ARUM: Application Resource Usage Monitor

Rashawn L. Knapp[1], Douglas M. Pase[2], and Karen L. Karavanic[1]

[1]Portland State University
Department of Computer Science
P.O. Box 751
Portland, OR 97207-0751
{knappr, karavan}@cs.pdx.edu

[2]IBM System x HPC Portfolio Development
P.O. Box 12195, Dept. E1WA/205/EE154
3039 Cornwallis Rd.
Research Triangle Park, NC 27709-2195
pase@us.ibm.com

Abstract. We present the design and initial implementation of ARUM, an Application Resource Usage Monitor for multi-core and multi-processor AMD and Intel systems running the Linux operating system. ARUM is a lightweight, easy-to-use tool that operates on unmodified binaries, requires no kernel modifications or special user privileges to support access to hardware counters, and is designed to measure both system level and application level metrics. The design contains four measurement aspects: process and thread level resource usage, architecture specific event counting, application level measurements, and measurements of the ambient environment. We describe the design of ARUM and its related goals and design requirements. We have implemented the first two measurement components. In this paper we present the implementation of these components and early results of using ARUM.

1 Introduction

Obtaining performance data for systems and applications is difficult and time consuming, but it is necessary to understand the underlying causes of poor performance of applications, benchmarks, and new systems; to accurately model future systems and predict expected performance of applications on future systems; and to compare the performance of applications and benchmarks across architectures or under various system configurations. Much of the difficulty of performance analysis arises from the complexity of the tools — building and modifying a tool so that it runs on a specific system, modifying applications with instrumentation imperatives or library calls (which changes the original source, requires recompilation, and creates an additional binary), and managing multiple source versions, binaries, and associated performance data. The complexity of the tools prolongs any performance study, but the process is also time consuming because of the need to use several tools over multiple executions of an application to obtain various kinds of performance data. Additional time is required to then compile, aggregate, and analyze the data collected from the various tools over the multiple executions.

Performance studies often require the use of multiple tools because it is difficult to collect a variety of performance data with only a single tool. Most tools collect specific kinds of data. For example, Unix and Linux utilities focus on system level

metrics, but provide limited detail about the performance behavior of an application that executes on the system; and tools that collect detailed data about application performance offer little insight into the context of the application's runtime environment. In practice, performance analysis requires a combination of system and application level data, for example, to determine how to improve the performance of an application, to discover the root causes of a performance bottleneck, or to design an architecture for a particular class of applications. In work that targets performance analysis for large scale clusters, we address the importance of combined application and system performance analysis [9]. The published literature describes instances of communications libraries, the Operating System, and hardware preventing well-developed applications from achieving good performance [6, 7, 25, 26]. In these examples, the primary cause of degraded application performance was system interference, but analysts only discovered this after using a combination of tools; furthermore, application level analysis, which did not include system information, led analysts to make modifications to application and library code, optimizations which yielded little improvement in performance.

Motivation for this project stems from the environment of high end Linux servers, where analysts and system designers need easy-to-use, lightweight performance tools which capture a description of an application's performance within the context of the runtime system. Such tools will enable better understanding of application and system performance, better construction of models of future systems; and better comparison of application performance across architectures or different system configurations.

We have designed a tool called Application Resource Usage Monitor (ARUM). To minimize complexity, use of ARUM does not require application relinking, source modifications, or recompiling. Additionally, the application and the desired measurements are specified to ARUM using a simple command line interface. To significantly decrease the time required to conduct a single performance study, ARUM is designed to incur minimal overhead, and it is designed to collect four types of performance data: process and thread level resource usage, architecture specific event counting, application level measurements, and measurements of the ambient environment. By combining four types of measurement, ARUM describes a whole system, allowing analysts to obtain a description of the application within the context of its runtime environment. This combination of measurements reduces the number of tools an analyst may need to use for a single performance study.

We have designed ARUM for scenarios where having measurement information about the whole system, including an application's execution, would be more beneficial than having information only about a single sub-component. ARUM is useful in situations where application behavior changes unexpectedly, and the causes of the behavior change are unknown. In cases where potential causes may be suspected, ARUM can be used to validate these hypotheses. ARUM is also useful for benchmarking current systems and for creating performance models for future systems.

ARUM's initial implementation operates on Linux server class machines. In this context, an application that ARUM monitors may be highly threaded and the machine may contain many processors or cores. In the current state of development, ARUM does not maintain a facility to coordinate the monitoring of a multi-node application, such as an MPI application running on a cluster. We have implemented the first two

of the four measurement facilities. In the following sections we discuss related work, we describe the design of ARUM and some of its implementation details, we show an example of using ARUM on a dual core Opteron system, and we present a performance study of the costs and overhead of ARUM in its current development state.

2 Related Work

A number of performance tools exist for high end systems, but we do not know of a lightweight tool for Linux platforms that describes application performance within the context of its runtime system, while maintaining ease of use. ARUM creates an adequate description of application performance by collecting four kinds of performance data: hardware event data, resource usage of threads and processes, application level data, and data from the ambient environment (similar to data reported by utilities like *sar* or *vmstat*). ARUM is easy to use because it operates on unmodified application binaries, and it does not require patching and recompiling of the Linux kernel.

System utilities for Unix and Linux are easy to use and collect data about the ambient environment (interrupts, memory usage, processes), but they do not correlate these data to application performance. OProfile [12, 13] is a system-wide profiler distributed with Linux. It records the value of the program counter whenever a certain number of hardware events have occurred; and it aggregates the data into profiles for each binary image on the system. Users configure both the number of events and the type of events. OProfile is easy to use, and it provides a system-wide view of applications running on a system. However, it does not offer finer-grained measurement of applications, and it provides limited information about the ambient environment.

Most parallel performance tools focus on the collection of application level metrics, like function call counts and function entry and exit times. A few parallel tools collect a combination of application and system level data: TAU [28], KOJAK [20, 31], Open | SpeedShop [27], and PerfSuite [11] directly support hardware counter access, but each requires kernel patching and recompiling to support hardware counter access. Paradyn [17] indirectly supports access to system information through its metric definition language [4]. These tools do not readily support the collection of ambient environment metrics.

The Portable Interface to Hardware Performance Counters (PAPI) [2, 14, 24] and the Performance Counter Library (PCL) [1] are hardware counter interfaces that share the goal of simplifying the task of creating cross-platform portable software that access hardware counter data. Each provides a single interface to access hardware counter data; both support a wide range of architectures, including IBM, Cray, AMD, and Intel; and both support several operating systems, including AIX, Linux, Solaris, Windows, and IRIX. Additionally, they include a number of derived metrics, rates, and ratios which are compositions of the raw metrics. TAU includes support for PCL. However, PCL does not support newer architectures, as its most recent version is several years old. PAPI continues to add support for new architectures and operating

system versions; and is currently used by several parallel performance tools to collect hardware counter data (KOJAK, TAU, Open | SpeedShop, PerfSuite, HPCToolKit [16] and PapiEx [21]).

Both PAPI and PCL require a kernel patch and recompilation to operate on Linux. The kernel patching required to use these interfaces, as well as many other tools which access hardware counters on behalf of an application, ensures that the counts obtained relate only to events counted while the application's processes or threads held the CPU. In order to do this, the process control block must maintain knowledge of counted events. However, this requires modification to the structure of the process control block; hence a kernel patch is necessary. Since PCL does not support newer architectures, and since both PAPI and PCL require Linux kernel patching, neither was suitable for implementing ARUM's access to hardware counters.

ARUM is designed to collect performance data about the ambient environment. Some of this data can be collected from non-CPU hardware counters, like those available on many memory controllers, graphics cards, and network cards. A new pre-release version of PAPI, called Component PAPI [23] provides a way to access both the cpu counters and non-CPU counters an enhanced PAPI interface. The benefit of this approach is the ability to obtain performance data from a variety of hardware devices, which enables a better understanding of the runtime environment. However, it appears from review of the PAPI-C documentation that installation requires patching the Linux kernel.

There are several kernel level tools which are useful for understanding system performance, but provide limited direct support for application performance. Kerninst [8, 29] is a tool developed for dynamically instrumenting (inserting code into a running executable) and measuring Solaris and Linux kernels. One of the main features of Kerninst is that it operates on an unmodified kernel. Kprobes [15] is a similar technology developed for Linux, but it requires a kernel patch. Both Kerninst and Kprobes are useful for understanding kernel performance; but they do not provide direct support for the application level. KTAU [22] collects process-centric kernel level measurements and system wide kernel measurements on parallel Linux systems. It is designed to provide its data to higher level application tools so that observations can be made about application performance and system performance. Although KTAU shares, with ARUM, the goal of combining application and system performance, it requires kernel patching to use.

ARUM combines application and system measurements to describe application performance in the context of the application's runtime environment. Several other research efforts explore the combination of application performance and system performance. TAU integrates the kernel measurements from KTAU [22]. CrossWalk extends application level profiling by profiling system calls made on behalf of the application [19]. Vertical profiling combines event data from several layers of a system (primarily from hardware counters) to create a visual description of whole system performance [3]. Although these approaches are promising, none completely satisfy the goals of ARUM. Integrating KTAU with TAU requires kernel patching; CrossWalk does not investigate the runtime environment outside the application's processes; and vertical profiling does not include specific enough application level measurement.

3 Design of ARUM

Our initial motivation for the ARUM project was the need for a tool for use by performance analysts and system designers of high end Linux-based servers. However, ARUM is potentially useful by a wider audience for analysis of application and system performance on current and near future server class machines and other high end platforms, like clusters. These goals imply several design requirements.

In order for ARUM to be an adoptable tool, it must be easy to use. To make ARUM easy to use, we designed it so that the applications we measure do not require source modification, relinking, or recompiling. Additionally, ARUM accesses hardware counters through a loadable kernel module, and this does not require a kernel patch and subsequent recompilation of the kernel. Furthermore, ARUM does not require root level privileges to use (however, root access is required to load the module). We also designed a simple command line interface through which the user specifies the measurements that ARUM will collect and the application that ARUM will launch.

A second requirement is that ARUM must be lightweight: It must minimize perturbation to the application and to the system; and its overhead must be acceptable to the target users (it should take significantly less time to conduct a performance study using ARUM than using current methods). In designing a lightweight tool, we carefully considered how ARUM collects measurements, how frequently it collects measurements, and the amount of measurement it performs in a single measurement interval. We discuss our techniques in greater detail in the following sections on ARUM's measurement facilities.

Our third requirement is that we support the architectures important to our target users: current, older, and unreleased Intel and AMD architectures. To this end, ARUM currently supports a subset of Intel and AMD architectures.

Finally, we want the measurement report to be descriptive enough so that application performance bottlenecks and unexpected system behavior are easy to detect. To create a descriptive report, we decided that ARUM needs to collect measurements about the application, its resource usage, its runtime environment, and hardware events. Figure 1 shows the design of ARUM, including its measurement modules. In the following sections, we present the design of these measurement techniques.

3.1 Thread Level Resource Usage

We collect resource usage information for each thread of the measured application, using the getrusage system call. ARUM does a `fork` and `exec` to launch the application. When the child process completes, we call `getrusage` once, asking it to report on ARUM's children. Currently we only present system time and user time. However, by expanding the options to ARUM's command line interface we can easily report other `getrusage` statistics, like the number of page faults.

We achieve two benefits from this type of measurement. First, we obtain a breakdown of resource usage. With the statistics that ARUM currently reports, we can compare the proportion of user time to system time. If we enhance ARUM so

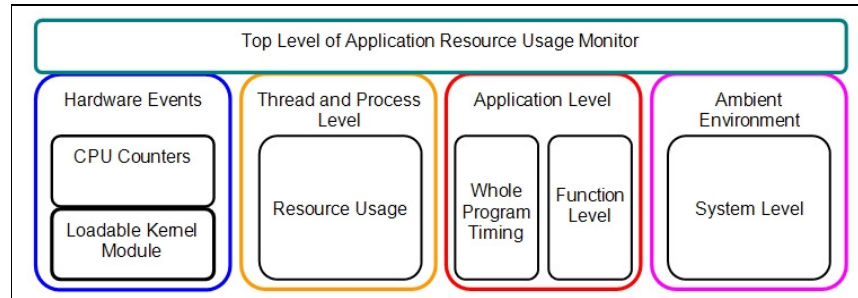

Fig. 1. ARUM Design. At its highest level, ARUM launches an application and makes calls to the measurement modules based on the measurement configurations requested by the user. ARUM contains four measurement facilities: one for accessing processor specific hardware events, one to measure resource usage of threads and processes; one to measure application level metrics; and one to measure the ambient environment. ARUM accesses per CPU counters through its user level counters module and through a loadable kernel module. ARUM measures process and thread level resource usage using the `getrusage` system call. ARUM collects two kinds of application level metrics. It uses timing routines to measure the entire program's execution, and it uses dynamic instrumentation to measure function level metrics. ARUM collects measurements about the ambient environment through its system measurement module.

that it reports additional `getrusage` statistics, we would be able to detect whether or not applications are resource constrained. For example, a high number of page faults could indicate a memory constraint. The second benefit is that the technique is lightweight - the system call happens after the application executes; so the measurements collected do not create additional overhead or perturbation to the application.

3.2 Hardware Events

To obtain architecture related performance data we use hardware counters to collect architecture specific metrics. This type of data is useful for understanding how an application interacts with an architecture. Architectures often support hundreds of measureable events, including instructions completed; instruction stalls; mispredicted branches; accesses, hits, and misses to the data and instruction caches; and floating point unit operations.

ARUM accesses hardware counters through a loadable kernel module, which installs `/dev/arum`. Using the device interface, ARUM makes standard calls to `open`, `close`, `read`, and `write` to manipulate and manage the hardware counters it is interested in. The kernel module's `open` procedure opens the device and uses a spinlock to inspect a flag that indicates if the counters are in use (and if the flag is not set, it sets the flag); `write` initializes data structures for storing counter data, configures the counters and other supporting registers, and starts counting; `read` stops counting, reads the values that accumulated in the counters and stores the values in the counter data structures, and copies the data into user space; and `close` cleans up

the counter data structures, unsets the "in use" flag within a spinlock construct, and closes the device. If an error occurs in any of these procedures, the module copies an error code into the user space and ARUM decides, based on the severity of the error, how to proceed —in most cases, ARUM disables the hardware counting mechanism and continues performing its other measurement tasks.

In our implementation of the hardware counter component, we minimize overhead and perturbation in two ways. First, we restrict when ARUM makes its system calls to the kernel module to immediately before launching the application (`open` and `write`) and immediately after the execution completes (`read` and `close`). This limits the amount of perturbation caused by ARUM. Second, to avoid the consequences of not knowing that a counter overflowed and to avoid the cost of generating an interrupt for each overflow occurrence, we use a kernel timer construct. We calibrate the timer based on cycle time and the width of the counters. When the timer expires, the module reads all of the enabled counters on all the processors, updates the accumulators in the counter data structures, resets the enabled counters to zero, resets the timer, and resumes counting. This technique will read some counters before they are close to overflowing; but it ensures that counters do not overflow, and it avoids the interrupt driven situation, which leads to a "noisy" environment in multi-processor and multi-core system where counters overflow at different times on different processor cores.

Our implementation counts events on the specified counters on all of the processors. We need to do this because we want to obtain valid counts regardless of application thread migration among the processors. We use the Linux shared workqueue to specify counter related work tasks and to assign the work to a thread on each processor. We use `schedule_delayed_work_on`, from the workqueue interface, to specify a processor; and we use several macros from `cpumask.h` to select the processors. For example, we use `for_each_online_cpu` to iterate through each online processor when we configure the counters and registers, when we enable and disable counting, and when we read the counters.

The hardware counter measurement facility is optional in ARUM. To collect counter data, the user supplies ARUM with a list of hardware events at the command line or with the name of a file that contains the list. Before opening the arum device, ARUM verifies, using the `CPUID` instruction, that it supports the processor architecture; and it verifies, using internal architecture specific files, that the architecture supports the events listed by the user. If either one of these verifications fail, ARUM disables the hardware counting mechanism and proceeds with launching the application and collecting the other kinds of measurement data.

Currently, we support hardware counter access in ARUM for several AMD and Intel architectures, including AMD Opteron, Intel Core microarchitecture, and the Intel NetBurst microarchitecture.

3.3 Application Level Measurements

Application level measurements give insight into the performance of an application. Application metrics range from course-grained to fine-grained measures. In general, course-grained measurements cause far less perturbation and overhead than

fine-grained measurements, but course-grained measurements often do not convey enough information about application performance. For example, a common course-grained metric is the total execution time or wall clock time of an application. This measure requires two system calls to obtain a timestamp, one immediately before the application begins execution and one immediately following execution. A performance analyst uses execution time as the most basic of measures to determine if an application runs as expected, but the metric does not contain enough detail if the execution time is greater than expected. To discover which parts of the application significantly contribute to the execution time, an analyst must collect additional finer grained measurements, such as counts, timings, and timestamps of function calls.

We combine both course-grained and fine-grained application measurements in ARUM. By default, ARUM collects the execution time. As part of the ARUM command line, we allow users to specify which functions to measure. We use dynamic instrumentation [5] to insert timing routines into the running executable at the specified function entries and exits. With this technique, we capture function call counts, the duration of each call, and derive the average time spent in each measured function. The function level report identifies which functions consume the most time, indicating to the analyst which functions have the most potential (through optimization) to improve the application's execution time.

One of the reasons why we decided to use dynamic instrumentation is because this method requires the least amount of the user's time and management. The user does not have to modify application source code with timing routines or calls to an ARUM library; the user does not have to recompile or relink the application; and the user does not have to manage multiple source and binary versions, maintaining a mapping between versions and measurements. Another advantage of dynamic instrumentation is that it operates on the running executable and does not modify the compiled binaries (a technique that differs from binary rewriting).

To maintain low overhead and to prevent perturbation that significantly alters the measured application from the unmeasured application, we must limit the amount of function level measurement. Dynamic instrumentation allows us to compare, on the fly, the overhead of instrumenting and measuring a function call to the execution time of a function. If the overhead to instrument and measure a function is not significantly less than the execution time of the function, then the function is not worth instrumenting. By doing this comparison on the fly, a tool can decide if subsequent calls to the function should be measured.

We are investigating techniques to dynamically restrict function level measurement, where ARUM decides whether or not to measure a function call. Related work in designing a lightweight tool that employs dynamic instrumentation is self-propelled instrumentation [18], a technique developed at the University of Wisconsin which collects application function traces for large scale parallel applications using a lightweight form of dynamic instrumentation. They show a significant reduction in overhead using self-propelled instrumentation compared to other forms of dynamic instrumentation.

3.4 Runtime Environment

In addition to resource usage, processor specific hardware events, and application level timing metrics, performance analysis requires information about the application's ambient environment. In the fully implemented ARUM, we collect data similar to that reported by utilities like *sar*, *iostat*, and *vmstat*, including information like the number of processes running on the system, the number of interrupts, IO statistics for disk and network devices, and main memory and virtual memory statistics. This type of information is important because it adds descriptive information about the environment that is outside the application process. For example, consider a situation where application level measurements show that an application spends most of its time in functions involving network communication, the analyst knows that the application's execution time is longer than what it should be, and the ambient environment measurements show a higher than expected number of dropped packets for one of the network devices. With the addition of the ambient environment information, the analyst obtains a performance picture that suggests investigating the network device before making code changes to the application.

4. Example of Using ARUM

In this section, we show an example of using ARUM with thread level resource usage measurement and hardware event counting. Of the four measurement components, we have implemented these two. In this example, we monitor the pChase Benchmark [30], and we configure ARUM to monitor four hardware events: data cache accesses; data cache misses; data cache accesses that miss in the L1 data TLB and hit in the L2 data TLB; and data cache accesses that miss in both the L1 and the L2 data TLBs.

In ARUM's command line, we provide a list of the hardware events we want ARUM to monitor and we provide the name and associated arguments of the executable that we want ARUM to measure. In the command line example that follows we request four hardware events; and we configure pChase to conduct five iterations, using two threads and four memory references per thread:

```
./Arum --events "dcache_accesses dcache_misses \
L1_DTLBmiss_L2_DTLBhit L1_L2_DTLBmiss" \
--launch "./pChase64_SMP -r 4 -t 2 -i 5"
```

ARUM produces a text report. In Figure 2, we show a report for the configuration given above. The first part of the report presents the thread level resource usage data; and in this example the user time is 2.3 seconds and the system time is 0.3 seconds. The next line in the report shows the wall clock time, as measured by ARUM, for the pChase execution. We can readily see that this execution of pChase is multi-threaded, as the user time is greater than the measured wall clock time. The user time measurement is a sum of the user time of all threads belonging to the application. The hardware event data is reported in the final portion of the report. This is presented in tabular format, showing the counts for each event on each cpu. We see a large

```
ARUM REPORT:
User time is 2.3 seconds
System time is 0.3 seconds
Application Elapsed time is 1.413597 seconds
Performance Counter Monitoring Results:
Event Name                CPU          Count
dcache_accesses            0       112630691
dcache_accesses            1       121917356
dcache_misses              0        19377411
dcache_misses              1        19547246
L1_DTLBmiss_L2_DTLBhit     0           71342
L1_DTLBmiss_L2_DTLBhit     1           70674
L1_L2_DTLBmiss             0          873462
L1_L2_DTLBmiss             1         1036276
```

Fig. 2. An ARUM Report. This report shows thread level resource usage data; the wall clock time for the measured application; and counts by CPU for four hardware events.

number of data cache accesses; of these accesses 16.6 % are data cache misses. We see that some of the data cache accesses miss in the L1 data TLB, but hit in the L2 data TLB. However a larger number of data cache accesses miss in both the L1 data TLB and the L2 data TLB.

5. Measuring the Overhead of ARUM

We conducted several experiments to measure the overhead of ARUM. We measured ARUM's pre-processing and post-processing stages which do not impact the execution time of the application because they occur outside the application's execution period. We also investigated the overhead that ARUM adds to the execution time of the monitored application.

We conducted all of our measurement studies on a 1.8 GHz dual-core AMD Opteron system with 2 GB of main memory, running CentOS 5, a Linux operating system that closely resembles Red Hat Linux. The AMD Opteron architecture provides four 48-bit performance counters per processor core, allowing each processor core to monitor a maximum of four different events.

In our studies, we used the pChase Benchmark, a pointer chasing memory performance benchmark that measures the latency and bandwidth of memory references [30]. The benchmark user specifies parameters such as the cache line size, the page size, the overall problem size, and the number of threads. The benchmark selects a page to reference and then references all cache lines before referencing the next page.

5.1 Performance Study of ARUM's Pre-Processing and Post-Processing Stages

We present a breakdown of the pre-processing and post-processing costs of ARUM. The pre-processing costs occur before ARUM launches the monitored application, and the post-processing costs occur after the monitored application completes its execution. As a result, these costs do not impact the execution time of the measured application. However, because these costs may be noticed by the user of ARUM, we think it is important to present this data.

5.1.1 ARUM Pre-Processing Costs

Before launching an application, ARUM first sets up the measurement environment. In Table 1, we show ARUM's setup costs, in microseconds, for configurations using only thread level resource usage measurement and for configurations using thread level resource usage measurement and one to four hardware counters per core. For each configuration, we present the arithmetic mean of 10 trials, which falls within a 90 percent confidence interval with five percent error.

We divide the steps that contribute to ARUM's setup costs into two phases. The first phase includes parsing the user supplied arguments and creating data structures to store the application's timing data and the thread level resource usage data. The second phase includes verifying that the requested hardware events are compatible with the processor architecture; creating data structures to store the hardware counter data; and configuring the hardware counters. ARUM bypasses the second phase if hardware event counting is not enabled.

From Table 1, we see an increase of nearly 40 times in the costs of the first phase in comparing the configuration without hardware event counting to the configuration with one monitored hardware event. We see the costs in the first phase continue to increase with the addition of hardware events, however, the increase is at most 10 percent. We see that the costs of the second phase grow with the number of counters per core that ARUM configures. The percent increase in the cost of the second phase going from the one-counter configuration to the two-counter configuration is 93.6%; the two-counter configuration to the three-counter configuration is 46.5%; and the three-counter configuration to the four-counter configuration is 32.8%.

The cost of each of the pre-processing phases increases with the number of counters, but the cost increase associated with the second phase is much greater. Although the cost of the second phase grows with the number of counters, there are several reasons why this cost is tolerable. First, this cost is a one-time setup cost. Second, it does not impact the monitored application. Finally, the setup costs are well

Table 1. ARUM Pre-Processing Costs

	Thread-Level Zero Counters per Core (μs)	Thread-Level One Counter per Core (μs)	Thread-Level Two Counters per Core (μs)	Thread-Level Three Counters per Core (μs)	Thread-Level Four Counters per Core (μs)
Phase 1	29	1149	1263	1277	1290
Phase 2	N/A	8855	16839	24667	32755
Total Setup	29	10004	18102	25944	34045

under 1 second for present-day and near-future multi-core machines which may have as many as 32 cores and 18 counters per core.

5.1.2 ARUM Post-Processing Costs

When the application's execution completes, ARUM processes the collected data and presents a report to the user. In Table 2, we show the post-processing costs, in microseconds, attributed to processing the collected data for configurations of ARUM ranging from zero hardware counters to four hardware counters per core. We report the average from ten trials for each configuration; this data falls within a 90 percent confidence interval with five percent error.

We divide the post-processing stage into two phases. The first phase includes reading the event counts from the hardware counters, and it only occurs when hardware event counting is enabled. The second phase includes obtaining the resource usage data for the application's threads and calculating the execution time of the monitored application. We see that the time required to process the collected data relates to the number of counters; we see that the first phase has the most notable increase in data processing costs for each subsequent configuration. The percent increases in the first phase for subsequent configurations, starting with the one-counter configuration and going to the two-counter configuration, are as follows: 78.5%, 40.7%, and 46.4%. With exception to the configuration without hardware event monitoring, the differences in costs for phase 2 are not statistically significant.

The cost increase for the first phase grows with the number of counters. At first glance, this may stand out as something that could affect the scalability of ARUM. However, we do not find this to be a limitation of scalability. The post-processing cost is a one-time cost which does not affect the monitored application. Furthermore, we project that the total post-processing cost for current and near-future multi-core systems will remain under one second.

5.2 ARUM's Impact on Applications

In our second set of experiments we investigate ARUM's impact on applications. We look at the difference in execution times of an application when measured with ARUM and without ARUM. In the experiments in which we used ARUM, we ran several configurations of ARUM: with only thread level resource usage measurement, and with thread level resource usage measurement and hardware counter measurement, where we varied the number of measured hardware events from one to four.

Table 2. ARUM Data Processing and Reporting Costs

	Thread-Level Zero Counters per Core (µs)	Thread-Level One Counter per Core (µs)	Thread-Level Two Counters per Core (µs)	Thread-Level Three Counters per Core (µs)	Thread-Level Four Counters per Core (µs)
Phase 1	N/A	2100	3748	5275	7720
Phase 2	15	66	66	67	66
Total Setup	0	2166	3814	5342	7786

To measure the overhead of ARUM, we look at the difference in execution times of the pChase Benchmark when ARUM is used and when it is not used. To measure the benchmark's total execution time when ARUM is not used, we inserted `gettimeofday()` calls at the beginning and the end of the benchmark's `main()` function. When ARUM is used, ARUM measures the benchmark's execution time by calling `gettimeofday()` prior to launching the application and again when the application's execution completes. We report these execution times in Table 3 for six configurations. For each configuration, we report the average execution time of pChase from 15 trials; these data fall within a 90 percent confidence interval with five percent error.

In all of the configurations, pChase was configured with two threads and four memory references per thread. We set the pChase iteration count to five. We evaluate ARUM for thread level measurement and for combined thread-and hardware counter measurement. For the combined cases we add, in order: data cache accesses (loads and stores); data cache misses; the number of data cache accesses that miss in the L1 data TLB and hit in the L2 data TLB; and the number of data cache accesses that miss in both the L1 and the L2 data TLBs.

In examining the data in Table 3, we see that the difference in average execution times is at most 38,281 microseconds. Furthermore, we see that the average execution time does not always increase for each subsequent configuration. For example, the third configuration, where ARUM monitors thread level resource usage and one hardware counter, shows a decrease in execution time from the configuration in which ARUM only monitors thread level resource usage. We examined the data presented in Table 3 to determine if the differences presented are of statistical significance. Using a 90 percent confidence interval, we found that the differences are not statistically significant. From this, we conclude that ARUM does not significantly impact this benchmark. We expect this to be true of other short running benchmarks and applications. For longer running applications, where ARUM reads the counters periodically during the application's execution as a way to handle counter overflow, we expect ARUM's overhead to be statistically significant, but we expect the overall impact to remain low since reading hardware counters is a lightweight technique.

Table 3. Average Execution Time of the pChase Benchmark Under Six ARUM Configurations

Without ARUM (sec)	Thread-Level Zero Counters per Core (sec)	Thread-Level One Counter per Core (sec)	Thread-Level Two Counters per Core (sec)	Thread-Level Three Counters per Core (sec)	Thread-Level Four Counters per Core (sec)
1.393174	1.404154	1.383462	1.396225	1.421743	1.398483

Conclusion

We presented the design of Application Resource Usage Monitor and a performance study showing the overhead of ARUM in its current stage of development. The ARUM design contains four measurement components: process and thread level resource usage, architecture specific event counting, application level measurements, and measurements of the ambient environment. These four components allow ARUM to collect performance data about an application in the context of its runtime environment. We designed ARUM as a lightweight tool, with ease of use and user adoptability as goals. To this end, ARUM does not require patching and recompilation of the Linux kernel and it does not require relinking or recompilation of the application. It runs from the command line, launching the application and collecting measurements as specified by arguments to ARUM.

We have implemented two of the four measurement aspects: ARUM collects basic thread level and process level resource usage statistics, and it collects hardware counter event data. We conducted a performance study of ARUM at this point in its development, and we presented that study in this paper. We investigated the overheads associated with ARUM's pre-processing and post-processing stages; and we investigated the overhead that ARUM adds to the monitored application. These initial overhead studies support our claim that ARUM is a lightweight tool, causing very low overhead to the monitored application. Future studies, to strengthen this claim, will include additional benchmarks monitored on both AMD and Intel architectures.

We plan to add hardware counter support for additional AMD and Intel architectures. We are also continuing development of ARUM by implementing the remaining measurement facilities: function level measurements using dynamic instrumentation and measurements of the ambient environment. The current implementation of ARUM targets high end server environments. In the future, we plan to extend ARUM to support with support for non-shared memory parallel environments, such as Linux clusters.

Acknowledgments

This work was sponsored in part by a grant from the PSU Center for Sustainable Processes and Practices.

References

1. BERRENDORF, R. AND MOHR, B. 2003. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (version 2.2)*. Zentralinstitut für Angewandte Mathematik (ZAM), 2003. Retrieved 12-18-07 from: http://www.fz-juelich.de/jsc/PCL/doc/pcl/pcl.pdf.

2. BROWNE, S., DONGARRA, J., GARNER, N., HO, G., AND MUCCI, P. 2000. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications 14*, 3 (Fall), 189-204.
3. HAUSWIRTH, M. 2005. *Understanding Program Performance Using Temporal Vertical Profiles*. Ph. D. Dissertation. University of Colorado, 2005.
4. HOLLINGSWORTH, J. K., MILLER, B. P., GONÇALVES, M. J. R., NAIM, O., XU, Z., AND ZHENG, L. 1997. MDL: A language and compiler for dynamic program instrumentation, In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, California, November 1997, IEEE Computer Society, Los Alamitos, CA, 201-212.
5. HOLLINGSWORTH, J. K., MILLER, B. P., AND CARGILLE, J. 1994. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the IEEE Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, IEEE Computer Society Press, 841-850.
6. JONES, T. R., BRENNER, L. B. AND FIER, J. M. 2003. *Impacts of Operating Systems on the Scalability of Parallel Applications*, Technical Report, Lawrence Livermore National Laboratory, 2003.
7. KALE, L. V., KUMAR, S., ZHENG, G. AND LEE, C. W. 2003. Scaling molecular dynamics to 3000 processors with projections: a performance analysis case study. In *Proceedings of the Terascale Performance Analysis Workshop, International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003. Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2660, 25-32.
8. *Kerninst API Programming Guide, release 2.1.2.* 2007. University of Wisconsin, Paradyn Parallel Performance Tool. Retrieved 3-01-2008 from http://www.paradyn.org/kerninst/release-2.1.2/kapiProgGuide.html.
9. KNAPP, R. L., KARAVANIC, K. L., AND PASE, D. M. 2007. Detecting runtime environment interference with parallel application behavior. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium* (IPDPS-SMTPS 2007), Long Beach, CA, March 2007, IEEE Computer Society.
10. KOLA, G., KOSAR, T., AND LIVNY, M. 2004. Profiling grid data transfer protocols and servers. In *Proceedings of the 10th International Euro-Par Conference*, Pisa, Italy, August 2004, DANELUTTO, M., VANNESCHI, M., AND LAFORENZA, D, Eds. Springer, 452-459.
11. KUFRIN, R. 2005. Measuring and improving application performance with PerfSuite. *Linux Journal 135* (July).
12. LEVON, J. 2007. *OProfile Internals*. Retrieved 12-18-07 from http://oprofile.sourcefourge.net.
13. LEVON, J. 2007. *OProfile Manual*. Retrieved 12-18-07 from http://oprofile.sourcefourge.net.
14. LONDON, K., MOORE, S., MUCCI, P., SEYMOUR, K., LUCZAK, R. 2001. The PAPI cross-platform interface to hardware performance counters. In Proceedings of the *Department of Defense Users' Group Conference*, Biloxi, Mississippi, June, 2001.
15. MAVINAKAYANAHALLI, A., PANCHAMUKHI, P., KENISTON, J., KESHAVAMURTHY, A. AND HIRAMATSU, M. 2006. Probing the guts of Kprobes. In *Proceedings of the Linux Symposium*, Ottawa, Ontario Canada, July 2006, *2*, 101-116.
16. MELLOR-CRUMMEY, J., FOWLER, R. AND WHALLEY, D. 2001. Tools for application-oriented performance tuning. In *Proceedings of the 15th international Conference on Supercomputing (ICS '01)*,Sorrento, Italy, June 2001, ACM, New York, NY, 154-165.
17. MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K. AND NEWHALL, T.

1995. The Paradyn parallel performance measurement tool, *IEEE Computer 28*, 11 (November), 37-46.
18. MIRGORODSKIY, A. V. 2006. *Automated Problem Diagnosis in Distributed Systems*, Ph.D. Dissertation, University of Wisconsin-Madison, 2006.
19. MIRGORODSKIY, A. V. AND MILLER, B. P. 2003. CrossWalk: A tool for performance profiling across the user-kernel boundary. In *Proceedings of the International Conference on Parallel Computing (ParCo '03)*, Dresden, Germany, September 2003, G.R. JOUBERT, W.E. NAGEL, F.J. PETERS, W.V. WALTER, Eds. Elsevier, The Netherlands, 745-752.
20. MOHR, B. AND WOLF, F., 2003. KOJAK – A tool set for automatic performance analysis of parallel programs. In *Proceedings of the 9th International Euro-Par Conference*, Klagenfurt, Austria, August 2003, H. KOSCH, L. BÖSZÖRMÉNYI AND H. HELLWAGNER, Eds. Springer, 2790, 1301-1304.
21. MUCCI, P. *PapiEx*. Retrieved 12-20-07 from http://icl.cs.utk.edu/~mucci/papiex/papiex.html.
22. NATARAJ, A., MORRIS, A., MALONY, A., SOTTILE, M. AND BECKMAN, P. 2007. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*, Reno, Nevada, November 2007.
23. PAPI. 2007. *Component PAPI Technology Pre-Release (PAPI 3.9.0)*. April 2007, Innovative Computing Laboratory, University of Tennessee, Knoxville. Retrieved 3-1-08 from http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI-C.html.
24. PAPI. *PAPI*. Innovative Computing Laboratory, University of Tennessee, Knoxville. Retrieved 5-25-06 from http://icl.cs.utk.edu/papi/.
25. PETRINI, F., KERBYSON, D. J., AND PAKIN, S. 2003. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Phoenix, AZ., November 2003, IEEE Computer Society, Washington, DC, 55.
26. PHILLIPS, J. C., ZHENG, G., KUMAR, S., AND KALÉ, L. V. 2002. NAMD: biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Baltimore, MD., November 2002, IEEE Computer Society Press, Los Alamitos, CA, 1-18.
27. SCHULZ, M., 2006. Open | SpeedShop, An open source framework for parallel performance analysis. Presentation at *Inter-Agency Working on High-End Computing*, July, 2006, UCRL-PRES-222050. Retrieved 3-01-07 from http://www.openspeedshop.org/presentations.html.
28. SHENDE, S. AND MALONY, A. D. 2006. The TAU parallel performance system. *International Journal of High Performance Computing Applications 20*, 2 (Summer), 287-311.
29. TAMCHES, A. AND MILLER, B. P. 1999. Using dynamic kernel instrumentation for kernel application and tuning. *The International Journal of High Performance Computing Applications 13*, 3 (Fall), 263-276.
30. *The pChase Benchmark Page*. Retrieved 2-12-07 from http://pchase.org/.
31. WOLF, F. AND MOHR, B. 2003. Hardware-counter based automatic performance analysis of parallel programs. In *Proceedings of the Mini-symposium 'Performance Analysis' Conference on Parallel Computing (PARCO '03)*, Dresden, Germany, September 2003, Elsevier.