# *Performance Optimization for the Intel Atom Architecture*

ABSTRACT: The quality of tools support has a direct impact on the effectiveness of your optimization efforts. Performance tools that target single processor core performance provide insight into the application and how the application is behaving at the level of the microarchitecture. Multi-core performance tools provide insight into how the application is executing in the context of Intel® Hyper-Threading Technology and multi-core processing. Finally, performance tools focused on power optimization provide insight into application behavior that impacts power utilization. Understanding the capabilities of tools and how to use them is critical to your Intel Atom processor migration. This article instructs on the performance optimization process for systems based on the Intel Atom processor. First, an overview of the optimization process is discussed followed by a description of the tools employed in this process.

## Optimization Process

Good software design seeks a balance between simplicity and efficiency. Performance of the application is an aspect of software design; however correctness and stability are typically prerequisite to extensive performance tuning efforts. A typical development cycle is depicted in Figure 1 and consists of four phases: design, implementation, debugging, and tuning. The development cycle is iterative and concludes when performance and stability requirements are met. Figure 1 further depicts a more detailed look inside of the tuning phase, which consists of single processor core optimization, multi-core processor optimization, and power optimization.

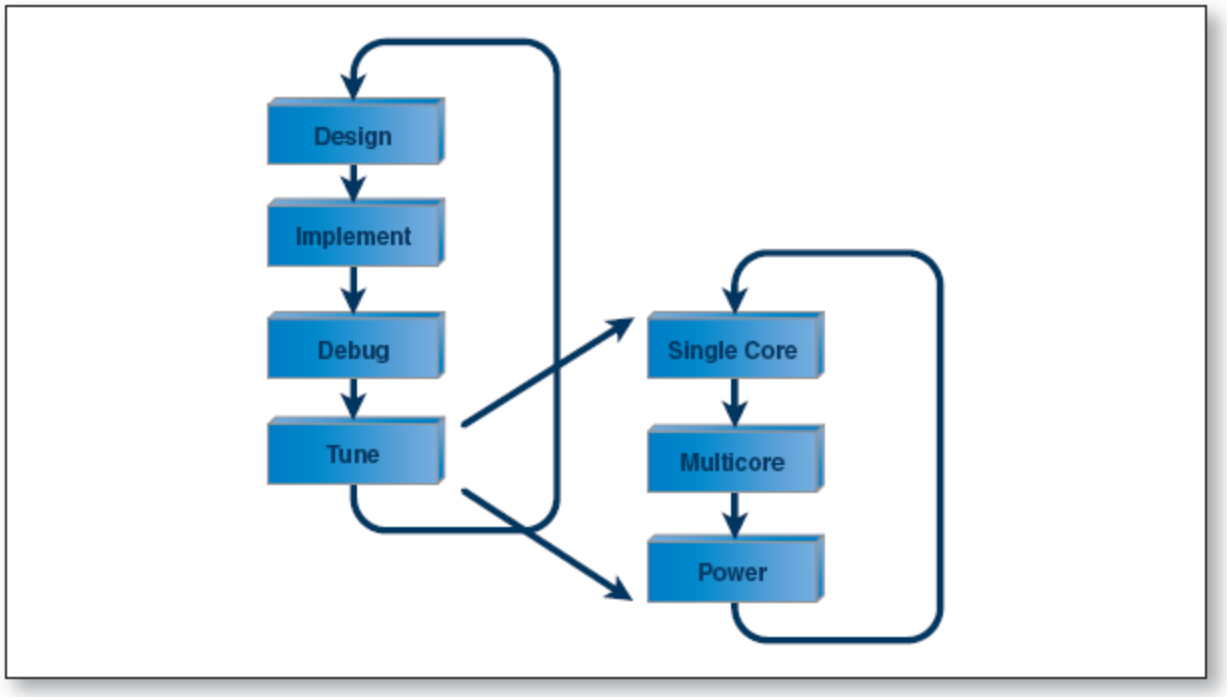


**Figure 1** Development and Optimization Process

One key fact to highlight about the optimization process is that changes made during this phase can require another round of design, implementation, and debug. It is hoped that a candidate optimization would require minimal changes, but there are no guarantees. Each proposed change required as a result of a possible optimization should be evaluated in terms of stability risk, implementation effort, and performance benefit.

Similarly, the tune step is also iterative with the goal of reaching a satisfactory equilibrium between single core, multi-core, and power performance. The components of the tune step are summarized as follows:

- *Single processor core tuning*. Optimization of the application assuming execution on one Intel Atom processor core. This step focuses on increasing performance, which is typically the reduction of execution time.
- *Multi-core tuning*. Optimization of the application taking advantage of parallel technology including Intel Hyper-Threading Technology and multiple processor cores. This step focuses on increasing performance, which is typically the reduction of execution time.
- *Power tuning*. Optimization of the application focusing on power utilization. This step focuses on reducing the amount of power used in accomplishing the same amount of work.

## Single Processor Core Tuning

Single processor core tuning focuses on improving the behavior of the application executing on one physical Intel Atom processor core. Intel Hyper-Threading Technology is not considered during this phase; it enables one physical processor core to appear as two cores and introduces issues more related to multi-core processing. This tuning step isolates the behavior of the application from more complicated interactions with other threads or processes on the system. This step is not entirely focused on what traditionally is called serial tuning because parallelism in the form of vector processing or acceleration technology can be considered.

The foundation of performance tuning is built upon complementary assertions that of the Pareto principle and Amdahl's law. The Pareto principle, colloquially known as the 80/20 rule, states that 80 percent of the time spent in an application is in 20 percent of the code. This observation helps prioritize optimization efforts to the areas of highest impact, namely the most frequently executed portions of the code. Amdahl's law provides guidance on the limits of optimization. For example, if your optimization can only be applied to 75 percent of the application, the maximum theoretical speedup is 4 times.

Single processor core tuning is itself comprised of multiple steps, which are characterized as first gaining an understanding of the application and then tuning based upon general performance analysis and tuning and then analysis and tuning specific to the Intel Atom processor. The single processor core tuning process is summarized by the following steps:

1. *Benchmark*. Develop a benchmark that represents typical application usage.
2. *Profile*. Analyze and understand the architecture of the application.
3. *Compiler optimization*. Use aggressive optimizations if possible.
4. *General microarchitecture tuning*. Tune based upon insight from general performance analysis statistics. These statistics, such as clock cycles per instruction retired, are generally accepted performance analysis statistics that can be employed regardless of the underlying architecture.
5. *Intel® Atom™ processor tuning*. Tune based on insight about known processor "glass jaws." These include statistics and techniques to isolate performance issues specific to the Intel Atom processor.

## Multi-Core Processor Tuning

The focus of multi-core processor tuning is on the effective use of parallelism that takes advantage of more than one processor core. This step pertains to

 

both Intel Hyper-Threading Technology and true multi-core processing. There are some issues specific to each; where appropriate these differences are highlighted. Second, at the application level, two techniques allow you to take advantage of multiple processor cores, multitasking and multithreading. *Multitasking* is the execution of multiple operating system processes on a system. In the context of one application, multitasking requires the division of work between distinct processes and special effort is required to share data between processes. *Multithreading* is the execution of multiple threads and by default assumes memory is shared, which introduces its own set of concerns. This article limits itself to discussion of multithreading because multitasking is a more mature technology and one where the operating system governs much of the policy of execution. Multithreading in the context of the Intel Atom processor is much more under the control of the software developer.

Developing software for multi-core processors requires good analysis and design techniques. A wealth of information on these techniques is available in literature by Mattson et al., Breshears, and many others.

Tuning of multithreaded applications on the Intel Atom processor requires ensuring good performance when the application is executing on both, logical processor cores available via Intel Hyper-Threading Technology, and multiple physical processor cores. General multithreading issues that affect performance regardless of the architecture must be addressed. These issues include for example lock contention and workload balance. One of the performance concerns when executing under Intel Hyper-Threading Technology is on the shared resources of the processor core. For example, the caches are effectively shared between two concurrently executing threads. In a worst case scenario, it is possible for one thread to cause the other to miss in the cache on every access. Tuning for multi-core processors adds another level of complication as the possible thread interactions and cache behavior can be even more complicated. It is possible for two threads to cause false sharing, which limits performance but can be easily addressed. Understanding techniques to analyze performance and how to mitigate these performance issues are essential.

Converting a serial application to take advantage of multithreading requires an approach that uses the generic development cycle, consisting of these five phases: Analysis, Design, Implementation, Debug, and Tune. There are threading tools that help with code analysis, debugging, and performance tuning.

1. *Analysis*. Develop a benchmark that represents typical system usage and comprised by concurrent execution of processes and threads. In many cases, the benchmark from the single core tuning phase and the

 

initial parallel implementation may be an appropriate starting point. Use a system performance profiler such as the Intel® VTune™ Performance Analyzer to identify the performance hotspots in the critical path. Determine if the identified computations can be executed independently. If so, proceed to the next phase; otherwise look for other opportunities with independent computations.

2. *Design*. Determine changes required to accommodate a threading paradigm (data restructuring, code restructuring) by characterizing the application threading model (data-level or task-level parallelization). Identify which variables must be shared and if the current design structure is a good candidate for sharing.
3. *Implementation*. Convert the design into code based on the selected threading model. Consider coding guidelines based on the processor architecture, such as the use of the PAUSE instruction within spinwait loops. Make use of the multithreading software development methodologies and tools.
4. *Debug*. Use runtime debugging and thread analysis tools such as Intel® Thread Checker.
5. *Tune*. Tune for concurrent execution on multiple processor cores executing without Intel Hyper-Threading Technology. Tune for concurrent execution on multiple processor cores executing with Intel Hyper-Threading Technology.

## Power Tuning

Tuning that is focused on power utilization is a relatively new addition to the optimization process for Intel architecture processors. The goal of this phase is to reduce the power utilized by the application when executing on the embedded system. One of the key methods of doing so is by helping the processor enter and stay in one of its idle states.

### *Basics on Power*

In an embedded system, power at its fundamental level is a measure of the number of watts consumed in driving the system.

Power can be consumed by several components in a system. Typically, the display and the processor are the two largest consumers of power in an embedded computing system. Other consumers of system power include the memory, hard drives, solid state drives, and communications. Power management features already exist in many operating systems and enable implementation of power policy where various components are powered down when idle for long periods. A simple example is turning off the display after a few minutes of idle activity. Power policy can also govern behavior

 

based upon available power sources. For example, the embedded system may default to a low level of display brightness when powered by battery as opposed to being plugged into an outlet.

Several statistics exist for characterizing the power used by a system including:

- *Thermal design power (TDP)*. The maximum amount of heat that a thermal solution must be able to dissipate from the processor so that the processor operates under normal operating conditions. TDP is typically measured in watts.
- *"Plug load" power*. A measure of power drawn from an outlet as the embedded system executes. Plug load power is typically measured in watts.
- *Battery power draw*. A estimate of power drawn from a battery as the embedded system executes. Typically, battery power draw is stated in watts and is based upon estimates from ACPI.

Your project requirements will guide which of these power measurements to employ and what goals will be set with regard to them.

The Intel Atom processor enables a number of power states, which are classified into C-states and P-states. C-states are different levels of processor activity and range from C0, where the processor is fully active down to C61 where the processor is completely idle and many portions of the processor are powered down. P-states, known as performance states, are different levels of processor frequency and voltage.

*Power Measurement Tools*

In order to determine if optimizations improve power utilization, a tool is required to measure power utilization. There are two categories of tools to measure power on an embedded system. The first category provides a direct measurement and employs physical probes to measure the amount of power used. These probes could be as simple as a plug load power probe between the device and the electrical outlet. They could require more extensive probes placed on the system board monitoring various power rails such as those required to execute the EEMBC Energybench2 benchmark.

The second category, power state profiling tools, employs an indirect method of measuring power utilization. Instead of directly measuring power, this class of tool measures and reports on the amount of time spent in different power states. The objective when using these tools is to understand what activities are causing the processor to enter C0 and to minimize them.

 

*Tuning Overview*

The goal of power tuning is two-fold:

- Minimize time in active state.
- Maximize time in inactive state.

On the surface it may seem like these goals are redundant; however in practice both are required. Power is expended in transitioning into and out of idle modes. A processor that is repeatedly waking up and then going back to sleep may consume more power than a processor that has longer periods in an active state. In general, the end result is for the system to be in idle mode 90 percent of the time. Of course, this end result depends on the specific workload and application. Techniques to meet this goal follow one of two tuning strategies, which are summarized as follows:

- *Race to idle*. The tasks are executed as quickly as possible to enable the system to idle. This approach typically entails aggressive performance optimization using similar techniques as single core and multi-core performance tuning.
- *Idle mode optimization*. Iteratively add software components executing on the system and analyze power state transitions to ensure these components are as nondisruptive to power utilization as possible.

High power utilization has several causes, including:

- Poor computational efficiency
- Poor memory management
- Bad timer and interrupt behavior
- Poor power awareness
- Bad multithreading behavior

## The Performance Tuning Cycle

It is important to note that the tuning process is not sequential, but iterative. It is typically not sufficient to step through the three phases only once. Changes made during the power optimization phase may require a new pass at single-core and multi-core optimization to meet performance targets. A subsequent multi-core focused optimization may place inappropriate demand on power and require further power optimization. The hope is that the changes made have less and less of an impact until an equilibrium is reached and the performance targets are met. This is when one can consider performance tuning to be complete. That said, performance regression tests should be run to ensure subsequent bug fixes and changes do not impact performance negatively.

 

# Power and Performance Analysis Tools Overview

Software tools for performance and power optimization aid in your analysis and tuning efforts. The specific tools detailed in this section are arranged according to the performance tuning phase. The information here provides further details on the capabilities and usages of the tools specific to performance optimization.

## Single Core Performance Tools

Tools for analyzing single core processor performance provide insight into how an application is behaving as it executes on one processor core. These tools provide different views on the application ranging from how the application interacts with other processes on the system down to how the application affects the processor microarchitecture. Many tools are available that provide profiling capability in different ways. Typically, they fall into one of the following categories:

- *System profilers*. Provide a summary of execution times across processes on the system.
- *Application profilers*. Provide a summary of execution times at the function level of the application.
- *Micro-architecture profilers*. Provide a summary of processor events across applications and functions executing on the system.

Two definitions relevant to profiling concern how the data is viewed. A *flat profile* is a correlation of processes and functions with the amount of time the profiler recorded in each. A flat profile does not show relationships between the processes and functions listed with any other processes or functions executing on the system. A *call graph profile* shows these relationships and contributions to the measured times between the caller functions and called functions.

Profilers obtain information by sampling or tracing the system while the application is executing. Three techniques for sampling the system are summarized as follows:

- *Operating system provided API*. Operating system provides capability to periodically sample and record information on executing processes.
- *Software instrumentation*. Application has code added to trace and record statistics.
- *Hardware performance monitoring counters*. Employed by microarchitecture profilers. Provides information on microarchitecture events such as branch mispredictions and cache misses.

Table 1 describes several tools used in single core performance analysis. These tools are not all equal. Some of the tools provide functionality that is a superset of others. For example, sysprof is capable of providing a call graph profile across all applications executing on a system; GNU gprof is not. However, gprof is available across a wide range of operating systems; sysprof is a Linux tool. An exhaustive list of profiling tools is outside the scope of this article; we merely list a few tools representative of the profiler categories above.

**Table 1** Single Core Performance Tools Examples

| Tool | Type | Description |
|---|---|---|
| Sysprof | System profiler | Easy to use, start and stop profiler. Provides system-wide flat profile and call graph information if debug information is present. |
| GNU gprof | Application profiler | Ubiquitous, widely available single application profiler. Requires recompilation to add instrumentation. Provides flat profile and call graph profile. |
| Oprofile | Microarchitecture profiler | Linux-targeted microarchitecture profiler. Enables event-based sampling using hardware performance monitoring counters. |
| Intel® VTune™ Performance Analyzer | Microarchitecture profiler | Windows† and Linux-targeted microarchitecture profiler. Enables event-based sampling using hardware performance monitoring counters. Powerful GUI enables easy visualization. |
| Intel® Performance Tuning Utility | Microarchitecture profiler | Similar to VTune Performance Analyzer with enhanced event-based profiling features. Basic block view. |

*System Profiling: Sysprof*

Sysprof is a Linux hosted and targeted system profiler that provides information across the kernel and user level processes. The tool offers a very simple user interface as depicted in Figure 2. To begin profiling, the user presses the Start button. If an application is being profiled it must be started independently of sysprof. The application itself does not require special instrumentation; however if detailed function-level information is desired then debug information should be provided. To stop profiling and show the collected results, the user clicks on the Profile button. Figure 2 displays a profile of an application viewed using sysprof. The screen is divided into three sections. The top left section labeled *Functions* is a listing of functions where the greatest amount of time was measured during profiling. Time per individual function includes the time spent in any function called as a result of the function, such as descendents in the call chain. Time is reported in two forms, self time and total time. *Self time* is the execution time inside the function and does not include called functions. *Total time* is the amount of

 

time inside the function and all of its descendents. The bottom left window is labeled *Callers* and is a list of functions that call the highlighted function in the Functions box. Time in the Callers window is relative to the highlighted function. The self time indicates how much time is spent in the caller function while the total time is the component of time spent calling the highlighted function. The sum of the total time column in the Callers box equals the total time of the highlighted function in the Functions box. The Descendents window shows a portion of the call graph of the highlighted function. To follow a path through the call graph further, click the right-facing triangle, which will show another level of the call graph. At each node of the call graph that represents a function, time is reported for it in both, self time and cumulative time. Self time has been previously described. *Cumulative time* is the time in the function and all of its descendents and is a fraction of the time spent by a caller higher in the call graph.

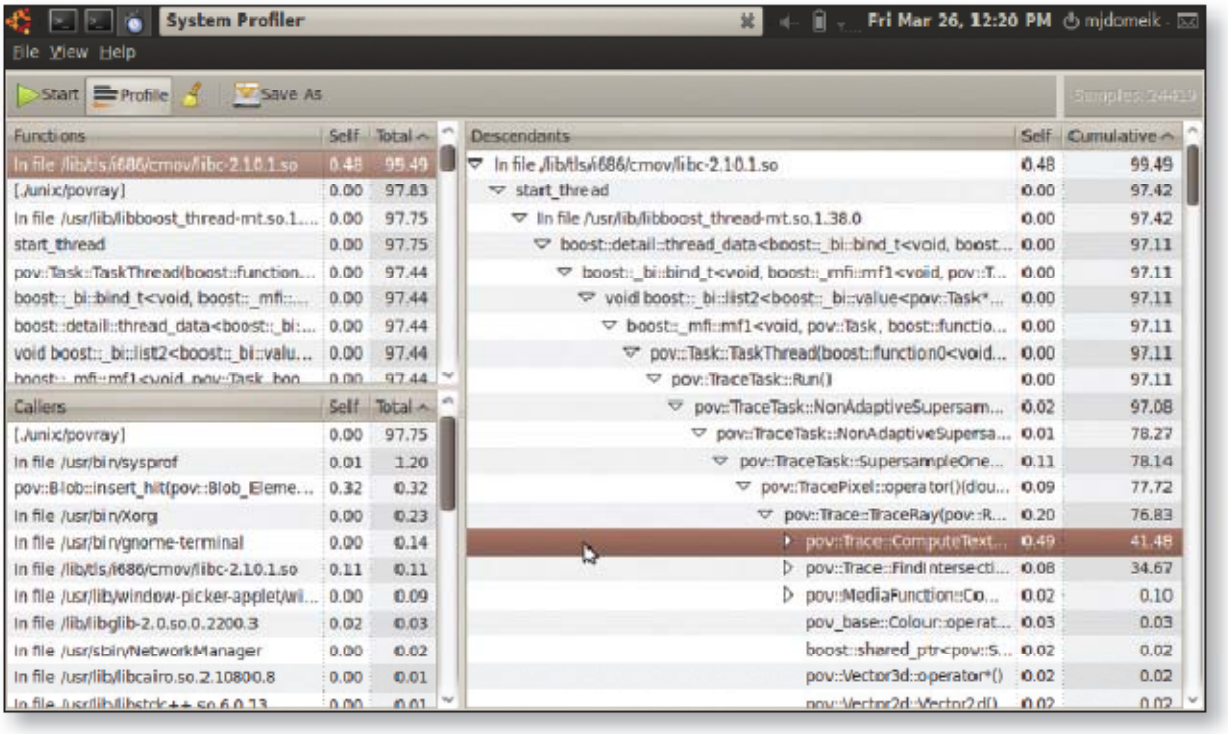


**Figure 2** Sysprof Display

A command line version of the tool is also supported. It is also possible to dump the profile results to a file for offline processing.

*Application Profiling: GNU gprof*

GNU gprof4 is an application-level profiling tool that serves as the output and reporting tool for applications that have been compiled and instrumented using the –pg option. This option is supported by GNU gcc and other compilers and results in instrumentation being added to the application to collect profile information. The instrumented application generates a profile data file (gmon. out is the default profile file name) when executed. Gprof is then employed to process the profile data and generate reports such as an ordered listing of the functions that consume the largest amount of execution time.

Figure 3 shows sample gprof profile output obtained by profiling the SPEC CPU2000 benchmark, 179.art5. The first report is a flat profile and shows a rank ordering of the various functions in the application based upon the amount of time recorded during the execution. Based upon this report, the function, *match*, had the longest amount of time spent in it, 183.25 seconds, which was 80.64 percent of the total execution time. The profile reports that the function, *match* was called 500 times. The self s/call column represents the average amount of time spent inside the function per call. The total s/call column represents the average amount of time spent inside the function and its descendents per call. For the function, *match*, these times are 0.37 seconds and 0.38 seconds respectively.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.64    183.25   183.25      500     0.37     0.38  match
 14.79    216.86    33.61      554     0.06     0.07  train_match
  1.93    221.25     4.39     4922     0.00     0.00  simtest2
  1.32    224.26     3.01     4922     0.00     0.00  reset_nodes2
  0.47    225.33     1.07      554     0.00     0.00  weightadj
  0.32    226.05     0.72     1054     0.00     0.00  reset_nodes
  0.32    226.77     0.72      554     0.00     0.00  simtest
  0.11    227.02     0.25 11080000     0.00     0.00  g
  0.07    227.17     0.15        1     0.15   191.15  scan_recognize
  0.01    227.19     0.02        3     0.01     0.01  init_bu
```

**Figure 3** Gprof Flat Profile Output

GNU gprof also provides call graph information. Figure 4 shows a portion of the call graph from the function, *match*, which shows the primary caller is

identified by index [2], *scan_recognize*. For further details on gprof, see the online documentation.

```
index % time    self  children    called     name
-----------------------------------------------
              183.25    7.75     500/500         scan_recognize [2]
[3]     84.1  183.25    7.75     500         match [3]
                4.39    0.00    4922/4922        simtest2 [5]
                3.01    0.00    4922/4922        reset_nodes2 [6]
                0.34    0.00     500/1054        reset_nodes [8]
                0.01    0.00    4422/4422        find_match [13]
                0.00    0.00      78/78          print_f12 [16]
```

**Figure 4** Gprof Call Graph Output

*Microarchitecture Profi ling: Oprofile*

Oprofile is a command line-based microarchitecture profiler providing access to the performance monitoring counters. Oprofile targets Linux systems and requires a kernel driver that acts as a daemon to collect the profile information. One of the positive aspects of the tool is that no instrumentation or recompilation of applications is required. In addition, Oprofile can profile optimized versions of applications.

The use model for Oprofile consists of configuring the daemon for profiling and instructing the daemon to begin collecting profile data. The utility, *opcontrol*, is used to issue commands to the collection daemon. The activity to monitor is then started, which typically implies user invocation of the application on a relevant benchmark. After the activity or application execution is complete, the user shuts down collection. A separate command line tool, *opreport*, is called with an option specifying the type of report desired. Other utilities are available that round out the functionality. The command line utilities that comprise oprofile and a description of each follows:

- *opcontrol*. Confi gures the collector, initiates and terminates collection.
- *opreport*. Displays profile in human readable form, merging available symbolic information where possible.
- *opannotate*. Displays profile information correlated with source and assembly code.
- *oparchive*. Saves profile for offline viewing and analysis.
- *opgprof*. Translates profile into gprof-compatible file.

Table 2 summarizes the steps for employing oprofile to collect and output a profile of an application reporting clock cycle information. Each step is

 

described followed by the command line to perform the action. These commands should be executed with root privileges.

**Table 2** Oprofile Profile Generation

| Step | Command line or Description |
|---|---|
| 1. Initialize the oprofile daemon | opcontrol –init |
| 2. Configure profile collection | opcontrol –setup –event=”default” |
| 3. Start profile collection | opcontrol –start |
| 4. Start activity | Begin activity to profile. |
| 5. Stop profile collection | opcontrol –stop |
| 6. Produce report | opreport –g –symbols |

Figure 5 shows the output of oprofile after collecting a profile of the 179. art application. The application was generated with debug information, which enables function level reporting as evidenced by line number of symbol names provided for the a.out application. The largest percentage of time, 44.2886 percent, was in the kernel (no-vmlinux). Using oprofile, it is possible to turn off collection of events from the kernel. The second through fifth highest ranked functions are inside of the 179.art application.

```
CPU: Intel Atom, speed 1000 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of
0x00 (core_p Core cycles when core is not halted) count 100000
samples  %        linenr info                 image name    app name       symbol name
957979   44.2886  (no location information)   no-vmlinux    no-vmlinux     /no-vmlinux
691019   31.9467  scanner.c:388               a.out         a.out          train_match
356627   16.4873  scanner.c:525               a.out         a.out          match
21716     1.0040  scanner.c:168               a.out         a.out          weightadj
15813     0.7311  scanner.c:90                a.out         a.out          simtest
```

**Figure 5** Oprofile Sample Output

Profile information can be collected based upon other processor events as well. For a complete list of events supported by oprofile on your particular target, use the –list-events option.

*Microarchitecture Profiling: Intel® VTune™ Performance Analyzer*

On desktop operating systems, the Intel VTune Performance Analyzer can create flat profiles, application call graph profiles, and microarchitecture profiles. The Intel® Application Software Development Tool Suite for Intel Atom Processor includes the VTune analyzer and the VTune analyzer Sampling Collector (SEP), a target-side profile collector for the Intel Atom processor. For embedded form factors that take advantage of Linux, SEP provides microarchitecture profiling capability. Using SEP requires installation of a kernel daemon that is specific to the particular Linux kernel

 

employed. The source code to the daemon can be built to enable collection on specific Linux kernels. The process of using SEP is similar to oprofile. Facilities for configuring collection, starting, and stopping are provided. Once complete, the profile is then transferred to a host environment for visualization inside of the VTune analyzer GUI.

Table 3 describes the steps and command lines employed to configure and collect a profile.

**Table 3** SEP Profile Generation Steps

| **Step** | **Command line or Description** |
|---|---|
| 1. Initialize the vtune_drv daemon | /opt/intel/vtune/vdk/insmod-vtune |
| 2. Configure and start profile collection. | sep –start -nb –d 0 |
| 3. Start activity | Begin activity to profile. |
| 4. Stop profile collection | sep –stop |
| 5. Produce report | Transfer profile data file to host environment for viewing. |

The SEP data collector supports additional options to further configure collection including:

- *Sampling*. Specify duration, interval between samples, sample buffer size, and maximum samples to count.
- *Application*. Specify an application to launch and profile.
- *Events*. Configure events and event masks. Use –event-list for a list of supported options.
- *Continuous profiling*. Aggregates data by instruction pointer, reducing space and enabling monitoring and output during execution.
- *Event multiplexing*. Enables collection of multiple events concurrently by modulating the specific event being measured while the application is profiled.

Figure 6 shows a flat profile of the 179.art application collected using SEP and transferred to a host system for analysis under the VTune analyzer GUI. The highlighted ratio in the top right shows the measurement for clocks cycles per instruction retired.

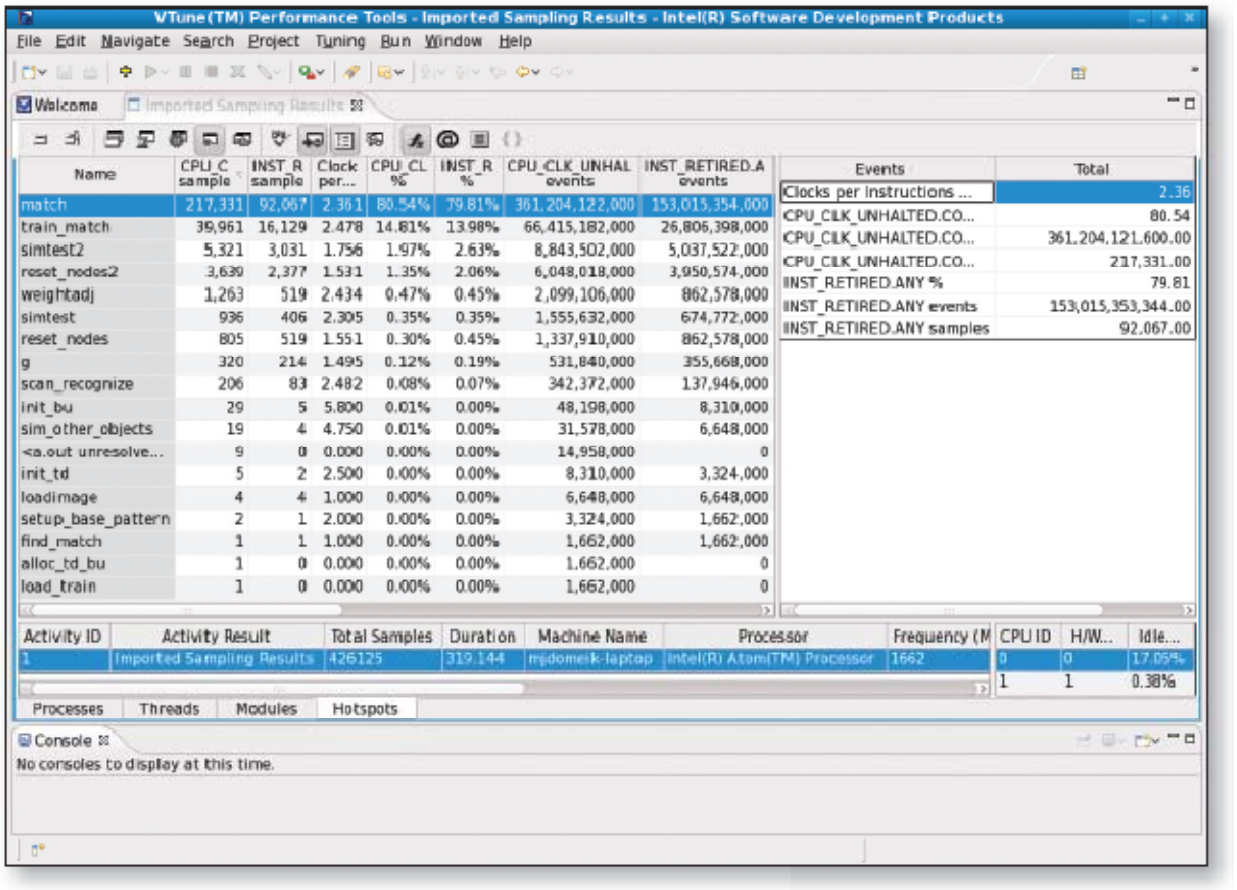


**Figure 6** VTune™ Analyzer Flat Profile View

*Microarchitecture Profiling: Event-based Sampling*

One issue with performance monitoring collection is that access to the performance counters requires kernel, or ring 0, access. Event-based sampling functions by setting up a performance monitoring counter to overflow periodically and then recording the instruction pointer location with the particular event. During profiling and as these events are recorded, a correlation of the number of events to instruction pointers is created. Implementing event-based sampling requires an interrupt handler to record these performance monitoring counter overflows and a driver that writes the counts to a file after collection is complete. The VTune analyzer includes its driver source code, which can be used as a model for other operating systems. In addition, a TBRW utility is included that enables a performance monitoring driver to read and write the VTune analyzer's data format, tb5. This enables other performance monitoring utilities to take advantage of the GUI provided by VTune analyzer.

*Microarchitecture Profiling: Intel® Performance Tuning Utility*

For more advanced microarchitecture profiling, the Intel® Performance Tuning Utility (Intel® PTU) leverages the same technology as the Intel VTune analyzer and offers sophisticated views of performance events. This tool is available on the whatif.intel.com site, which means it is an experimental tool. Some of the capabilities of Intel PTU include:

- *Basic block analysis*. Creates and displays a control flow graph and hotspots corresponding to basic blocks in the graph.
- *Events over IP graph*. Generates a histogram of performance events distributed over application code.
- *Loop analysis*. Identifies loops and recursion in the application to aid optimization.
- *Result difference*. Compares the results of multiple runs to measure changes in performance
- *Data access profiling*. Identifies memory hotspots and relates them to code hotspots.

Intel PTU is integrated into Eclipse†, which places requirements on the system under test to be able to execute the Eclipse environment. Figure 7 shows a screenshot of the basic block analysis feature of Intel PTU.

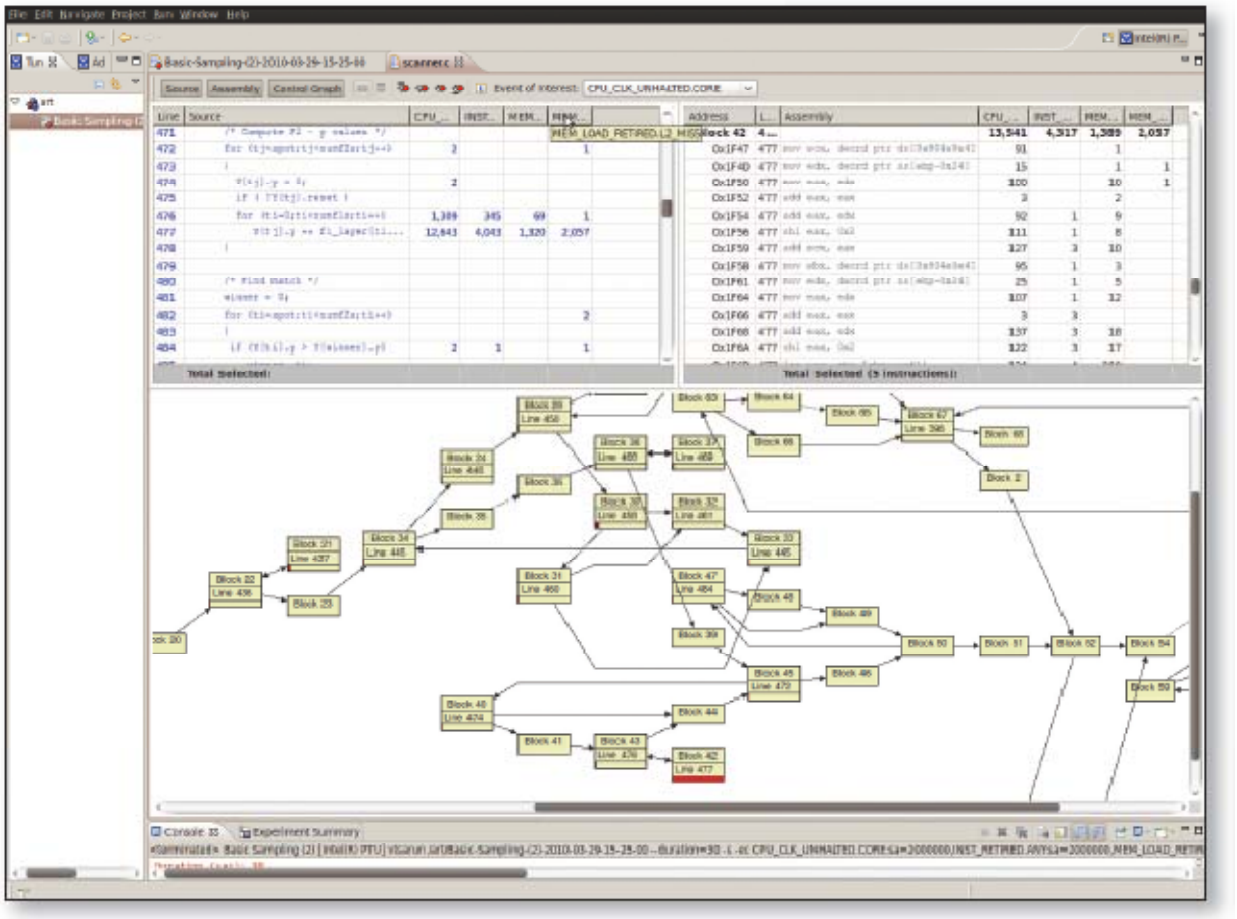

**Figure 7** Intel® PTU Basic Block View

## Multi-Core Performance Tools

Unique tools for analyzing performance related to multi-core processors are still somewhat few in number. System profilers can provide information on processes executing on a system; however interactions in terms of messaging and coordination between processes are not visible. Tools that offer visibility into this coordination typically must be cognizant of the particular API in use. POSIX Threads is a commonly employed multi-core programming API and therefore has relatively broad tools support.

### *Intel® Thread Profiler*

The Intel® Thread Profiler identifies thread-related performance issues and is capable of analyzing OpenMP†, POSIX, and Windows† multithreaded applications. When used to profile an application, some of the key capabilities include:

- The display of a histogram of aggregate data on time spent in serial or parallel regions.
- The display of a histogram of time spent accessing locks, in critical regions, or with threads waiting at implicit barriers for other threads.

Intel Thread Profiler employs what is termed *critical path analysis* where events are recorded including spawning new threads, joining terminated threads, holding synchronization objects, waiting for synchronization objects to be released, and waiting for external events. An execution flow is created that is the execution through an application by a thread, and each of the listed events above can split or terminate the fl ow. The critical path is defined as the longest flow through the execution from the start of the application until it terminates. The critical path is important because any improvement in threaded performance along this path would increase overall performance of the application.

Data recorded along the critical path includes the number of threads that are active and thread interactions over synchronization objects. Figure 8 depicts the Intel Thread Profiler GUI divided into two sections: Profile View and Timeline View. On top is the Profile View, which gives a histogram representation of data taken from the critical path and can be organized with different filters that include the following:

- Number of active threads on the critical path.
- Object view: identifies the synchronization objects encountered by threads.
- Thread view: shows the contribution of each thread to the critical path.

Benefits of these filters and views include:

- Knowledge of the amount of parallelism available during the application execution.
- Helping locate load imbalances between threads.
- Determining what synchronization objects were responsible for the most contention between threads.

The Timeline View shows the critical path over the time that the application has run. The critical path travels from one thread to another and shows the amount of time threads spend executing or waiting for a synchronization object.

 

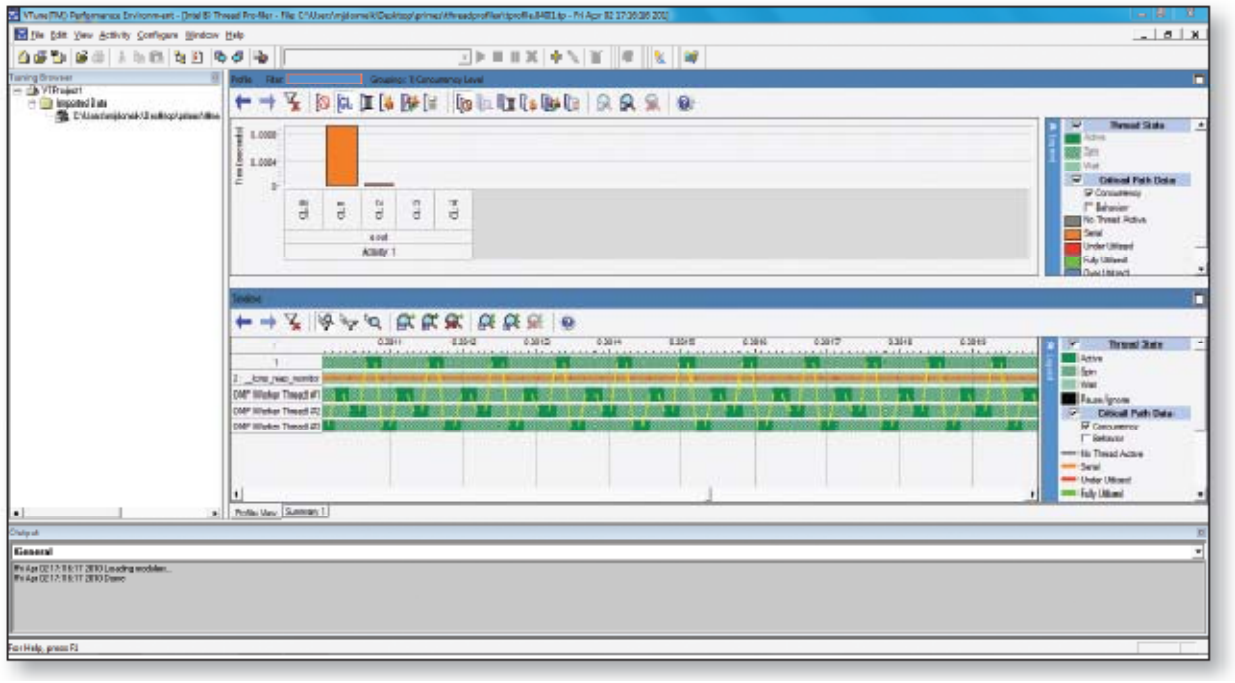

**Figure 8** Concurrency Level and Timeline View

*CriticalBlue Prism†*

CriticalBlue Prism† is another example of a toolsuite aimed at optimized software development for multi-core and/or multithreaded architectures. Prism can be used across the full range of activities needed to migrate existing sequential single core software onto a multi-core platform.

CriticalBlue Prism's what-if scheduling can be used to explore the benefit of Intel® Hyper-Threading Technology on multi-core execution performance.

Prism's analyses are based on a dynamic tracing approach. Traces of the user's software application are extracted either from a simulator of the underlying processor core or via an instrumentation approach where the application is dynamically instrumented to produce the required data. Once a trace has been loaded into Prism the user can start to analyze the application behavior in a multi-core context. In addition to standard profiling data showing functions and their relative execution times, Prism provides the user with specific insight relevant in a multi-core processor context. Examples of the views and analyses available in Prism are:

- Histogram showing activity over time by individual function and memory.
- Dynamic call graph showing function inter-relationships and frequency.
- Data dependency analysis between functions on sequential code.

 

- What-if scheduling to explore the impact of executing functions in separate threads.
- What-if scheduling to explore the impact of varying the numbers of processor cores employed.
- What-if scheduling to explore the impact of removing identified data dependencies.
- What-if scheduling to explore the impact of cache misses on multi-core execution performance.
- What-if scheduling to explore the benefit of Intel Hyper-Threading Technology on multi-core execution performance.
- Data race analysis between functions on multithreaded code.

Figure 9 is a screen shot of Prism analyzing sequential code where the user has forced several functions to execute in their own threads and a trial schedule has been generated on 4 cores. This trial schedule was modeled on unchanged sequential code and enables the user to exhaustively test and optimize the parallelization code prior to making code changes. For more information on Prism, see www.criticalblue.com.

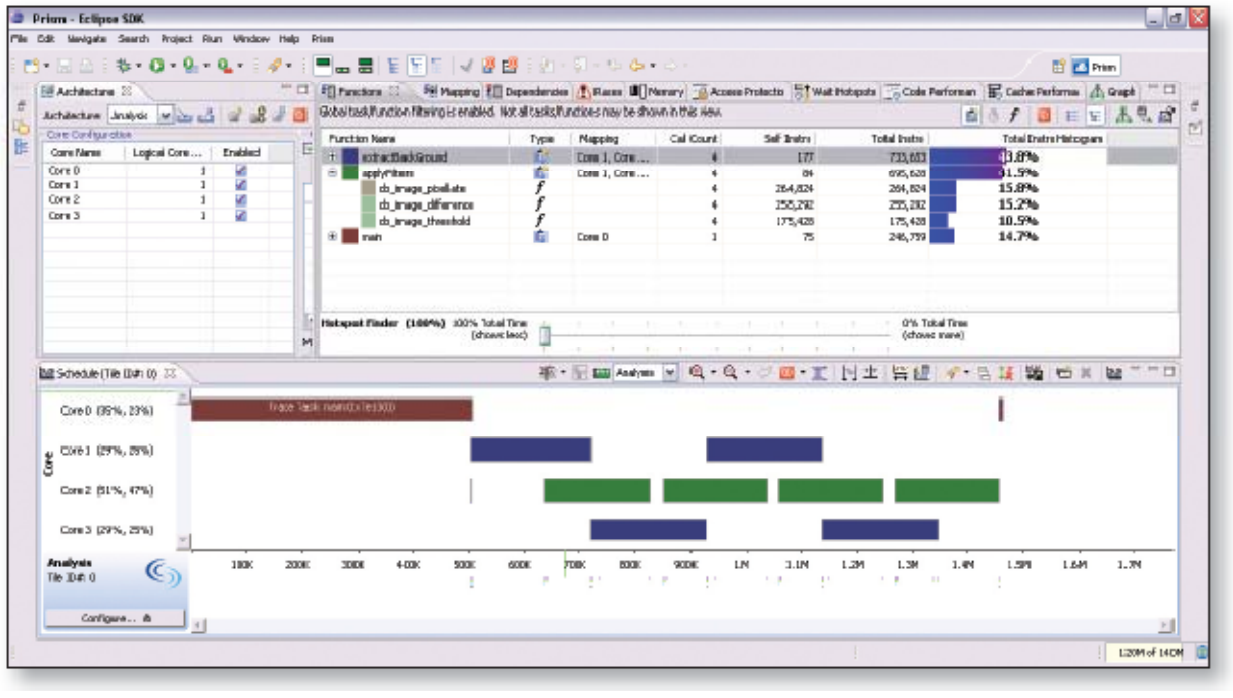


**Figure 9** CriticalBlue Prism What-if Exploration Running on Sequential Code

## Power Performance Tools

As previously mentioned, the “race to idle” power optimization strategy is implemented by employing the single-core and multi-core performance tools mentioned previously. The focus of this section is on tools to assist with idle mode optimization.

Two types of tools assess power performance. The first type of tool measures the actual power used by the device via physical probes, a technique referred to as *probe-based profiling*. The second type of tool employs counters in the platform that measure power state transitions, a technique referred to as *power state-based profiling*. For the sake of completeness a brief description of each type of tool follows; however only power state-based profiling is discussed at length and employed in the case study.

Probe-based profiling employs an external device to measure and record the power utilized by the system as a specific application executes. Typically, there is some mechanism to correlate the power readings with points in the application. An industry example of such a tool is the TMS320C55x† Power Optimization DSP Starter Kit, which integrates National Instruments Power Analyzer to provide a graphical view of power utilization over time. The Intel Energy Checker SDK6 is another probe-based profiling tool that targets desktop and server platforms. This tool measures power from the AC adaptor using a measurement tool such as those available from Watts up? and enables correlation with specific regions of application code. The data transfer assumes a shared file system, which currently limits applicability to desktop and server computing platforms.

Power-state profiling tools rely upon software interfaces into the platform’s power states, which, instead of providing a measure of power utilization, provide the number of times transitions occur between the platform power states. The process of idle-mode optimization works by enabling increasing application functionality and inspecting the recorded power data at every stage. In many cases, additional power state transitions will be recorded. Many of these additional transitions are necessary because as more functionality of the application is enabled, more processing is required. However, at each step, power state differences should be measured, understood, and optimized away if truly unneeded.

PowerTOP is a Linux targeted tool that performs power state profiling and targets idle mode optimization techniques. The tool executes on the target device with an operating mode similar to the common Unix† tool, *top*, where the tool provides a dashboard-like display. The intent is that the display

 

would provide real-time updates as your applications execute on the target device. Figure 10 displays a screenshot of PowerTOP and highlights its functionality. The tool provides six categories of information, which are summarized as follows:

- *C state residency information*. The average amount of time spent in each C state and the average duration that is spent in each C state.
- *P state residency information*. The percentage of time the processor is in a particular P state.
- *Wakeups per second*. The number of times per second the system moves out of an idle C state.
- *Power usage*. An estimate of the power currently consumed and the amount of battery life remaining.
- *Top causes for wakeups*. A rank ordered list of interrupts, processes, and functions causing the system to transition to C0.
- *Wizard mode*. Suggestions for changes to the operating system that could reduce power utilization.

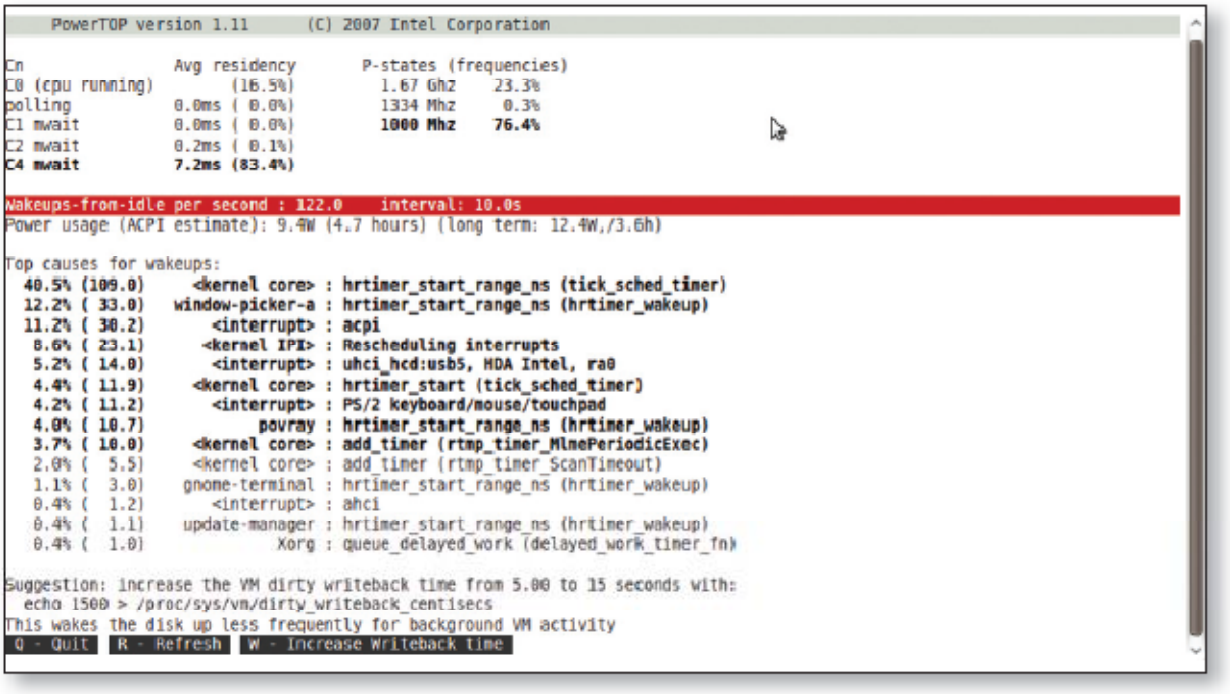


**Figure 10** PowerTOP

For more information about performance optimization and architecture options, please refer to the book *Break Away with Intel® Atom™*

 

*Processors: A Guide to Architecture Migration* by Lori Matassa and Max Domeika.

## About the Authors

**Lori Matassa** is a Sr. Staff Platform Software Architect in Intel's Embedded and Communications Division and holds a BS in Information Technology. She has over 25 years experience as an embedded software engineer developing software for platforms including mainframe and midrange computer system peripherals, as well as security, storage, and embedded communication devices. In recent years at Intel she has contributed to driver hardening standards for Carrier Grade Linux, and has led the software enablement of multi-core adoption and architecture migration for embedded and communication applications. Lori is a key contributor to Intel's Embedded Design Center, with numerous whitepapers, blogs, and industry contributions on a variety of topics critical to embedded migration.

**Max Domeika** is an embedded software technologist in the Developer Products Division at Intel, creating tools targeting the Intel architecture market. Over the past 14 years, Max has held several positions at Intel in compiler development, which include project lead for the C++ front end and developer on the optimizer and IA-32 code generator. Max currently provides embedded tools consulting for customers migrating to Intel architecture. In addition, he sets strategy and product plans for future embedded tools. Max earned a BS in Computer Science from the University of Puget Sound, an MS in Computer Science from Clemson University, and a MS in Management in Science & Technology from Oregon Graduate Institute. Max is the author of *Software Development for Embedded Multi-core Systems* from Elsevier. In 2008, Max was awarded an Intel Achievement Award for innovative compiler technology that aids in architecture migrations.



This article is based on material found in book *Break Away with Intel® Atom™ Processors: A Guide to Architecture Migration* by Lori Matassa and Max Domeika. Visit the Intel Press web site to learn more about this book: http://www.intel.com/intelpress/sum_ms2a.htm