause one TLB miss. By varying the loop limit as well as the sampling frequency, the number of obtained data samples was verified against the expected number of samples.

IPC is measured by counting the number of completed instructions and completed processor cycles. We validated our measurements for the number of processor cycles without the aid of a micro-benchmark. Given that the clock speed of the PowerPC 970FX is equal to 2.3 GHz [23], one second of measurement of the system was verified against the expected number of samples which should be equivalent to 2.3 Giga cycles. Before we elaborate on the micro-benchmark we used to validate our measurements of completed instructions, we will give a brief explanation of PowerPC 970FX processor's architecture.

The PowerPC 970FX processor contains ten execution pipelines, and can maintain a maximum of 215 instructions in various stages of execution across all of its pipelines at any given time [23]. Logic within the processor will attempt to extract maximum instruction-level parallelism (ILP) by considering data dependencies, branch prediction, etc., with the goal of filling all ten pipelines with instructions. Since the goal of our micro-benchmark is to obtain an accurate count of completed instructions, the micro-benchmark must contain no ILP that the processor can detect, so that the processor will retire one instruction per one processor cycle (IPC of one). To ensure an IPC of one, the micro-benchmark must also cause no processor stalls, which can be caused by cache misses, or branch mispredictions.

Based on the above analysis, we constructed a micro-benchmark which consists of arithmetic operations contained within an indexed loop. The result of each individual arithmetic operation is used as input into the following arithmetic operation. Therefore there are direct data dependencies between all the arithmetic instructions between consecutive loop iterations.

The arithmetic-logic unit (ALU) can forward the result of one ALU operation to the input of another ALU operation without using the cache at all (called forwarding). By strictly using arithmetic operations (adds) no stalls will result from memory accesses. In addition, since all the instructions are within a loop, backwards branches are predicted as taken and therefore no stalls will be introduced by branch mispredictions. We used this micro-benchmark to validate our measurements, and monitored the system to obtain an IPC of one.

We have discussed the set of micro-benchmarks we used to validate our performance results gathered using the PMU. The next section will describe the micro-benchmarks we used to validate our software instrumentation.

4.4.2 Software Instrumentation

The performance monitoring capabilities of KOV include monitoring sleep locks, and counting object invocations. Instrumentation of sleep locks counts the number of threads that add and remove themselves from a queue, as well as the thread identifications numbers (IDs) which perform these operations. We verified the correctness of our sleep lock instrumentation by first instantiating a single BlockedThreadQueues object (please refer to Section 3.2.1) within our micro-benchmark and then creating a number of threads which add and remove themselves from a queue using the BlockedThreadQueues object. We verified the correctness of our sleep locks instrumentation while varying the order in which the threads add and remove themselves from the queue, and the total number of threads and their IDs.

Instrumentation of object invocations counts the number of invocations by logging all clustered object identification numbers passed to the DREF macro (please refer to Chapter 3.2.1). Correctness was verified by creating several clustered objects and invoking them within an indexed loop. The identification number of each clustered object, which is known to the micro-benchmark, was verified against the log obtained from DREF during the run of the micro-benchmark. By varying the amount of clustered objects used, the loop size, and the order in which the objects use the DREF macro, we verified the correctness of our instrumentation.

Chapter 5

Concluding Remarks

We presented the design and implementation of the Kernel Object Viewer (KOV), an object-oriented monitoring system which:

1. combines hardware-level and software-level monitoring capabilities,
2. simultaneously monitors user-level and kernel-level software components,
3. can attribute performance bottlenecks to specific object-level code segments, and
4. provides sophisticated visualization support to aid in system performance debugging.

Our design goals were to create a tool which could aid programmers and operating systems designers in performance analysis and identifying bottlenecks. Towards this end, we have shown how KOV can be used to analyze the performance of different applications on the K42 operating system. We demonstrated how KOV can be used to accurately identify a performance bottleneck and subsequently identify the cause of that bottleneck. We also showed how KOV can be used to ease qualitative and quantitative analysis of a running application and the underlying system substrate.

In this dissertation we make three primary contributions. First, we describe a system that dynamically tracks important performance metrics using Hardware Performance

Counters:

1. at object instance-level granularity,
2. requiring no changes to the code,
3. adding no overhead when monitoring is not required, and
4. allowing monitoring overhead to be varied by dynamically changing the sampling frequency.

The second contribution is a comprehensive, system-wide scanning facility which extracts all live processes and their objects. The third contribution is a novel mechanism to track lock acquisitions and contention in a way that requires no changes to the code.

During the course of development of KOV we encountered many obstacles and challenges. Several of them were conceptual, whereas others were implementation details. In the following section we discuss the lessons we learned by developing KOV.

5.1 Lessons Learned

Throughout KOV's development we encountered three main problems. The first problem concerned visualization. Essentially, the list of live processes and their objects, composed of potentially thousands of objects had to be represented in a concise and legible fashion. The second problem was to design a method which could extract all live processes and their objects in a general way. Since there can be hundreds of Clustered Object classes, the technique we used had to be able to scan them all without knowing what each object class implements explicitly. The third problem concerned the technique with which we would obtain the performance monitoring data. The technique had to be able to attribute performance measurements to object instances (since K42 is object oriented), and secondly it had to be dynamic to incur no overhead when not in use such that it could be usable in real systems.

To solve the visualization problem, we decided that the tree approach would be a suitable method of visualization because it introduces levels of abstraction by design. For example, the root of the tree is the machine name, the first level shows all the processes running on that machine, the second level shows high level objects like the memory manager, memory regions, and so on, where eventually the leafs show file representatives, and individual memory segments.

To extract the list of live processes and their objects, we considered several approaches. Initial approaches to extract this information revolved around the Process object. The Process object keeps track of the memory manager, memory regions, and the hardware address translator (HAT) for a particular process. As such it appeared a suitable choice because once the memory manager, memory regions, and HAT were obtained, they themselves would reference further objects, such as file representatives, and more. However, quite quickly we realized that some objects in a process' address space cannot be reached by looking at the Process object alone. For example, references to globally accessible kernel objects are stored in static variables of the Clustered Object Manager, and thus other classes do not need to store them explicitly. At that point, we considered including the Clustered Object Manager in our scan along side the Process object, but very soon the list of objects we'd have to scan explicitly kept growing. In addition, we also would have had to include file system nodes, network sockets, and many more objects. Eventually, we decided that this approach was not only tedious, but also error prone. We could not guarantee that all objects were included in the scan without performing exhaustive analysis. Even if the analysis was made, and all the relevant objects were found, once K42 was modified in any way and new Clustered Objects were added, the scan results would become invalid.

To overcome this obstacle, we studied K42's implementation in great detail. Eventually, we discovered that the Global Translation Table (GTT) (please refer to Section 2.4.2) contains a reference to the Root object of every Clustered Object. Using the Root object, subsequent Representative objects could be found by consulting Local Translation Tables on each processor. This approach essentially solved all of the problems we had with the previous technique because reading the GTT provides the scan with all Clustered Objects. By K42's design, all new Clustered Objects have to register with the GTT.

The last major problem we encountered required KOV to acquire performance monitoring data dynamically at object granularity. Towards this end we utilized Hardware Performance Counters (HPCs), which were used extensively and expanded upon by Azimi et al to include multiplexing support that overcame previous hardware limitations associated with using HPCs. Our further work with HPCs lead to the development of our own interrupt service routine which extracted the object pointer needed to trace back performance measurements to object instances. By taking advantage of C++ convention, we designed a stack walk routine to read the context object pointer from the interrupted program's stack.

Inadvertently, as a direct consequence of our implementation, we realized that our performance measurement technique could be extended to measure spin lock contention dynamically. Since the implementation of atomic memory operations is composed of a unique sequence of instructions, such as the load-linked and store-conditional instructions, the next goal became to configure the Performance Monitoring Unit (PMU) properly. Further in-depth study of the PMU and its architecture resulted in the measurement technique we described in Section 3.2.1.

5.2 Future Work

The system presented in this dissertation can track performance at object instance-level granularity and graphically display the information that was gathered. However, ultimately we envision a tool capable of utilizing the performance data gathered to automatically optimize the system and relieve performance bottlenecks. As previously mentioned, K42 allows for a distributed implementation of its services as part of the Clustered Objects design paradigm. Distributed implementations can offer better scalability, but they also typically suffer greater overheads when scalability is not required. Distributed implementations also tend to optimize certain operations, improving their scalability, while increasing costs of other operations. In order to provide a means for coping with the tradeoffs of using distributed implementations, K42 enables hot-swapping, a technique for dynamically replacing a live Clustered Object instance with a different, but compatible, instance [8]. This mechanism can be used to switch between shared and distributed implementations and additionally enable other forms of dynamic adaptation. Future work on KOV will focus on utilizing K42's hot-swapping capabilities aimed at optimizing performance on a per-object basis.

Bibliography

[1] *ApacheBench.* http://www.apache.org.

[2] *SPECjbb 2000.* http://www.spec.org/jbb2000/.

[3] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'02)*, pages 1–16, November 2002.

[4] AMD. Athlon processor x86 code optimization guide. Technical report, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf.

[5] J. Appavoo. *Clustered Objects.* PhD thesis, University of Toronto, 2005.

[6] J. Appavoo, M. Auslander, D. Edelsohn, D. M. Da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a linux api on the scalable k42 kernel. In *Proceedings of the USENIX Annual Technical Conference (USENIX'03)*, pages 323–336, June 2003.

[7] J. Appavoo, M. Auslander, D. M. Da Silva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42's performance monitoring and tracing infrastructure. Technical report, IBM, http://www.research.ibm.com/K42/white-papers/PerfMon.pdf, August 2002.

[8] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, January 2003.

[9] J. Appavoo, K. Hui, M. Stumm, R. Wisniewski, D. M. Da Silva, O. Krieger, and C. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*, pages 3–8, November 2002.

[10] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization lite. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS'97)*, pages 43–48, May 1997.

[11] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*, pages 101–110, June 2005.

[12] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'98)*, pages 151–160, June 1998.

[13] A. Baumann, J. Appavoo, D. M. Da Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS'04)*, pages 21–27, October 2004.

[14] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'00)*, November 2000.

[15] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 271–282, October 2000.

[16] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX'04)*, pages 15–28, June 2004.

[17] IBM Corporation. The power4 processor introduction and tuning guide. Technical report, http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg247041.pdf.

[18] Intel Corporation. Intel itanium 2 reference manual for software development and optimization. Technical report, http://www.intel.com/design/itanium2/manuals/251110.htm.

[19] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architecture and Compilation Techniques (PACT'03)*, pages 220–231, December 2003.

[20] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson. Fine grained kernel logging with klogger: Experience and insights. In *Proceedings of the European Conference on Computer Systems (EuroSys'07)*, pages 259–272, March 2007.

[21] J. Fenlason. *GNU gprof*. SunOS 5.8 : Man pages, January 29 1993.

[22] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In

Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99), pages 87–100, February 1999.

[23] IBM. *PowerPC 970FX User Manual.* December 2005.

[24] Tavant Technologies Inc. *InfraRED: Opensource J2EE Performance Monitoring Tool.* http://infrared.sourceforge.net/versions/latest/, May 17 2006.

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, June 1997.

[26] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. M. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of the European Conference on Computer Systems (EuroSys'06)*, pages 133–145, April 2006.

[27] LivePerf. *Application Performance Monitoring Tool.* http://www.sharewareconnection.com/liveperf.htm, 2008.

[28] Microsoft. *Microsoft Windows XP: System Monitor overview.* http://lb1.www.ms.akadns.net/resources/documentation/windows/xp/all/proddocs/en-us/sag_mpmonperf_01.mspx?mfr=true.

[29] J. C. Mogul. Emergent (mis)behavior vs. complex systems. In *Proceedings of the European Conference on Computer Systems (EuroSys'06)*, pages 293–304, April 2006.

[30] M. Olszewski, K. Mierle, A. Czajkowski, and A. Demke Brown. JIT instrumentation: A novel approach to dynamically instrument operating systems. In *Proceedings of the European Conference on Computer Systems (EuroSys'07)*, pages 3–16, March 2007.

[31] J. Osier. *Online Manual on GNU gprof.* http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html, 1993.

[32] P. Panchamukhi. Kernel debugging with kprobes: Insert printk's into the linux kernel on the fly. Technical report, IBM, http://www.ibm.com/developerworks/library/l-kprobes.html, August 19 2004.

[33] B. M. Posey. *Windows Server 2003 Performance Tuning.* http://www.windowsnetworking.com/articles_tutorials/Windows-Server-2003-Performance-Tuning.html, July 2005.

[34] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium*, pages 49–64, July 2005.

[35] Apache HTTP Server Project. *Apache HTTP Server Version 1.3 Documentation.* http://httpd.apache.org/docs/1.3/.

[36] J. Reinders. *VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers.* Intel Press, 1st edition, 2005.

[37] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The paragon performance monitoring environment. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'93)*, pages 850–859, November 1993.

[38] D. M. Da Silva, O. Krieger, R. W. Wisniewski, A. Waterland, D. Tam, and A. Baumann. K42: an infrastructure for operating system research. In *Proceedings of the ACM SIGOPS Operating Systems Review*, pages 34–42, April 2006.

[39] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. M. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference (USENIX'03)*, pages 141–154, July 2003.

[40] B. Sprunt. Pentium 4 performance monitoring features. *IEEE Micro Journal*, 22(4):72–82, July/Aug 2002.

[41] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI'94)*, pages 196–2005, June 1994.

[42] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In *Proceedings of the Conference on Virtual Machine Research and Technology Symposium (VM'04)*, pages 57–72, May 2004.

[43] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI'99)*, pages 117–130, February 1999.

[44] Microsoft TechNet. *Windows 2000: Overview of Performance Monitoring.* http://www.microsoft.com/technet/prodtechnol/Windows2000Pro/reskit/part6/proch27.mspx?mfr=true.

[45] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'03)*, pages 15–21, April 2003.

[46] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the USENIX Annual Technical Conference (USENIX'00)*, pages 13–26, June 2000.