acmqueue

# Power-Efficient Software

**Power-manageable hardware can help save energy, but what can software developers do to address the problem?**

Eric Saxe, Sun Microsystems

The rate at which power-management features have evolved is nothing short of amazing. Today almost every size and class of computer system, from the smallest sensors and handheld devices to the "big iron" servers in data centers, offers a myriad of features for reducing, metering, and capping power consumption. Without these features, fan noise would dominate the office ambience, and untethered laptops would remain usable for only a few short hours (and then only if one could handle the heat), while data-center power and cooling costs and capacity would become unmanageable.

As much as we might think of power-management features as being synonymous with hardware, software's role in the efficiency of the overall system has become undeniable. Although the notion of "software power efficiency" may seem justifiably strange (as software doesn't directly consume power), the salient part is really the way in which software interacts with power-consuming system resources.

Let's begin by classifying software into two familiar ecosystem roles: resource managers (producers) and resource requesters (consumers). Then we examine how each can contribute to (or undermine) overall system efficiency.

## POWER-EFFICIENT RESOURCE MANAGEMENT

The history of power management is rooted in the small systems and mobile space. By today's standards, these systems were relatively simple, possessing a small number of components, such as a single-core CPU and perhaps a disk that could be spun down. Because these systems had few resources, utilization in practice was fairly binary in nature, with the system's resources either being in use—or not. As such, the strategy for power managing resources could also be fairly simple, yet effective.

For example, a daemon might periodically monitor system utilization and, after the system appeared sufficiently idle for some time threshold, clock down the CPU's frequency and spin down the disk. This could all be done in a way that required little or no integration with the subsystems otherwise responsible for resource management (e.g., the scheduler, file system, etc.), because at zero utilization, not much resource management needed to be done.

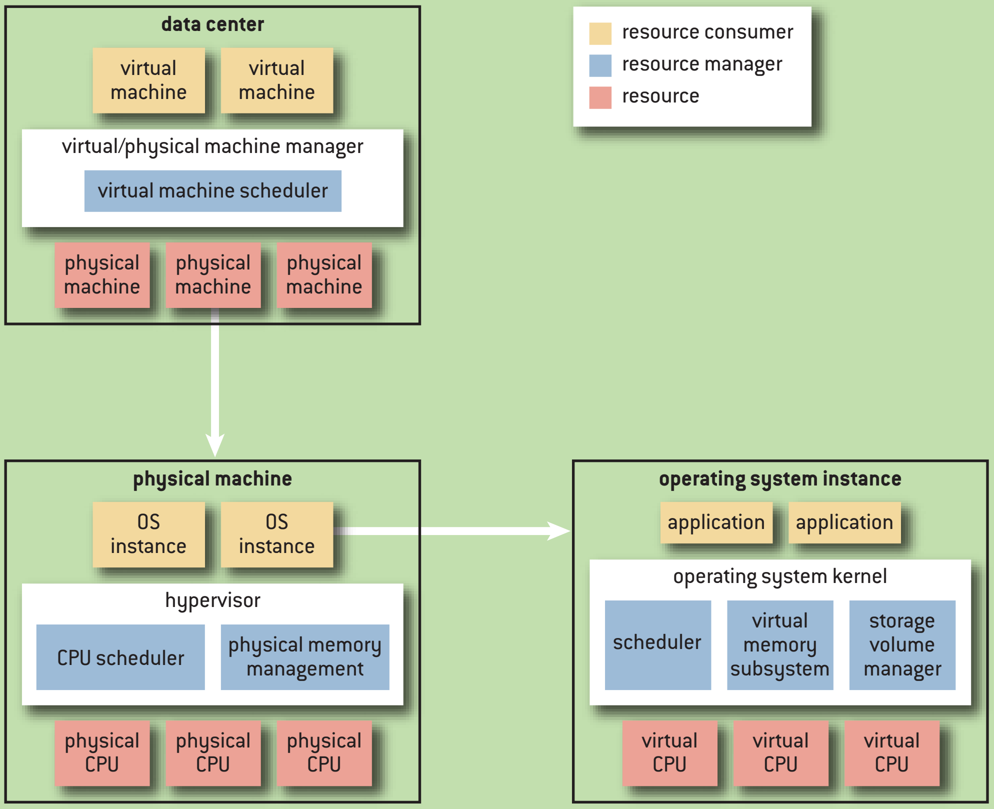
By comparison, the topology of modern systems is far more complex. As the "free performance lunch" of ever-increasing CPU clock speeds has come to an end, the multicore revolution is upon us, and as a consequence, even the smallest portable devices present multiple logical CPUs that need to be managed. As these systems scale larger (presenting more power-manageable resources), partial utilization becomes more common, where only part of the system is busy while the rest is idle. Of course, CPUs present just one example of a power-manageable system resource: portions of physical memory may (soon) be power manageable, with the same being true for storage and I/O devices. In the larger data-center context, the system itself might be the power-manageable resource.

Effective resource management on modern systems requires that there be at least some level of resource manager awareness of the heterogeneity brought on by varying resource power states and, if possible, some exploitation of it. (Actually, effective resource management requires awareness of resource heterogeneity in general, with varying power states being one way in which that resource heterogeneity can arise.) Depending on what is being managed, the considerations could be spatial, temporal, or both.

## SPATIAL CONSIDERATIONS

Spatial considerations involve deciding which resources to provision in response to a consumer's request in time. For an operating-system thread scheduler/dispatcher, this might determine to which CPUs runnable threads are dispatched, as well as the overall optimal distribution pattern of threads across the system's physical processor(s) to meet some policy objective (performance, power efficiency, etc.). For the virtual memory subsystem, the same would be true for how physical

FIGURE 1 **A Hierarchy of Resource Managers**



memory is used; for a file system/volume manager, the block allocation strategy across disks; and in a data center, how virtual machines are placed across physical systems. These different types of resource managers are shown in figure 1.

One such spatial consideration is the current power state of available resources. In some sense, a resource's power states can be said to represent a set of tradeoffs. Some states provide a mechanism allowing the system to trade off performance for power efficiency (CPU frequency scaling is one example), while others might offer (for idle resources) a tradeoff of reduced power consumption versus increased recovery latency (e.g., as with the ACPI C-states). As such, the act of a resource manager selecting one resource over another (based on power states) is an important vehicle for making such tradeoffs that ideally should complement the power-management strategy for individual resources.

The granularity with which resources can be power managed is another important spatial consideration. If multicore processors can be power managed only at the socket level, then there's good motivation to consolidate system load on as few sockets as possible. Consolidation drives up utilization across some resources, while quiescing others. This enables the quiesced resources to be power managed while "directing" power (and performance) to the utilized portion of the system.

Another factor that may play into individual resource selection and utilization distribution decisions is the characteristics of the workload(s) using the resources. This may dictate, for example, how aggressively a resource manager can consolidate utilization across the system without negatively impacting performance (as a result of resource contention) or to what extent changing a utilized resource's power state will impact the consumer's performance.

### TEMPORAL CONSIDERATIONS

Some resource managers may also allocate resources in time, as well as (or rather than) space. For example, a timer subsystem might allow clients to schedule some processing at some point (or with some interval) in the future, or a task queue subsystem might provide a means for asynchronous or deferred execution. The interfaces to such subsystems have traditionally been very narrow and prescriptive, leaving little room for temporal optimization. One solution is to provide interfaces to clients that are more "descriptive" in nature. For example, rather than providing a narrow interface for precise specification of what should happen and when:

```
int	schedule_timer((void)*what(), time_t when);
```

a timer interface might instead specify what needs to be done along with a description of the constraints for when it needs to happen:

```
int	schedule_timer((void)*what(), time_t about_when,
		time_t deferrable_by, time_t advanceable_by);
```

Analogous to consolidating load onto fewer sockets to improve spatial resource quiescence, providing some temporal latitude allows the timer subsystem to consolidate and batch process expirations. So rather than waking up a CPU *n* times over a given time interval to process *n* timers (incurring some overhead with each wakeup), the timer subsystem could wake the CPU once and batch process all the timers allowable per the (more relaxed) constraints, thus reducing CPU overhead time and increasing power-managed state residency (see figure 2).

## EFFICIENT RESOURCE CONSUMPTION

Clearly, resource managers can contribute much to the overall efficiency of the system, but ultimately they are forced to work within the constraints and requests put forth by the system's resource consumers. Where the constraints are excessive and resources are over-allocated or not efficiently used, the benefits of even the most sophisticated power-management features can be for naught while the efficiency of the entire system stack is compromised.

Well-designed, efficient software is a thing of beauty showing good proportionality between utilization (and useful work done) and the amount of resources consumed. For utopian software, such proportionality would be perfect, demonstrating that when no work is done, zero resources are used; and as resource utilization scales higher, the amount of work done scales similarly (see figure 3).

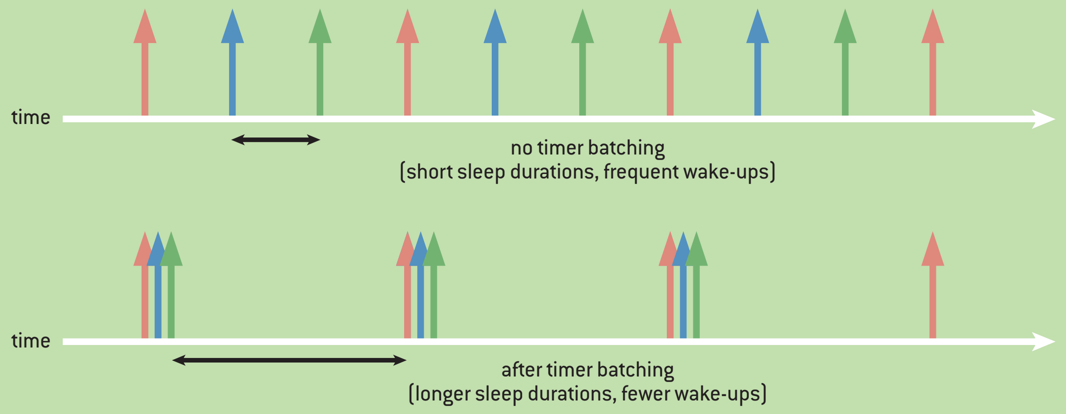
Real software is not utopian, though, and the only way to have software consume zero resources is not to run it at all. Even running very well-behaved software at the minimum will, in practice, require some resource overhead.

By contrast, inefficient software demonstrates poor proportionality between resource utilization and amount of work done. Here are some common examples:

- A process is waiting for something, such as the satisfying of a condition, and is using a timer to periodically wake up to check if the condition has been satisfied. No useful work is being done as it waits, but each time it wakes up to check, the CPU is forced to leave an idle power-managed state. What's worse, the process has decided to wake up with high frequency to "minimize latency" (see figure 4).
- An application uses multiple threads to improve concurrency and scale throughput. It blindly creates as many threads as there are CPUs on the system, even though because of an internal

FIGURE 2

**Benefits of Batch Processing Periodic Timers**


time
no timer batching
(short sleep durations, frequent wake-ups)
time
after timer batching
(longer sleep durations, fewer wake-ups)

bottleneck, the application is unable to scale beyond a handful of threads. Having more threads means more CPUs must be awakened to run them, despite little to no marginal contribution to performance with each additional thread (see figure 5).

FIGURE 3

**Ideal Efficiency**

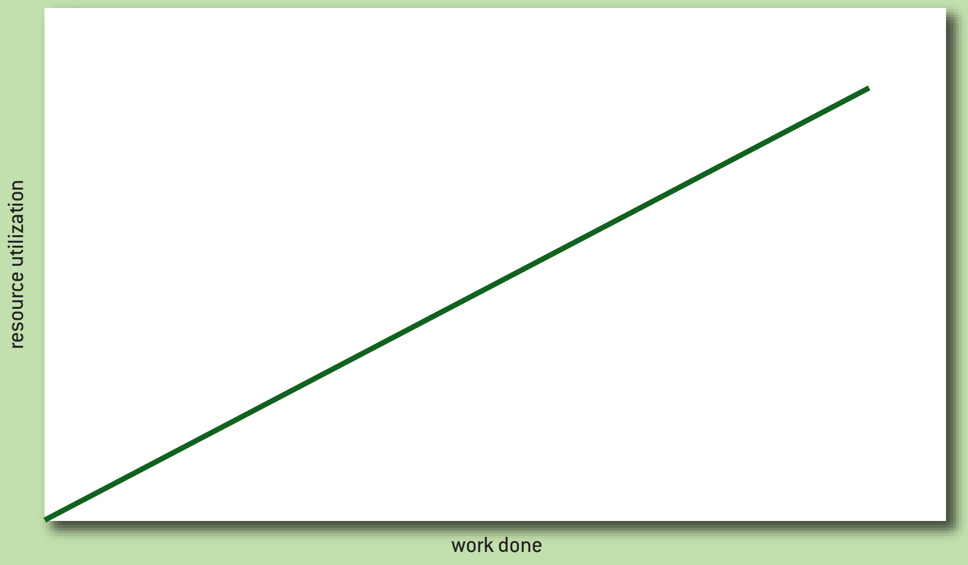
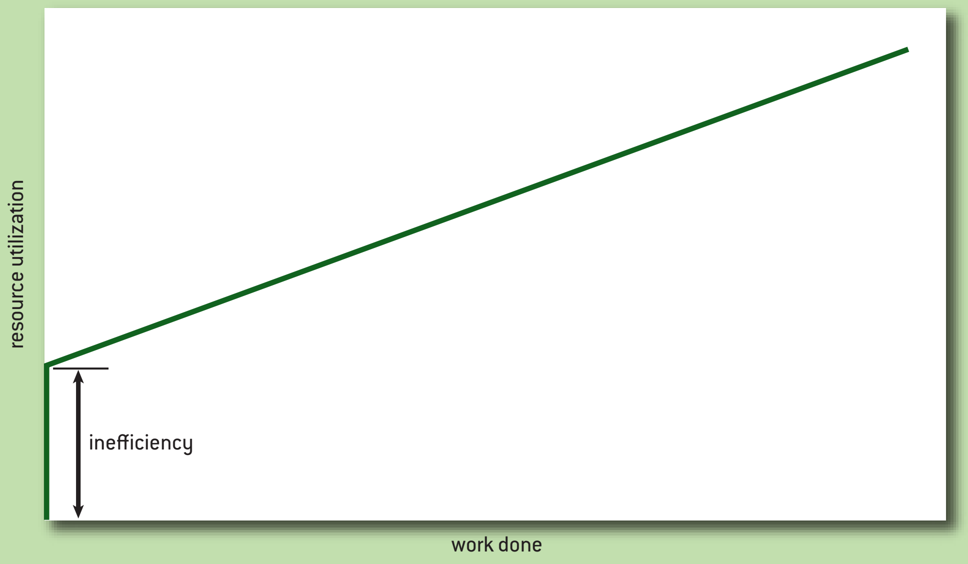


FIGURE 4

**"Idle" Inefficiency**



- A service slowly leaks memory, and over time its heap grows to consume much of the system's physical memory, despite little to none of it actually being needed. As a consequence, little opportunity exists to power manage memory since most of it has been allocated.

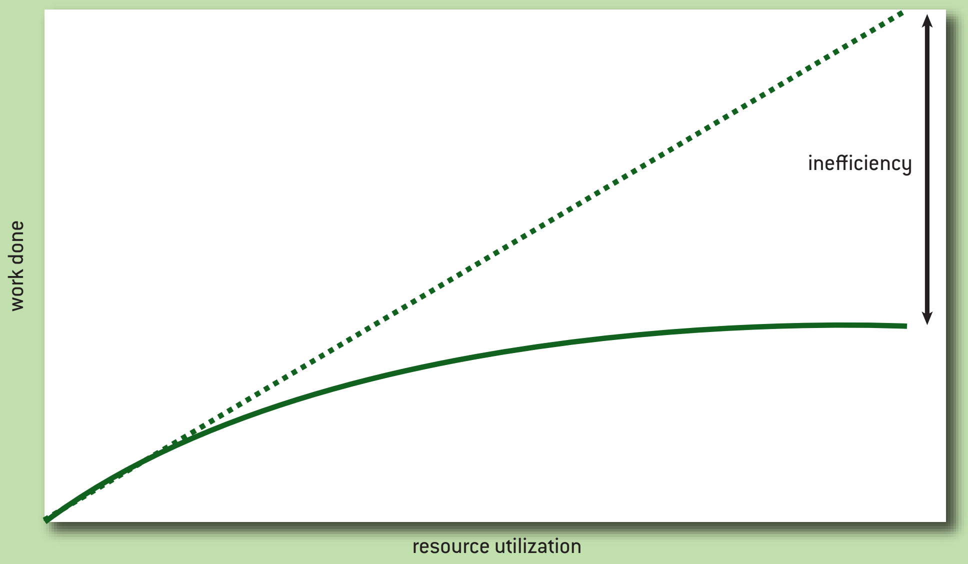
## OBSERVING INEFFICIENCY IN THE SOFTWARE ECOSYSTEM

Comprehensive analysis of software efficiency requires the ability to observe the proportionality of resource utilization versus useful work performed. Of course, the metric for "work done" is inherently workload specific. Some workloads (such as Web servers and databases) might be throughput based. For such workloads, one technique could be to plot throughput (e.g., transactions per second) versus {cpu|memory|bandwidth|storage} resource consumption. Where a "knee" in the curve exists (resource utilization rises, yet throughput does not), there is an opportunity either to use fewer resources to do the same work or perhaps to eliminate a bottleneck to facilitate doing more work using the same resources.

For parallel computation workloads that use concurrency to speed up processing, one could plot elapsed computation time versus the resources consumed, and using a similar technique identify and avoid the point of diminishing returns.

Rather than using workload-specific analysis, another fruitful technique is looking at systemwide resource utilization behavior at what should be zero utilization (system idle). By definition, the system isn't doing anything useful, so any software that is actively consuming CPU cycles is instantly suspect. PowerTOP is an open source utility developed by Intel specifically to support this methodology of analysis (see figure 6). Running the tool on what should be an otherwise idle system, one would expect ideally that the system's processors are power managed 100 percent of

FIGURE 5 **Scaling Inefficiency**



the time, but in practice, inefficient software (usually doing periodic time-based polling) will keep CPUs fractionally busy. PowerTOP shows the extent of the waste, while also showing which software is responsible. System users can then report the observed waste as bugs and/or elect to run more efficient software.

## DESIGNING EFFICIENT SOFTWARE

Efficiency as a design and optimization point for software might at first seem a bit foreign, so let's compare it with some others that are arguably more established: performance and scalability.

- Well-performing software maximizes the amount of useful work done (or minimizes the time taken to do it), given a fixed set of resources.
- Scalable software will demonstrate that performance proportionally increases as more resources are used.

Efficient software can be said to be both well performing and scalable, but with some additional constraints around resource utilization.

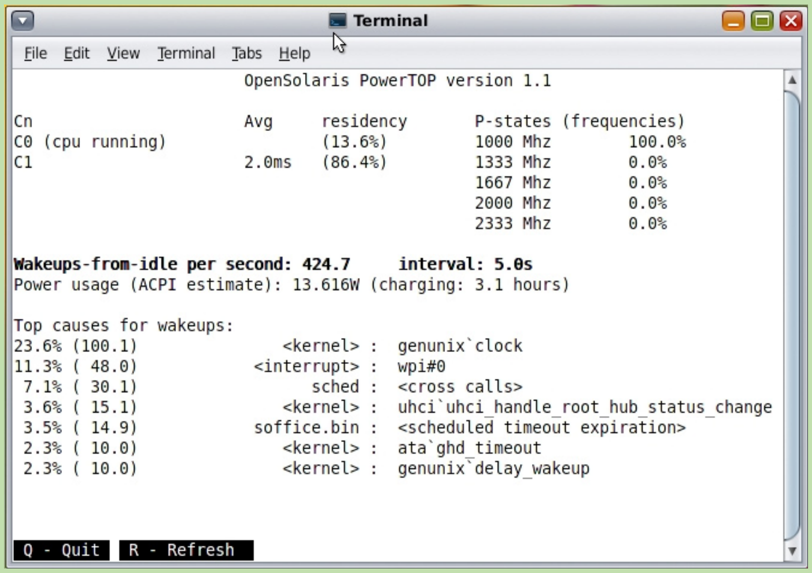
- Given a fixed level of performance (amount of useful work done or amount of time taken to do it), software uses the minimal set of resources required.
- As performance decreases, resource utilization proportionally decreases.

This implies that in addition to looking at the quantity and proportionality of performance *given the resource utilization,* to capture efficiency, software designers also need to consider the quantity and proportionality of resource utilization *given the performance*. If all this seems too abstract, here are some more concrete factors to keep in mind:

- When designing software that will be procuring its own resources, ensure it understands what resources are required to get the job done and yields them back when not needed to facilitate idle resource power management. If the procured resources will be needed on an intermittent basis,

FIGURE 6

**PowerTOP**



```
                        OpenSolaris PowerTOP version 1.1

Cn                      Avg     residency       P-states (frequencies)
C0 (cpu running)                (13.6%)         1000 Mhz        100.0%
C1                      2.0ms   (86.4%)         1333 Mhz        0.0%
                                                1667 Mhz        0.0%
                                                2000 Mhz        0.0%
                                                2333 Mhz        0.0%

Wakeups-from-idle per second: 424.7     interval: 5.0s
Power usage (ACPI estimate): 13.616W (charging: 3.1 hours)

Top causes for wakeups:
23.6% (100.1)               <kernel> :  genunix`clock
11.3% ( 48.0)            <interrupt> :  wpi#0
 7.1% ( 30.1)                  sched :  <cross calls>
 3.6% ( 15.1)               <kernel> :  uhci`uhci_handle_root_hub_status_change
 3.5% ( 14.9)            soffice.bin :  <scheduled timeout expiration>
 2.3% ( 10.0)               <kernel> :  ata`ghd_timeout
 2.3% ( 10.0)               <kernel> :  genunix`delay_wakeup


 Q - Quit  R - Refresh
```

have the software try to leverage features that provide hints to the resource manager about when resources are (and are not) being used, to facilitate active power management.

WITH RESPECT TO CPU UTILIZATION:
- When threads are waiting for some condition, try to leverage an event-triggered scheme to eliminate the need for time-based polling. Don't write "are we there yet?" software.
- If the above isn't possible, try to poll infrequently.
- If it can't be eliminated, try at least to ensure that all periodic/polling activity is batch processed. Leverage timer subsystem features that provide latitude for optimization, such as coarsening resolution or allowing for timer advance/deferral.

WITH RESPECT TO MEMORY UTILIZATION:
- Watch for memory leaks.
- Free or unmap memory that is no longer needed. Some operating systems provide advisory interfaces around memory utilization, such as madvise(3c) under Solaris.

WITH RESPECT TO I/O UTILIZATION
- If possible, buffer/batch I/O requests.

## DRIVING TOWARD AN EFFICIENT SYSTEM STACK

Every so often, evolution and innovation in hardware design bring about new opportunities and challenges for software. Features to reduce power consumption of underutilized system resources have become pervasive in even the largest systems, and the software layers responsible for managing those resources must evolve in turn—implementing policies that drive performance for utilized resources while reducing power for those that are underutilized.

Beyond the resource managers, resource consumers clearly have a significant opportunity either to contribute to or undermine the efficiency of the broader stack. Though getting programmers to think differently about the way they design software is more than a technical problem, tools such as PowerTOP represent a great first step by providing programmers and administrators with observability into software inefficiency, a point of reference for optimization, and awareness of the important role software plays in energy-efficient computing. Q

**LOVE IT, HATE IT? LET US KNOW**
feedback@queue.acm.org

**ERIC SAXE** is a staff engineer in the Solaris Kernel Development Group at Sun Microsystems. Over the past 10 years at Sun, he has worked on a number of scheduler/dispatcher-related kernel components including the CMT (chip multithreading) scheduling subsystem, the Power Aware Dispatcher, and MPO (the Solaris NUMA framework), and he is the lead inventor for several related U.S. patents. He graduated from the University of California at San Diego with a B.S. in computer engineering. He lives in the Bay Area with his wife and three children.

© 2009 ACM 1542-7730/10/0100 $10.00