



Hardening Execution through Precise Process Accounting



CONTENTS

4	Introduction
4	Problem Statement
5	PPA Usage Goals
5	Hardening Requirements
10	Prior Work
10	PPA Design
17	Future Direction and Long-Term Focus
17	Derivative Works
18	Conclusions
19	Appendix A – References
19	Appendix B – Glossary
20	About the Author

Abstract

This paper provides an overview of the Motorola Precise Process Accounting (PPA) Linux OS patch for carrier-grade environments. PPA improves precise CPU and scheduling activity accounting aids for five-nine availability as well as hardens systems capacity management, execution performance and incident root-cause. This paper presents use cases and hardening for precise CPU, scheduling latency and event accounting, followed by implementation advice for the Linux kernel.

Introduction

Current CPU accounting introduces many gaps in carrier-grade environments; Precise Process Accounting (PPA) is an additional timing and activity framework in the kernel that coexists with current CPU accounting. PPA refers to precise:

- measurement of per thread execution intervals,
- measurement of system wide execution intervals,
- accounting of scheduling events,
- accounting of scheduling latencies,
- and enforcement of execution limits.

PPA requirements are derived from Motorola's years of development and deployment experience in carrier-grade environments. PPA is intended to solve day-to-day operational problems in carrier-grade environments, improve reliability, harden the system during development and expose risks or probable incidents that otherwise would remain unknown until deployment.

PPA hardens a Network Element (NE) in several ways:

System Characterization: system performance, latency and execution behavior can be adequately characterized during development to prevent surprises later, for example, after field deployment.

Capacity Management: available compute bandwidth can be managed reliably, based on available CPU bandwidth the NE can selectively process inbound requests, throttle back low priority activities or distribute/migrate load.

Field incident root-cause: high CPU load or excessive latencies are difficult to spot and establish their root cause, and often get attributed to unrelated causes. To carrier-grade customers (i.e., carriers) that require immediate analysis and resolution of incidents, PPA provides data or clues to solve complex incidents.

Problem Statement

Generally, for desktop users, precise CPU utilization may not be required. However, in high reliability environments (i.e., carrier-grade) precise CPU utilization is important. In carrier-grade environments CPU usage is continually sampled and monitored. For example, in a Telco central office's maintenance console, any readings out of the ordinary draw immediate attention, raise alarms, result in detailed analysis and vendor inquiry. However, there are measurements beyond just CPU accounting that are of high importance and will be discussed later. Considering complex networks where carrier-grade NEs are deployed, the feature set developed on a platform is exhaustive and, yet, much of the reliability hinges on accurate execution accounting, something considered trivial and taken for granted by end users.

Figure 1 shows an example of thread execution between tick intervals. The samples (taken every tick) occur during idle time and, therefore, the overall accounting will be accrued to idle. Increasing sampling may improve CPU accounting but impacts performance. Scaling (i.e., HZ) is sometimes used for other needs, such as improving scheduling latency, timeout granularity of sleep or polling calls.

Although Figure 1 exposes a situation in which excessive CPU utilization is not reported, the inverse

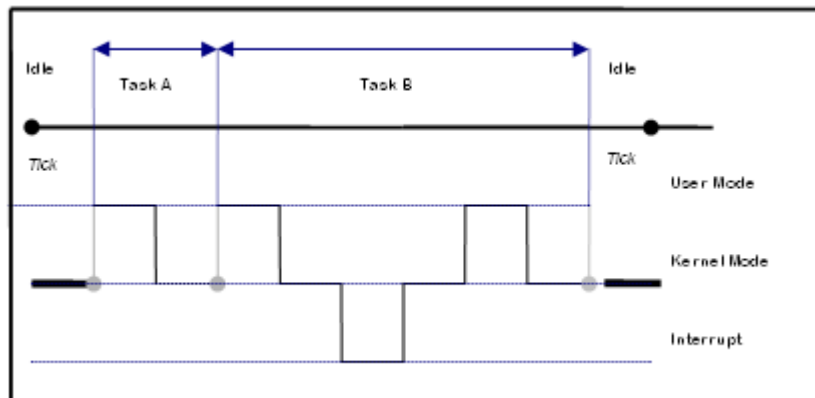


Figure 1: Precise CPU and activity accounting

of over-utilized CPU capacity may be just as disruptive. For example, it can lead to such undesired actions as alarms, load throttling, load distribution or activation of standby. Today, processors may execute many instructions during a tick interval. For example, a 1GHz processor (modest for today's standards) completing one instruction per cycle may execute a 50-line C function 6000 times in a 1-ms tick interval. Therefore, certain execution patterns may easily go unaccounted and lead to incidents.

PPA introduces overhead in hot kernel code and, since precision is not relevant to all environments, the whole PPA framework can be disabled. On the other hand, in critical environments where precise CPU, activity and scheduling accounting is required, PPA can be enabled incrementally. Each level is designed to provide additional insights to complex issues. The need for PPA has primarily arisen from the carrier-grade environment. However, PPA is applicable to any environment where such level of precision is required.

PPA Usage Goals

The goals that have been defined follow user preferences and practices identified over years of deployment and use in carrier-grade environments. Users prefer simple and available means to identify CPU/execution-related performance bottlenecks and there is resistance to use profilers (for example, those based on CPU performance counters), OS trace tools or instrumentation of executables. The latter are used as a last resort; sometimes familiarity with processor architecture is required. Users prefer to use the same means in production systems that were used during development. The goal of PPA implementation is to make the framework simple to use and available in all execution environments, and to provide top-down performance measurements (from system down to individual threads). The data PPA provides maybe all that's needed or provide enough insights to apply the above mentioned tools.

Hardening Requirements

This section addresses PPA requirements identified in carrier-grade environments, while the final section will cover how PPA fulfills these requirements. The manual page should be consulted for actual features supported by the version of PPA in use.

User Requirements

1. All CPU Usage Intervals shall be time-stamped with a high resolution counter.

Without precise time-stamping users are blind to CPU usage and probable incidents. The NE may not be hardened properly, since actual CPU usage and schedule measurements were unknown during the implementation phase. Developers rely on occurrence or lack of incidents. However, not having incidents does not necessarily mean the system is hardened. It is unknown to what extent incidents are due to statistical accounting. In many cases, these occurrences are not root-caused properly and the incident may be masked by a random fix. For five-nines reliability, unknown CPU execution and scheduling measurements are a problem.

2. All measurements and accounting activity shall be available per OS-visible CPU.

Symmetric Multi-Processing (SMP) NEs are common and there are many flavors of SMP, such as:

- Simultaneous Multi-Threading (SMT),
- Chip Multi-Processing (CMP),
- CMP/SMT, and
- Non Uniform Memory Access (NUMA).

In addition, the cache configurations may vary on CMP architectures, as the second- or third-level cache may be shared or dedicated. Application CPU affinity is also common. A carrier-grade application is usually composed of several functional areas, such as: High Availability (HA), System Management, Communication layer, Database and Mobility Management. CPU affinity is utilized to isolate these functionalities to guarantee performance (execution QoS) and response time; the OS visible CPUs are partitioned among functionalities. Processes may either be allowed to float among a subset of CPUs or may be bound to a specific CPU. With new SMP architectures, the traditional execution SMP model may not apply and, therefore, per-CPU statistics are required to assess performance problems. With per-CPU statistics available, it is possible to determine execution conflicts or incorrect affinity assumptions. For example, a process may accomplish more work on one CPU than all others combined or two non-cooperative processes may be executing

on different SMT threads. The aim is to provide immediate and general insights to SMP performance in order to resolve conflicts or complement use of profiling tools.

3. The Precise Accounting framework must be usable in the field, on production systems.

The framework must be lightweight to be usable in the field, as customers will not allow the use of profiling, process trace and OS trace tools on a production system servicing paying customers, even less so during peak hours. The means used to provide critical performance, execution and schedule insights must be part of the delivered OS, exhibit unnoticeable performance degradation, and be used during general operation of the NE. Given that most challenging incidents occur during peak hours (i.e., rush hour), it's important that the precise accounting framework is lightweight and scalable.

4. Fully enabled accounting should have no measurable performance degradation when running SPECint2006 or SPECint_rate2000 (<http://www.spec.org>).

The Precise Accounting framework must exhibit small overhead and should be unnoticeable when executing SPEC benchmarks. The benchmark is indicative of many loads and is a generic measure used by hardware and OS vendors, as well as product groups to compare hardware versions or operating systems. A noticeable drop in benchmark performance is unacceptable.

5. Per-task measurement of execution user, system and interrupt (in thread context) modes shall be accumulated. This applies to kernel-created threads as well. Enable of thread accounting can be performed from another thread/process.

Precise and per-thread user, system and interrupt time must be supported. There may be other execution states of interest that further identify performance issues, but these are fundamental. Precise thread measurements help isolate performance issues, as well as root-cause incidents (for example, third-party/vendor application/library) and performance problems. Need for accurate user mode accounting is obvious because, generally, most work is accomplished in user mode.

Excessive system mode execution exposes performance issues in device drivers and system calls (i.e., poll(), getmsg()). This doesn't neces-

sarily mean problems in system code but may expose unexpected kernel execution behavior to users.

Interrupt execution in thread context may aid or degrade performance, for example if the interrupted thread is the recipient of the data, the interrupt is moving up the stacks it may benefit from cache reuse. However, unrelated interrupts may degrade performance of a functional area. It is a common practice to bind network interrupts and communication processes to same CPU(s). Today, hardware supports distribution of interrupts among OS-visible CPUs (usually based on priority of current CPU activity) or interrupts may be bound to a CPU or a subset of CPUs. Interrupt measurements provide valuable data, and various interrupt and process affinity schemes may be applied for optimal performance.

These measurements must be supported for kernel-created threads, as well. Kernel threads run in privileged mode. However, they can also make system calls, so general execution should be accumulated like user mode execution is for user threads and, likewise, for system calls.

Lastly, precise usage (usr, system, int) of several threads/processes may be used to determine loading conditions and scale services based on the current usage thresholds. Therefore, not only does overall system usage matter but also the sum of individual threads/processes.

6. Per-task occurrence count of system calls, interrupts and signals (by signal number) should be accumulated.

Per-thread occurrence count of system calls and interrupts allows users to further characterize performance on an SMP platform. For carrier-grade applications, processes tend to call a small subset of system calls after initialization, for example, a subset of Socket calls. The system call count provides insights to average execution per system call, while the interrupt count provides insights to average interrupt execution.

Signals are widely used in carrier-grade applications, and signals are expensive. Under some circumstances, a process may be flooded with signals such as signal I/O, flow control on signaling link or link down event. Insights into such events are required and especially useful when dealing with third-party software.

7. System-wide measurement of execution user, system, interrupt and idle modes shall be accumulated.

At the very least, users require overall system idle time and used time to manage capacity. Interrupt usage measurements are important to analyze performance issues due to CPU affinity. These measurements maybe all that's needed to analyze performance bottlenecks or may lead to further analysis, for example, per-thread statistics. In addition to analysis, these measurements are used to enforce execution thresholds. Once these thresholds are crossed, the NE will take appropriate actions to reduce the load or the load may be distributed.

8. System-wide occurrence count of system calls, interrupts and signals should be accumulated.

System calls, interrupts and signal occurrence counts are required for the same reasons as in the per-task occurrence requirement.

9. Per-process execution security limits should be precisely enforced.

System administration and maintenance activities as well as network accessed services (for example, xinetd) require precisely enforced execution limits. Simple exploits, like displacing cache contents, may lead to continuous 10x performance degradation and, on SMP, the exploit may be even more severe. On poorly hardened system it's not security exploits but ordinary services that may vastly degrade performance, as these activities may be low priority and degrade capacity.

10. Per-process execution profiling and virtual timers shall expire precisely.

To lesser extent, but with some interest, users require precise expiration of virtual and profiling timers. The data is made available with PPA and notification is configurable through current signal framework.

11. Facilitate per-thread security limit.

In Multi-Threaded (MT) applications, it's desirable to limit the per-thread execution limit. Setting a per-process execution limit is costly for highly-threaded processes because summation of all thread usage time is required in system interrupt handlers. However, for many applications, only a few threads perform most of the work. PPA can only facilitate CPU usage measurements that can be used to enforce per-thread execution limits. Expansion of per-thread

signals and the implementation of such a framework are not addressed by PPA.

12. All accounting activities may be disabled either at build or run-time. During run-time, accounting may be scaled variably. Fully disabled accounting during run-time should have no measurable impact on performance.

Most carrier-grade users want all of the features listed here. The probable overhead is taken into consideration when doing capacity planning and, in carrier-grade environments, predictability is more important. However, some environments, like mobile, may be sensitive to PPA (for example, time-stamping – `gettimeofday()`) and require options to scale measurements. Enabled Idle-time measurements at all times are acceptable but, beyond that, all measurements are run-time controllable – from complete disable of precise accounting to incremental enable of accounting – interrupt, system, CPU enforcement, timers, etc.

13. Measurements shall be continuous and available on-demand during run-time (for example, HA or CPU monitor), requiring no playback.

In carrier-grade environments, NEs manage capacity and the measurements must be instantly available for real-time analysis. For example, High Availability may use these measurements to activate a standby NE or the platform may use the measurements to prioritize requests, accepting only higher priority requests and denying low priority requests. Therefore, discrete sampling and playback analysis are not acceptable.

14. Per-process execution profiling and virtual timers shall expire precisely.

To a lesser extent, but with some interest, users require precise expiration of virtual and profiling timers. The data is made available with PPA. However, expiration is only configurable through current signal framework. If system mode accounting is enabled, then precise profile and virtual time will be accounted. If system mode is disabled, only precise profile time will be supported. Of most interest is use of SIGPROF and tools like gprof that use the signal to sample where execution time is spent. With precise accounting enabled, SIGPROF will reflect actual execution interval (usually a tick) of the process, aiding gprof.

15. Reasons and count of thread reschedule occurrences, such as preempt due to higher priority task ready to run, preempt due to time-slice expire and voluntary block or reschedule, shall be accumulated.

Reasons for scheduling decisions should be captured. These measurements are used for application tuning - for example, response time or improving performance primarily to prioritize activities relative to each other. Threads with high preempt count due to higher priority or, conversely, if another thread has a high reschedule count and response deadlines are not met, indicate that priorities should be re-evaluated. High voluntary reschedule count may indicate I/O operations should be consolidated, either by removing poll/epoll or implementing a high resolution sleep to consolidate I/O operations. High time-slice expiration reschedule count may indicate the need for the priorities or the time-slice (i.e., static priority) to be evaluated. These value measurements are used to associate incidents related to scheduling latencies or conflicts, for example, timeouts in user devices due to call setup latencies. A typical request may pass through several protocol layers (MTP1-MTP3, M2UA, M3UA and SCTP for SS7 architectures) before reaching call processing. Insights into scheduling latencies along the way are key in addressing incidents that a customer reports. This requirement and the next work together to associate incidents to latencies of various types.

16. Amount of time spent on run queue shall be accumulated, average scheduling latency shall be accumulated and schedule latency measured as the interval from thread activation to execution.

Thread excessive time spent on run queue measurement and schedule latency measurement provide insight into reasons for missed deadlines. From the time a task is scheduled to run (due to an asynchronous event – i.e., another process or packet arrival) until its context is switched to execute constitutes the schedule latency interval. Additionally, from the time a task is preempted (left in runnable state) until it executes again constitutes the run queue scheduling latency. This value provides users with vital measurements that expose reasons for missed deadlines.

17. Latency measurement occurrence count for schedule latency interval, time spent on run-queue, interrupt call interval and system call interval shall be accumulated.

In addition to the previous two requirements, users want to measure the occurrence count when these intervals cross a certain threshold. The count of schedule latencies, either due to initial schedule latency or reschedule while on run queue, over a certain threshold (for example 10 ms) provides vital data on reasons for missed deadlines, which is key in carrier-grade environments. Similar reasons can be applied to system calls and time spent in interrupts. This data can later be used to root-cause timeouts on services for incoming requests.

18. Deferred interrupt processing on each OS-visible CPU should be supported.

Typical carrier-grade activities make heavy use of timers and IP protocol stacks. CPU usage measurements for these activities, along with other deferred interrupt activities, are not available. It is important to have these measurements available during development and after deployment, as these activities may degrade application performance, various affinity schemes may be applied to reduce conflicts or the application may be redesigned, for example, by implementing one timer task and using IPC to communicate expiration.

19. Precise limit enforcement of CPU usage based on UID, GID or a list of process should be supported.

Today, CPU usage enforcement limits meet minimum user requirements, with the imposed limits applying only to one process. CPU execution limits, based on a User ID, Group ID or a list of PIDs, are required to isolate functional areas, enforce security and guarantee execution cycles. Carrier-grade NEs run conflicting applications, which compete for CPU, memory and I/O resources. These conflicts are difficult to model and virtualization is one way to isolate and control resource contention, and guarantee run-time execution. However, virtualization itself requires tuning, adds development overhead and cost. Consolidation is the primary reason for virtualization in carrier-grade environments. Users would like to have these conflicts resolved by the underlying OS and solving CPU execution limits is at the top of their list. Exclusive affinity is an additional method of CPU QoS execution, but it is currently unavailable.

20. Thread overload detection and limiting should be supported.

Runaway processes and threads, such as errors in looping code and bad invariant, are the most common incidents in carrier-grade environments. Users want to protect against system overloads, especially because many such overloads are recoverable or caused by low-priority activities. An overload policy needs to consider duration of continuous execution (detection) as well as means to impose and report overload protection (usually to HA software).

21. Context switch duration average should be known.

Effects of a context switch go well beyond just the context switch itself (i.e., cache, MMU). Knowing average context switch duration helps identify the cost of blocking; I/O may be coalesced by sleeping to prevent excessive context switching.

22. Provide data on transient threads or process between two measurement samples; both the count and CPU time consumed should be available.

Transient threads confuse developers and field engineers. Even with precise accounting, it's not possible to account for CPU usage when there is a high count of transient threads. These occurrences are not necessarily incidents and often are caused by legitimate operations (client/server or request/response) in which transient threads or processes (shell or Perl scripts) are spawned and exit between sampling intervals. At the very least, the number of transient tasks and accumulated CPU usage should be provided on demand to aid tools in reporting CPU usage and to fill in the unaccounted CPU reporting between a sampling interval.

Design Requirements

PPA has been implemented with the following requirements in mind:

1. Push post processing up to the application. Minimum work should be done in the kernel, which leaves most processing to intelligent tools in user mode.
2. Modifications are mostly implemented on the generic kernel code. For portability and maintenance, only the generic code base should be modified, thus, minimizing updates to architecture-specific files and limiting assembler code.

3. In situations where the execution outcome conflicts with the measurement, the execution takes precedence. For example, no locking should be required by the Precise Accounting framework. Other than the overhead needed to capture the measurements, the implementation should not alter kernel execution in critical paths. Accuracy is sacrificed for performance, primarily when the measurement accuracy is negligible.
4. Precise Accounting should seamlessly work along side with current accounting. Enabling or disabling Precise Accounting should have no impact on current native tick accounting. Current tick accounting is the basis for some /proc files and some system calls that provide CPU usage. The switch to Precise Accounting must be transparent with the exception that the values will be precise.
5. Acquisition of measurements shall be scalable and selectable in raw binary format, where selectable means all or a subset of the measurements shall be accessible. All information needed to interpret results should be available. Accounting measurement data should consider SMP architectures available today. The combination of NUMA, CMP and SMT may result in high memory overhead with per-CPU measurement options. Summarizing CPU reporting of several CPUs should be available to save on memory and processing overhead.
6. Measurements may be reset at any time to default values. Accounting measurements collected over a long period of time are not indicative of recent trends and, in carrier-grade environments, the system may be up for months or even years. The user should have the ability to reset the measurements to default values to determine most recent trends.

Limitations

Certain measurements may not be precise due to architecture (NUMA) time drift between CPUs. This scenario, essentially, only applies to a small subset of measurements, in which the interval difference of timestamps is between two CPUs. Affected measurements may include schedule latency and time spent on run queue, for example, if a thread migrates after the initial timestamp. Although not precise, the measurements may still be indicative of execution behavior.

Clock frequency scaling, used for a high resolution counter, is not taken into account, as power management is not a requirement in carrier-grade

environments¹. Carrier-grade architectures like ATCA have defined power envelopes so the system may execute from batteries only in the case of a power failure. To handle those situations, there are external power supplies to keep the system up. Power consumption and heat are considered at the board or server level. For the ATCA chassis, there are maximum per-slot power limitations. Power savings through frequency scaling are disabled – ACPI Processor Power States and CPU Frequency scaling. When the clock is scaled down, the intervals will be shorter and charged CPU usage will appear lower. Some frequency scaling governors depend on accurate CPU usage measurements. With low CPU use the frequency is scaled down, assuming less work needs to be done. However, this will conflict with latency and performance.

Prior Work

This section discusses progress made in precise accounting and compares these efforts against the precise accounting hardening requirements.

Linux 2.6

The recent stable release of the Linux kernel has introduced precise measurement of CPU execution and schedule latency. These values are used to determine the sleep average and priority for timeshare threads. Additionally, certain POSIX margin codes are supported thanks to precise measurement introduced in version 2.6². For each thread, the precise execution time (lumped together) is maintained, including idle thread. In addition, there are measurements related to task execution delays due to block I/O or swapping in³. To get at this data, the kernel needs to be compiled with CONFIG_SCHEDSTATS and CONFIG_TASK_DELAY_ACCT. However, access to idle time is not supported and almost none of the hardening requirements are supported.

Micro-state Accounting

The Micro-state Accounting patch meets several of the mentioned requirements and is an improvement over current tick-based accounting. It also implements few additional features beyond the hardening requirements. Those available include per-thread CPU usage, with several execution states supported, such

as precise execution, interrupt mode execution, time spent on active, inactive queue, futex, poll sleep or page fault - some of these additional measurements maybe of interest in carrier-grade environments. Per-interrupt execution measurements and additional timing sources are supported to compensate for power management frequency scaling. Micro-state Accounting partially, or fully, supports requirements 1, 3, 4, 5 and 12. However, many hardening requirements are not supported, for example, idle, user, per-CPU thread and system wide measurements, and CPU execution enforcement, to name a few.

PPA Design

PPA Implementation

Figure 2 is a state diagram of the Linux scheduler and Table 1 is the associated state table. Since not all of the states and transitions are captured⁴, use is only intended for PPA implementation to capture execution states of interest. Not included is the blocked state, which is implicitly entered to/from scheduler transition when returning from an interrupt or exception to user or system mode. The purpose of the state diagram and table is to identify insertion points to start end-measurement of CPU execution, latencies and occurrence counters.

Five execution CPU states are considered. These states may be entered and exited through thread-internal or scheduler transition. Internal refers to transitions made internal to the thread, such as system call and interrupt in thread context, and scheduler transitions made via a scheduler to another thread.

User: application running in least privileged mode

System Mode: application running in privileged mode

Exception Mode: considered here are only page faults resolved by the OS; the scope is limited, since exceptions are highly platform-specific. Exception time is currently not accumulated by PPA but may be implemented in the future.

Idle: idle task

Int: execution of interrupts

Embedded within some states are schedule points, with exception of USR:

Synchronous preemption

vol – voluntary due to a blocking call. From an execu-

1 Linux Foundation (previously OSDL) CGL 4.0 Specifications are available from <http://www.osdl.org>, and SCOPE Alliance Linux Profile is available from <http://www.scope-alliance.org>

2 The CPT and TCT POSIX Margin Codes are implemented with precise CPU execution.

3 Based on Linux kernel version 2.6.18

4 For example a thread may be in a blocked or exiting state.

	<i>Idle</i>	<i>Sys</i>	<i>Usr</i>	<i>Int</i>	<i>Exc</i>
<i>Idle</i>		S_{vol}		f^2	S_{vol}
<i>vol</i>		$S_{rfs - SMP}$		$S_{rfi - on SMP}$	$S_{rfe - SMP}$
		$S_{rfs - preempt}^1 - SMP$		$S_{rfi - preempt}^5 - SMP$	$S_{rfe - preempt}^5 - SMP$
<i>Sys</i>	S_{vol}	S_{rfs}	/	S_{rfi}	S_{rfe}
<i>vol, rfs</i>		$S_{rfs - preempt}^5$		$S_{rfi - preempt}^5$	$S_{rfe - preempt}^5$
<i>rfs</i> ³		S_{vol}		/	S_{vol}
		/			/
<i>Usr</i>		/		/	/
<i>Int</i>	/	S_{rfs}	/	S_{rfi}	S_{rfe}
<i>rfi</i>		$S_{rfs - preempt}^5$		$S_{rfi - preempt}^5$	$S_{rfe - preempt}^5$
<i>rfi</i> ⁷		S_{vol}		/	S_{vol}
		/			/
<i>Exc</i>	S_{vol}	S_{rfs}	/	S_{rfi}	S_{rfe}
<i>vol, rfe</i>		$S_{rfs - preempt}^5$		$S_{rfi - preempt}^5$	$S_{rfe - preempt}^5$
<i>rfe</i> ^{7,4}		S_{vol}		/	S_{vol}
		/			/

Table 1: Scheduler state table

required, followed by another internal transition to User mode.

- Timestamps are taken, stopped and accumulated on internal and scheduler transitions. PPA captures User, System, Interrupt and Idle modes, but it does not accumulate time for exceptions. An exception will be accumulated to the mode which the CPU was executing at the time of the exception.

5 May resume a preempted thread in system mode if preemption is enabled (2 or more privileged stack frames present)

6 Thread internal state transition

7 May occur while returning to privileged mode if preemption is enabled (2 or more privileged stack frames present)

8 Return from exception to exception considered invalid

- Intervals between internal transitions are used to accumulate thread precise accounting measurements. The sum of internal and scheduler transition intervals is used to accumulate system-wide precise accounting measurements.

- Per-thread time-stamping points are determined by following the dashed lines. The origin and target must be time-stamped and accumulated per each state, including nested transitions.

- In addition to usage accounting activity, accounting is also accumulated to provide counts on state entry and exit occurrence counts, both system-wide and per-task.

- The diagram aids in identifying a kernel generic

versus a hardware architecture time stamp code, represented by the dark shaded states.

- Majority of the time-stamping is relevant to single CPU but there are few intervals that use time-stamps between two CPUs. PPA uses same means as the kernel scheduler for time-stamping, for example, compensation for clock drift.
- There are transition points of short finite duration used to start/end time-stamp intervals. These measurements are dropped, for example, when switching between threads.
- There is code introduced in 'hot' execution paths and, therefore, options for run-time enable/disable are supported.
- Thread creation occurs in SYS mode. PPA memory is allocated and measurement starts upon return to user mode (return from fork()).
- Thread exit may occur in SYS or EXC modes. In these cases memory is reclaimed, threads usage is updated, and count of exiting threads and sum usage is updated.
- A signal is processed by resumption of a signal handler upon return to user mode and terminated with a system call. This time is accumulated by transitions from SYS to USR and USR to SYS, and charged to the signal handling thread.

PPA User Interface

There are several '/proc' files visible to the user to access and manage precise accounting. Following are the files and the format. However, the man page should always be referenced for most recent format and supported files. In addition, there are tools (ppa-top and others) and PPA acceptance tests (proof of correctness and others) that are not discussed here, for which the latest documentation should be referenced. At the time when this paper is being written, PPA is supported on x86, x86_64, Itanium and PPC32, while additional architectures maybe supported in the future. Additionally, PPA has been ported to Monta-Vista CGE 4.0.1, Debian 2.6.11 HP Telco Edition and 2.6.18.6 stock kernel. Even though a given architecture may not be supported, majority of the features are implemented in generic kernel code.

Located in /proc/<pid>/:

rawppa – this file contains raw precise accounting measurements for the process. The format of this file is shown in Figure 2. This file can be read and written to reset values to defaults. Only Total, Sys-

tem and Interrupt modes are populated, with other fields being filled with 0. This may change in future releases.

ppactl – this is an ASCII file, which can be read and written, and is used to manage per-thread precise accounting.

Located in /proc/<pid>/task/<tid>:

rawppa – this file contains raw precise accounting measurements for the thread. The file format is the same as that of Figure 2 and all fields are populated.

Located in /proc/:

rawsysppa – this contains raw system-wide precise accounting measurements. The format of the file is shown in Figure 3 and the file can be read or written to reset values to defaults.

Located in /proc/sys/kernel:

ppa – this is an ASCII file, which can be read and written, and is used to manage system-wide precise accounting and will override thread settings.

/proc/<pid>/ppactl

This file contains several configurable values to control process accounting. The updated values in this file take effect on the next context switch of the thread so, if the thread is executing, the values will not be enforced until reschedule of the thread.

Global flag: Enables or disables overall accounting for this process. The value is either 1 or 0 respectively. When enabled, precise interrupt accounting is also enabled. If this flag is disabled, all subsequent PPA features are disabled.

System Mode flag: Enables or disables system call accounting for this process. The value is either 1 or 0 respectively. If the Global flag is disabled, this flag is ignored.

Precise enforcement of CPU limits: enables precise enforcement of CPU usage execution (SIGXCPU). If the System Mode flag is enabled, both profile and virtual timers will expire precisely.

Scheduling latency threshold: defines schedule latency threshold. Scheduling latency greater then or equal will increment the schedule latency count (located in rawppa). The value is in nano-seconds. A value of 0 disables this feature.

Run-queue latency threshold: defines run queue latency. Scheduling latency greater then or equal will increment the run queue latency count (located in rawppa). The value is in nano-seconds. A value of 0 disables this feature.

Interrupt latency threshold: defines interrupt execution threshold. Interrupt execution interval greater then the threshold will increment the interrupt execution interval. The value is in nano-seconds. A value of 0 disables this feature.

System call interval threshold: defines system call execution. Threshold execution interval greater then the threshold will increment the system call execution interval. The value is in nano-seconds. Sleep within system call will not be applied to this threshold. A value of 0 disables this feature.

Example values stored in ppactl file may be:

```
1 1 1 15000000 15000000 0 0
```

To execute, enable overall and system mode accounting for the process, enable precise enforcement of CPU usage and profile/virtual timers, set schedule and run queue latency thresholds to 15ms.

The interrupt and system call thresholds are disabled. The file can be updated through 'echo' command to the file and changes will be applied to number of values supplied, with the remaining values remaining at their previous settings.

/proc/<pid>/rawppa and /proc/<pid>/task/<tid>/rawppa

This file contains thread and process measurements in raw binary format. This file is accessed via open(), read(), write(), and close() system calls. The user can seek to the file offset of interest or utilize I/O calls that keep the file offset unchanged. When written, all values in the file are reset to 0. By default file permissions are 'rw-r--r--'. All the collected measurements are summarized in Table 2.

The header contains the version, CPU count and a reserved field.

Header	0		2		4 - value 4 bytes in size			
	Version		CPU Count		Reserved			
Total CPU	cpu 0				cpu n-1	} 8 - byte values	
System Mode	cpu 0				cpu n-1		
Interrupt System Mode	cpu 0				cpu n-1		
Interrupt User Mode	cpu 0				cpu n-1		
System Call Count	cpu 0				cpu n-1		
Interrupt Count	cpu 0				cpu n-1		
Sched Block Count	cpu 0				cpu n-1		
Resched Count	cpu 0				cpu n-1		
Time Slice Expired	cpu 0				cpu n-1		
Preempt Higher Pri	cpu 0				cpu n-1		
Schedule Latency	cpu 0				cpu n-1		
Time on Run Queue	cpu 0				cpu n-1		
Latency Counters	0		8		16			24
	Latency Count		Runq Latency Count		Interrupt Count			System Call Count

Table 2: rawppa file

- The version number may be used by tools to interpret the data based on the version supported. The contents of this file may change between major revisions (2 bytes long).
- The CPU count is used to determine the number of each measurement type. This value may be less than the number of OS-visible CPUs, if CPU scaling is implemented and enabled (2 bytes long).
- Reserved field for future use (4 bytes long).

The next six arrays store execution measurements for the thread or process. Each element is 8-bytes long and dimension is of length CPU count stored in the header. All values are in nano-seconds.

- Total CPU Time – total time executed by the thread or process. This includes user, system modes and interrupts that preempt the thread or process.
- System Mode CPU time – system mode execution by the thread or process. This also includes interrupts that preempt the thread or process while executing in system mode.
- Next two vectors store interrupt execution in thread or process context. The first stores preemption of system mode execution and the second stores preemption of user mode execution.
- The next two vectors contain the system and interrupt counts, only valid for threads.

Accurate user mode execution may be determined by subtracting system mode execution and interrupt time of both modes. Accurate system mode execution can be determined by subtracting system mode interrupt execution.

The next four vectors store scheduling measurements, which are only valid for threads. Each element is 8-bytes long and dimension is of length CPU count stored in the header.

- The occurrence count thread blocked.
- The occurrence count thread was rescheduled for execution.
- The occurrence count thread was preempted due to time slice expiration.
- The occurrence count thread was preempted due to higher priority task ready to run.

The total number of context switches can be deter-

mined by adding the occurrence count of task blocking and preemptions, either due to timeslice expire or higher priority task.

The next two vectors store schedule latency metrics which are only valid for threads. Each element is 8-bytes long and dimension is of length CPU count stored in the header.

- Total schedule latency time - this is the interval from the time a thread is ready to execute until it actually executes on a CPU.
- Total run queue latency time – this is the interval from the time the thread was preempted until it executed again on some CPU.

To determine the average schedule latency, the total schedule latency should be divided by the number of times the task is blocked. To determine the average run-queue latency, the total run-queue latency time should be divided by the preemption count.

The next four values are each 8 bytes in length and are used to measure the defined thresholds, valid only for threads.

- Latency count – stores the occurrence count that exceeded the scheduling latency threshold.
- Run queue latency count – stores the occurrence count that exceeded defined run queue latency threshold.
- Interrupt execution interval – stores the occurrence count that exceeded the defined interrupt execution interval.
- System call execution interval – stores the occurrence count that exceeded the defined system call execution interval.

/proc/sys/kernel/ppa

This file defines same values as per-thread ppactl control file. However, these values are globally enforced and an additional field is defined to control CPU scaling. The feature is currently not supported. For accurate system-wide usage measurement, the first two fields in this file must be enabled because overall system-wide measurements are sum of per-thread measurements. However, this does not apply to all system-wide measurements. Values defined in this file take precedence over values defined in the per-thread ppactl file.

The additional field included is CPU scaling:

- CPU scaling – CMP and SMT may result in large count of OS-visible CPUs. Since PPA measure-

ments are per-CPU, this may result in excessive memory use. This option allows dividing the total number of OS-visible CPUs and lump accounting of CPU usage. For example, for an SMP system with 16 OS-visible CPUs, setting this value to 4 will group CPU usage of four consecutive CPUs, resulting in an array of four elements. Measurements for CPUs 0-3, 4-7, 8-11 and 12-15 will be combined in index 0, 1, 2 and 3 respectively. Setting this value to 16 (following the example)

will result in one element. This value will be used for newly created threads. Currently, this feature is not supported.

/proc/rawsysppa

This file contains system-wide measurements. For these measurements to be accurate, they must be globally enabled. As in the case of the rawppa file, all values are binary and can be accessed in the same way (i.e., open(), lseek(), pread()). Likewise, writing

Header	0	2	4 – value 4 bytes in size
	Version	CPU Count	Reserved
System Usage Time	cpu 0	cpu n-1
System Call Count	cpu 0	cpu n-1
Interrupt Usage Time	cpu 0	cpu n-1
Interrupt Count	cpu 0	cpu n-1
Idle Usage Time	cpu 0	cpu n-1
Softirq Usage Time	cpu 0	cpu n-1
Softirq Count	cpu 0	cpu n-1
Tasklet Usage Time	cpu 0	cpu n-1
Tasklet Count	cpu 0	cpu n-1
Tasklet Hi Usage Time	cpu 0	cpu n-1
Tasklet Hi Count	cpu 0	cpu n-1
Timer Usage Time	cpu 0	cpu n-1
Timer Count	cpu 0	cpu n-1
Net Tx Usage Time	cpu 0	cpu n-1
Net Tx Count	cpu 0	cpu n-1
Net Rx Usage Time	cpu 0	cpu n-1
Net rx Count	cpu 0	cpu n-1
Context Usage Time	cpu 0	cpu n-1
Context Count	cpu 0	cpu n-1
Signal Count	cpu 0	cpu n-1

8 - byte values

4 – byte values

Table 3: rawsysppa file

to the file sets all values to defaults - mostly 0. Additionally, measurements are accumulated per-CPU and scaling will not apply to system-wide statistics. By default, file permissions are 'rw-r--'. Time-based measurements are in nano-seconds. All the system-wide collected measurements are summarized below; refer to Table 3 for reference:

- Header – same as per-thread or process.
- The next two arrays store system mode execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two arrays store interrupts mode execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next array stores idle execution time. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two arrays store softirq mode execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two arrays store tasklet execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header. (mention why tasklets).
- The next two arrays store high priority tasklet execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two arrays store timer activity execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two arrays store network transmit softirq execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two vectors store network receive softirq execution time and occurrence count. Each element is 8-bytes long and dimension is of length CPU count stored in the header.
- The next two vectors store average context execution time and occurrence count - these values are periodically sampled. Each element is 8-bytes long and dimension is of length CPU count stored in the header.

- The next vector stores signal occurrence count. Each element is 4-bytes long and dimension is of length CPU count stored in the header.

User time is not accumulated, but it can be determined by subtracting system and idle mode execution from measurement interval. Interrupt time is not subtracted from system or idle mode execution. If system mode measurements are disabled, then only idle, interrupt and user mode execution is available. If PPA is disabled, only idle time is available and collection of idle time can not be disabled.

Future Direction and Long-Term Focus

Future focus is on tools which can aid developers on hardening the NEs and aid field personnel to immediately collect appropriate data and root-cause incidents. Part of the effort will focus on options that will allow users to easily read sorted output based on the various measurements PPA reports today, for example, list threads with highest scheduling delay. In addition to tools, there will be focus on graphical interpretation of measurements and ability to diff several runs, to facilitate system characterization and root-cause performance-related incidents.

In the measurement area, future direction will be to expand the framework to measure latencies of specific system calls and OS intervals of interest. Additional states for accounting are also being evaluated.

Lastly, several CGL 4.0 requirements related to PPA are being evaluated for future work, namely:

- AVL.5.1 'Kernel-Level Non-Intrusive Application Monitor Without Modifying Application Code'
- AVL.5.3 'Process-Level Non-Intrusive Application Monitor'

Derivative Works

Hardening in carrier-grade environments, not to be confused with Security Hardening, is a never ending task that spans all kernel subsystems. As new hardware is released and new features added to the OS, hardening needs to be revisited. At the top of the list, beyond CPU execution hardening, are incidents related to lack of memory hardening. A derivative of this work is to harden memory accounting and isolation between functional areas executing in carrier-grade environments.

Precise Memory Accounting

Given that carrier-grade systems stay up for months or even years at a time, accurate count of free and in-use memory is fundamental in carrier-grade environments. Without proper knowledge of system free memory (current free + reclaimable), carrier-grade systems risk five-ninths availability. Without accurate memory accounting, carrier-grade systems can't be correctly analyzed when low memory incidents occur in the field. Likewise, HA can't determine low memory conditions (i.e., failover), so there is no way to accurately debug the cause of memory exhaustion and there is no way to root-cause problems related to low memory. In addition to precise system memory usage, per-process system usage is important. Specifically, the need to disambiguate memory shared with other processes and to determine the extent to which each process contributes to the overall memory usage. Future work will focus on some of the following areas. In some cases, this may involve porting upstream contributions (for example NUMA) targeting other market segments to carrier-grade environments. Few items of interest are listed.

- a. Inaccurate free memory with Mlocked pages present.
- b. Premature exhaustion of memory and impact on system.
- c. The affect on memory over-commit policy.
- d. Process per-segment precise accounting.

Precise Memory Enforcement

Although time-share applications are not critical, the customer still requires decent response from such applications (i.e., billing). Work in this area is intended to isolate real-time from time-share applications, to start with prevent excessive Mlocked pages and enforce memory allocation per UID, GID or group of processes. The aim is to draw a line between real-time and time-share memory usage at system startup, so neither application type will interfere with the other. Some real-time applications lock excessive count of pages leading to poor execution of other key but, perhaps, not real-time processes. On the other hand, some time-share applications may allocate excessive pages requiring page reclaim and lead to missed deadlines for real-time applications. It's not always possible to pre-allocate memory for real-time applications. For example, a HA restart may require to drop all memory references and start from scratch. Currently, only per-process locked page count limit can be enforced. However, several tasks in same/different user/group may lock excessive

memory. Thrashing is more likely to occur as the pageable size of the total memory shrinks. Thrashing has serious side effects for critical real-time applications, such as unpredictable response time. Even if the task is real-time hardened, it will require implicit memory allocations in the kernel (on system calls). These allocations will stall as the OS looks for memory. By having the ability to keep an ample reserve of free memory for paging/swapping applications, both real-time and time-share apps may coexist.

Conclusions

Today, native CPU accounting and alternative solutions do not meet the necessary requirements of carrier-grade systems. Prior to deployment, NE operational characteristics are lacking and, thus, an NE may be shipped with undesired and unexpected execution behavior. After deployment, there are insufficient means to identify and root-cause field incidents and determine NE overload conditions. PPA hardens the system while in development and allows detailed insight into system behavior to root-cause field incidents. PPA has been deployed in the field for over five years by major carriers all over the world and has proven to be reliable characterizing tool, root-causing NE execution behavior. In addition, PPA has also aided carriers in capacity management in times of excessive peak use or calamities.

Appendix A – References

Linux Foundation (previously known as OSDL)
<http://www.linux-foundation.org>

SCOPE Alliance Linux Profile
<http://www.scope-alliance.org>

Linux Kernel
<http://www.kernel.org>

Precise Process Accounting
<http://sourceforge.net/projects/ppacc>

Appendix B – Glossary

Listed by appearance order:

HA	High Availability
NE	Network Element
PPA	Precise Process Accounting
CGL	Carrier Grade Linux
SMT	Simultaneous Multi-Threading
CMP	Chip Multi-Processing
NUMA	Non-Uniform Memory Access
QoS	Quality of Service
OS	Operating System
MT	Multi-Threaded
SCTP	Stream Control Transmission Protocol
Mlocked	Virtually mapped pages locked by a user process

About the Author

Mario Smarduch is the lead architect for Motorola Software Group's Embedded Systems, Open Source and Linux Technology Group. In his role, he architects open source solutions across Motorola's product line. He was also involved in defining the company's Linux requirements for vendors. Mr. Smarduch has 20 years of experience in designing, deploying and debugging network elements in wireless and telecommunication environments. He has worked with all major UNIX variants, Linux and embedded operating systems on a variety of hardware platforms. Mario joined Motorola in 1999 as the member of iDEN Common Controller Platform group where he was one of the few engineers to deploy the first Nextel systems.

Before joining Motorola, Mr. Smarduch held various positions at Compaq, Tandem, Sun Microsystems and Lucent Technologies, where he designed and debugged network equipment to resolve customer issues, OS kernel and application problems.

Currently, Mr. Smarduch is a member of the Linux Foundation Carrier Grade Linux group. He is also is the Chair of the CGL Specifications group that is focused on standardizing an application hardening guide for network equipment providers. Mr. Smarduch has been engaged in open source and Linux for over six years. He developed a unique Linux kernel feature to precisely measure CPU utilization called Precise Process Accounting. He has engaged the Linux community on several occasions to fix critical kernel problems.

Mr. Smarduch holds a bachelor's degree in electronics engineering from DeVry Institute of Technology and a master's degree in computer science from Illinois Institute of Technology.



MOTOROLA