

Improved Power Management for FreeBSD

Brandon J Thomson
G00312321

for CS 695 with Prof. Fei Li

December 21, 2010



Abstract

This project aims to use insights from the green computing literature studied in this course to improve the power management code in the FreeBSD operating system: specifically, the CPU speed-state management. We demonstrate a definite reduction in job completion time and a possible reduction in energy use with a more efficient speed-state manager that runs in the FreeBSD kernel.

Contents

1	Introduction	2
1.1	Motivation	3
1.2	Special Considerations	3
2	Problem: FreeBSD’s CPU Speed State Management is Suboptimal	4
2.1	Slow Frequency Increase	6
2.2	Slow Frequency Decrease / Excessive Transitions	6
2.3	Burst Problem	7
3	Improved Algorithm	9
3.1	Measurement of Task Completion Time	11
4	Future Work	12
4.1	Linux Features That Would Be Useful	13
A	Configuration Details	14
A.1	Test System	15
A.2	Basic Configuration Changes	16
A.3	Kernel Build Tips	16
A.4	Instrumentation and Measurement	17
A.4.1	Timing	18
B	Speed State Transition Delay Measurement	19
B.1	Comparison to Time Slice Duration	20
B.2	Source Code	20
C	Kernel Patches	23
C.1	/usr/src/sys/kern/kern_clock.c	23
C.2	/usr/src/sys/kern/kern_cpu.c	26

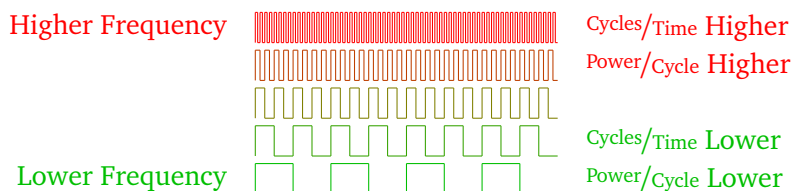


Figure 1: Standard CPU speed trade-offs: Higher speeds allow more work to be done in a certain amount of time, but also use more power for each unit of work. In this paper, we assume that the user would prefer to use higher speeds when the system is under load even though they are less efficient.

1 Introduction

Most CPUs support multiple speed states. To optimize the competing goals of CPU power consumption, task throughput, and task latency, it is necessary for a host operating system’s software to actively manage these hardware states.

It is usually the case that slower speed states achieve more cycles per watt and are thus more efficient than faster speed states (Figure 1), although there may be some “critical speed” above the minimum speed which achieves the best cycles per watt in the case that the system is continuously busy[1].

Faster speed states by definition complete more cycles per unit time, and thus the goal of minimizing power consumption will be at odds with the goal of achieving maximum compute throughput.

Slowing the CPU speed dynamically is desirable when the system does not have enough jobs to keep it continuously busy, which is the main condition studied in this paper. We operate under the assumption that unless instructed otherwise, the user would prefer the system to switch up to the maximum speed state available whenever the system becomes sufficiently busy to warrant it.

Optimizing the state of devices with multiple speed states to minimize power consumption while meeting certain goals is a problem that has been considered many times in the literature in many different forms. Unfortunately these problems are sometimes framed in an idealized perspective such that the algorithms produced do not lend themselves well to use in actual implementations. Naïvely implementing such algorithms is unlikely to produce satisfactory

results. Ideas should always be tested on real hardware.

In this paper we are specifically considering improvements to FreeBSD's speed-state power management code. FreeBSD is a free, general-purpose UNIX-derived operating system. It is available from <http://www.freebsd.org>.

Due to limited hardware availability for testing, we'll be looking at the Intel x86 architecture specifically. We expect most of the code produced to be applicable to other architectures as well, but this has not been tested yet.

1.1 Motivation

Various versions of FreeBSD are probably currently running on hundreds of thousands or millions of computers worldwide, many of which are servers which are constantly running, 24 hours a day.

Even small amounts of energy usage reduction in the operating system, when multiplied by this large install base, translate to big total savings. For example, assume the average server running FreeBSD has 4 CPUs and runs for 18 hours per day. If there are 1 million such machines in the world¹ and our improvements lead to an average of just 1 watt saved per CPU, this would lead to a total of 72 megawatt hours saved every single day. As we will show the current algorithm is not very efficient with certain workloads, and so a savings of 1 watt per CPU may even be a conservative estimate.

At a typical rate of 7¢ per kilowatt hour, 72 megawatt hours translates to a total of about \$5000 in total saved by FreeBSD users every day, and a measurable reduction in greenhouse gas emissions as well.

1.2 Special Considerations

Unlike academic work in this area where practical concerns with potential implementations are often ignored in favor of making mathematical analysis of a problem simpler or more tractable,

¹It's hard to know exactly how many machines are running FreeBSD since people can download as many copies as they want for free and there's no reliable way to track usage. Netcraft used to run a survey which showed about 2.5 million websites running on FreeBSD as of June 2004 [2], but such surveys only show machines used to host websites, can over count because some machines host multiple sites, and can undercount because hosts sometimes hide the operating system they are using for security reasons. A more recent survey [3] points to FreeBSD serving 3.3% of all hosts on the Internet matching certain criteria.

here practical concerns are **paramount**: the goal of the project is to get the source code changes accepted into the next version of FreeBSD. If the code does not work better *in practice*, it will be rejected.

Open source projects maintained by volunteers also tend to favor simplicity above accuracy: minor optimizations which improve an algorithm's time or space complexity² but add many extra lines of code or which make the code more difficult to understand are not always appreciated.

A final factor to consider is that we must think in the context of the entire system that is FreeBSD. For example, if FreeBSD already records some information for other purposes which is useful for the work we are doing here, we should probably try to reuse this information rather to record our own slightly different information which may appear more optimal in the abstract. In this way we keep the *additional* computational and memory overhead due to our own code to a minimum.

In other words, it is **not** the absolute time or space complexity of our algorithm that matters, but rather the **additive effect** on the entire system.

2 Problem: FreeBSD's CPU Speed State Management is Suboptimal

FreeBSD's current speed-state algorithm is better than nothing, but increases CPU speed after a fairly long delay when the system is under full load and doesn't reduce speed very quickly after the CPU becomes idle. This both causes unnecessary delays in the completion of jobs, and doesn't use available power optimally (Figure 2).

The speed-scaling daemon `powerd` is implemented as a userland application rather than being in the kernel, and doesn't integrate well with the operating system's sleep-state code. This is probably because they were written at different times by different people³, as is typical for large open source projects.

A simplified version of FreeBSD's current speed-scaling algorithm (`powerd`) with the default

²For example, A very common practice seen in FreeBSD when recording data over time is to keep a weighted average rather than a full time series, even if the full time series would be more optimal. The weighted average only requires one word in memory and a less complex data structure.

³Colin Percival wrote the predecessor to `powerd` (FreeBSD's current speed-scaling controller) with later updates by Nate Lawson. Mr. Lawson also updated `acpi_cpu.c`, the code that controls processor sleep states, which was originally written by Michael Smith.

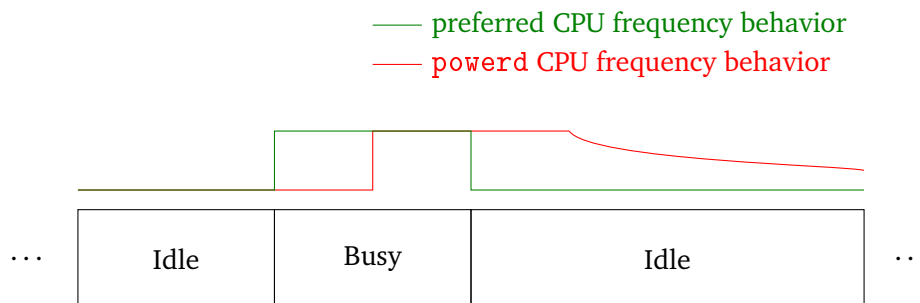


Figure 2: `powerd` is better than nothing, but it takes a long time to detect that the system has become busy and increase the frequency, and then only reduces the frequency slowly after the system has become idle.

Algorithm 1: `powerd` (old algorithm)

```

while true do
  if load > 95% then
    | quadruple frequency
  else if load > 75% then
    | increase frequency slightly
  else if load < 50% then
    | reduce frequency slightly
    | sleep for 250 ms
end

```

Figure 3: A simplified version of FreeBSD’s current speed-scaling algorithm (`powerd`). Note that “load” is a percentage representing how often the system was busy during the previous 250 ms interval, referred to elsewhere as the “Observation Window”. The machine’s level of busy-ness is recorded by the kernel during job scheduling.

configuration settings is shown in Figure 3. Note that “load” is a percentage representing how often the system was busy during the previous 250 ms interval, referred to elsewhere as the “Observation Window”. The machine’s level of busy-ness is recorded by the kernel during job scheduling. Also, in practice quadrupling the frequency is the same as increasing the frequency to maximum since most real CPUs don’t offer a range of frequencies that varies by more than 4x.

More details about the algorithm can be found by reading the actual source code for the daemon⁴. There is also a nice summary of FreeBSD’s entire ACPI layer by one of the maintainers

⁴The source can be viewed online at <http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/usr.sbin/powerd/powerd.c>

in [4] and [5].

2.1 Slow Frequency Increase

Based on the default 250 ms polling delay and 75% threshold, it should be clear that, in the worst case, the system can be at 100% load for 437.5 ms before the CPU's frequency will be increased to maximum. Given that we can transition the CPU from any speed to maximum speed in less than 0.9 ms (see Section B, [Speed State Transition Delay Measurement](#)), this seems like a long time to wait.

In the model given in [1], the algorithm may decide to change speed at any time and the time between transitions can be arbitrarily small. We think some delay prior to transitioning is warranted given that transition time is so large (transition time is assumed to be 0 in [1]), but we also think the maximum delay in transition to maximum speed should be less than 437.5 ms in the default case.

One obvious fix would be to keep the existing algorithm and reduce its polling interval from 250 ms to something shorter. The problem is that this increases background system load by quite a significant amount even when the system is mostly idle. As a userland application, `powerd` has to be scheduled by the kernel to run at regular intervals, a setup which has significantly more overhead than simply running an algorithm inside the kernel itself.

This sort of polling is particularly discouraged with tickless kernels because it will force the CPU to wake from low-power sleep more often than it would otherwise need to [6]. For `powerd`, we'd be waking from low-power sleep just to check the CPU's speed, which is counter-productive to our power-saving goals: it would have been better for the CPU to just stay asleep until there was some *real* work to do.

A better design would not rely on periodic polling at all. Our new algorithm is implemented in the kernel and does not use polling.

2.2 Slow Frequency Decrease / Excessive Transitions

The current algorithm (`powerd`) gradually reduces frequency down to the minimum over a long period of time after the system becomes idle; it took more than 20 s to reach the minimum

frequency on our test system. This behavior (actual measurement from running system) is shown in Figure 4.

`powerd` also reduces the frequency step-by-step instead of reducing it by an appropriate amount based on recent system load. Step-by-step decreases are undesirable when the system is mostly idle because the CPU could be sleeping during that time instead. Sleeping uses less power than transitioning.

One advantage of keeping the CPU speed “too high” for a while after the system has become mostly idle is that busy periods tend to be correlated: if the system has recently been busy, we can say with more certainty that it is likely to be busy 250 ms from now than 50 s from now. This is probably why `powerd` was designed to decrease CPU speed slowly.

However, the associated disadvantage with this behavior is that the CPU will consume extra power without getting much benefit for it as long as the speed remains high and the system remains mostly idle.

If the system can increase its frequency more quickly in response to demands on compute power, there is less necessity to leave the frequency high after busy periods to in anticipation of additional busy periods. This should allow us to save power on most workloads by reducing the frequency more quickly and in bigger steps after the system becomes idle.

2.3 Burst Problem

The slow frequency decrease mentioned in the previous section is normally not that big of a deal, but when small bursts of activity occur frequently the system can be held at a higher-than-optimal speed state for extended periods of time. An example is shown in Figure 5.

Reducing speed more quickly when the system goes idle should help solve this problem. Alternately, the system could attempt to detect this type of usage and intentionally delay increasing system speed. That would make the burst activities complete more slowly, but would also save energy.

More information about how this data was recorded is available in Section [A.4, *Instrumentation and Measurement*](#).

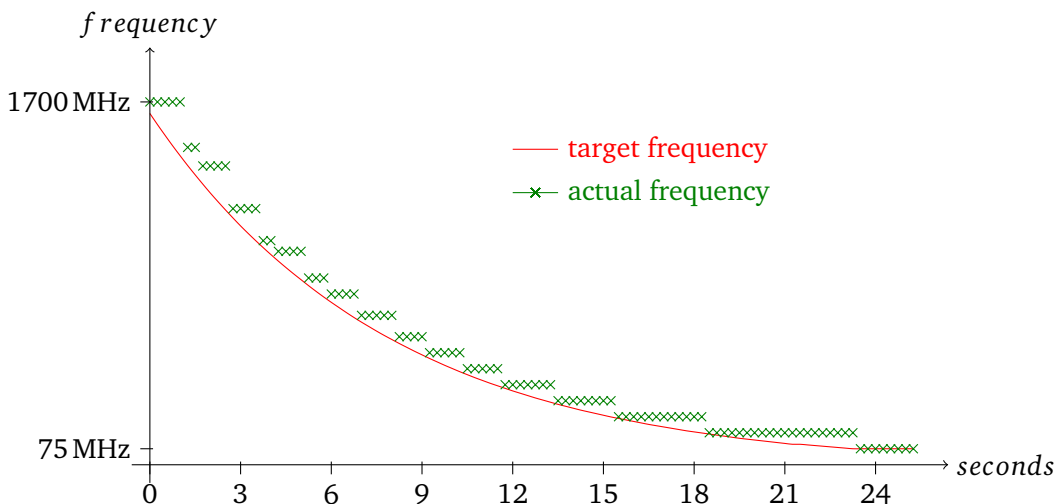


Figure 4: Behavior of current algorithm shown on a system with 17 speed states ranging from 75 MHz up to 1700 MHz over a period of 26 seconds. System was previously busy and thus is at maximum speed at $t = 0$, but is idle enough during the entire interval shown. Thus, at every decision point (represented by each \times), the algorithm always decides to decrease target frequency.

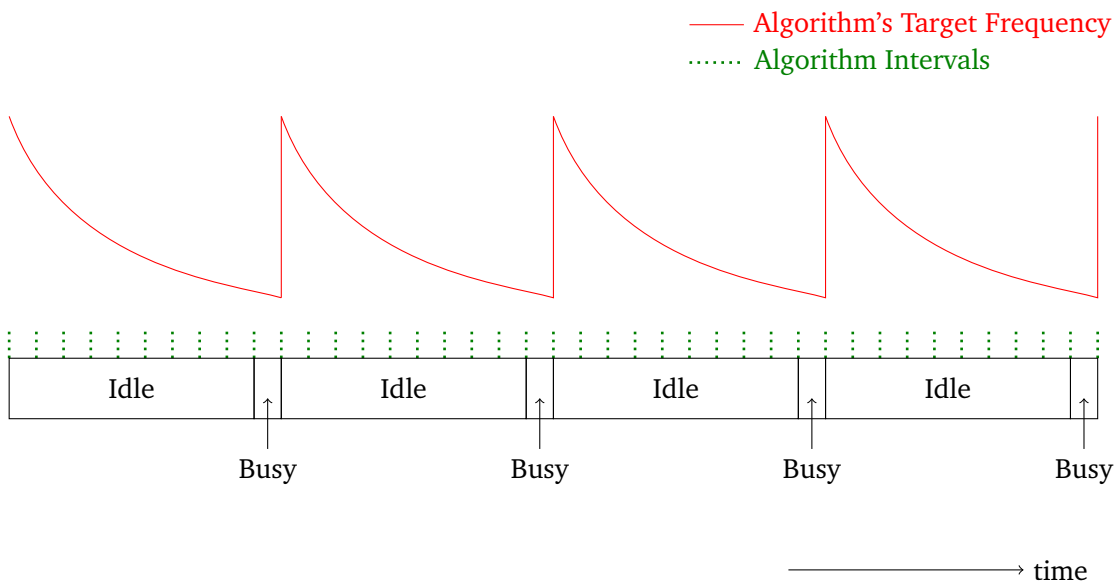


Figure 5: Since the original algorithm increases speed faster than it decreases speed, periodic short bursts of activity which cover at least 95% of the prior observation window can keep CPU speed elevated higher than it needs to be for long periods of time.

3 Improved Algorithm

In this section we present an algorithm we believe is an improvement on the one currently included with FreeBSD, and some measurement results to support our claim.

Based on the issues described in the previous section, our main goals for the improved algorithm are:

1. to increase frequency faster when the system becomes busy,
2. to reduce excess time spent in the high speed state when the system is not busy, and
3. to improve performance with “bursty” system loads.

The biggest change is that the new algorithm is implemented directly in the kernel, rather than as a userland daemon which “polls” periodically. This lets us detect when the CPU is too busy or idle quickly and reduces overhead compared to `powerd` which runs in user mode.

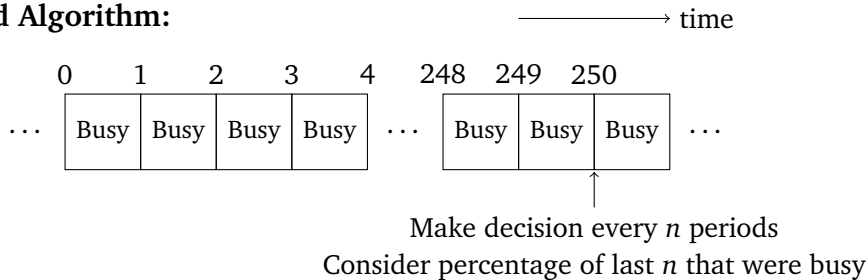
The soonest we can reliably detect that the system has become busy is when the kernel “ticks” and realizes it interrupted something involving real work (such as a user process, an interrupt handler, or itself), as opposed to the idle loop.

Our new algorithm is very simple. The general idea is to increase the speed after a certain fixed number of these detected busy periods, and decrease the speed after a certain fixed number of idle periods. Figure 6b shows this idea graphically.

Currently, the algorithm increases CPU speed to maximum on the first set of consecutive busy periods detected, but reduces one speed step at a time for each set of consecutive idle periods detected. Increasing to maximum speed immediately is probably not optimal, but since it’s difficult to predict how long the CPU will remain busy this may be the best we can do. It seems to work better than increasing one step at a time, at least on our test CPU, and the behavior is similar to what the old `powerd` does.

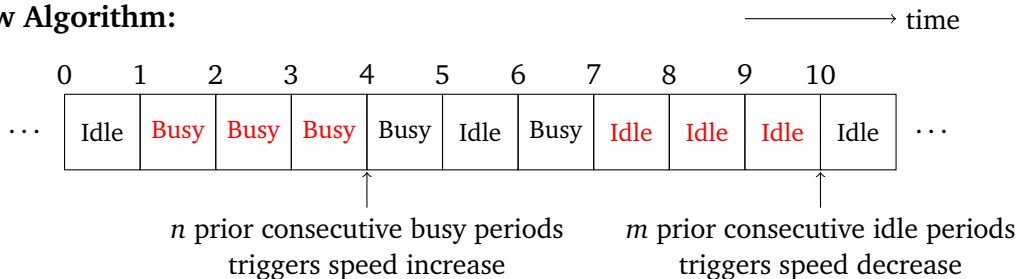
Decreasing speed in larger intervals than one step at a time may be beneficial as well, and more testing would reveal what strategy works best for our specific test CPU. Unfortunately, the best approach is likely to vary from CPU model to CPU model, and it may be hard to write a single algorithm which performs well on every CPU with any arbitrary combination of speed states.

Old Algorithm:



(a) Old algorithm observes percentage of busy/idle periods in fixed-size windows of size n . (Here, $n = 250$... close to the default setting). Due to the way the code is implemented, making n small will dramatically increase constant overhead. A large n is a practical requirement.

New Algorithm:



(b) New algorithm switches immediately after certain number of consecutive busy/idle periods. Here, $n = m = 3$. This algorithm is efficient with small n since it is implemented in the kernel.

Figure 6: A simplified comparison of the ways the old and new algorithms respond to busy/idle periods as detected by the kernel tick.

Patches to the kernel source which implement this algorithm are given in Section [C, *Kernel Patches*](#). We don't include the entire source files in this document since they're too long.

The next section describes a few measurements which should help provide assurance the new algorithm behaves better than the old one. These measurements are not comprehensive but provide a good start.

3.1 Measurement of Task Completion Time

The primary improvement we're able to conclusively demonstrate through measurement is in responsiveness. Since the old speed-scaling algorithm is slow to respond, big jobs which are released when the CPU is at the minimum speed end up taking longer to complete than they need to. This measurement is designed to show how the improved responsiveness of the new algorithm leads to faster completion times for these types of jobs compared to the old `powerd`.

At present, there's no way to disable the new speed-scaling algorithm without booting from a new kernel⁵, so these measurements were taken on different kernels: one with our changes, and one GENERIC kernel. The GENERIC kernel was used to measure both the `powerd` results and the behavior when the frequency was forced to maximum.

The test simply consists of running a python script which runs in a loop for 100000 iterations. This constitutes an essentially fixed amount of work, and so if it is finished faster in one test over another, that is because the CPU was running at a faster speed during more of the job's lifecycle in one test over another.

The results are shown in in Table 1. Notice that the improved algorithm is able to achieve a job runtime closer to what the CPU can achieve when it is forced to run at maximum speed continuously, but at a much lower temperature (indicating less power consumed during idle time). `powerd` is slower because it waits longer to increase the speed.

Another interesting fact is how much greater the variation is between runs when using `powerd` compared to the other options, as shown by the value after the \pm symbol. This is an artifact of the large window size `powerd` observes over: sometimes you get lucky and the window interval begins right before the CPU becomes busy, and sometimes you get unlucky

⁵FreeBSD does have a mechanism called `sysctl` that makes it easy to tune values in the kernel at runtime and we plan to use this to enable/disable the new algorithm on the fly, but this hasn't been implemented yet.

	Old powerd Algorithm	New Algorithm	Maximum Speed Forced
Runtime	386.2 ms \pm 30.7 ms	279.2 ms \pm 2.1 ms	229.2 ms \pm 0.1 ms
CPU Temp	34.0 °C	34.0 °C	45.0 °C

Table 1: A comparison between the old speed-scaling algorithm, the new algorithm, and the system behavior when the speed is forced to the maximum. For both the old powerd algorithm and the new algorithm, the CPU is idling at minimum speed when the test begins. Times shown are averaged over 10 trials as mean \pm std_dev. Lower is better for both CPU Temp and Runtime.

and the CPU has some busy periods in the first window, but not enough to trigger the frequency increase.

We also recorded CPU temperature as a proxy for power usage. This temperature is measured through a sensor on the CPU die. Unfortunately, the sensor is not sensitive enough to detect any differences between the old powerd algorithm and the new algorithm: neither of them heats the CPU from its idle temp of 34 °C at 600 MHz.

Some other methods which may be effective at measuring the differences in power consumption are described in Section [A.4, Instrumentation and Measurement](#), but we have not tested these methods yet. We do anticipate that the new algorithm uses less power than the old one in addition to completing jobs faster because it decreases the CPU speed so much more quickly. We need to develop a more sensitive measurement capability in order to show this.

4 Future Work

The improvements are still a work in progress. Not every issue raised in Section [2, Problem: FreeBSD's CPU Speed State Management is Suboptimal](#) has been addressed. The code also needs a lot of practical improvements (better organization, documentation, etc) before I can consider submitting a patch to the FreeBSD project.

Currently, the algorithm relies on the kernel tick to detect whether the CPU is idle or busy. It might be possible to learn that the CPU has become busy before this time using CPU performance counters or advanced hardware features on certain platforms. Our method is more general, but possibly less efficient on some platforms with these features.

Detecting that the CPU has become idle can definitely be done sooner by including a hook

in the kernel's idle loop, but we have not implemented this yet. Detecting the CPU is idle sooner should reduce power consumption by a tiny amount.

Support for CPUs that can have each core's speed adjusted independently is still needed in FreeBSD: these were not supported by FreeBSD's original algorithm and my changes do not fix the problem. This is a big change and will require a lot of `cpufreq` to be rewritten.

Before submitting a patch to the FreeBSD project, it would also be prudent to have measurements from many systems which show the benefit of the new algorithm rather than just the IBM R51 shown in this paper. I would have documented results on more systems for this report, but you need to install my experimental kernel to measure the results so there is some significant effort and risk involved.

4.1 Linux Features That Would Be Useful

Current versions of Linux have two features[7] which reduce CPU power consumption compared to FreeBSD:

Tickless Scheduling Only wake the CPU from sleep when necessary, instead of at regular fixed intervals.

Scheduler Power Saving Mode Consolidate processes onto similar CPU cores in SMP systems.

Tickless kernels are now possible because of new hardware features that didn't exist in older CPUs. Some work has been done towards implementing a tickless kernel in FreeBSD[8], but as of Dec 2010, nothing is ready for production.

The big advantage of a tickless kernel is that deeper sleep states become more useful. If the delay between kernel ticks is less than the minimum latency of a particular sleep state, the state can never be used because it would make kernel timing become inaccurate.

Some CPU sleep states which are feasible at a given kernel tick rate based solely on their latency may still be useless due to power costs associated with the transition [6].

This paper has mostly considered speed state management, however [1] suggests that further improvements may be possible by considering speed and sleep states simultaneously. Linux's *Scheduler Power Saving Mode* takes the next step of also making the process scheduler aware of CPU power management, an approach which would have benefits for FreeBSD as well.

References

- [1] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. “Algorithms for power savings”. In: *ACM Trans. Algorithms* 3 (4 2007). ISSN: 1549-6325. URL: <http://doi.acm.org/10.1145/1290672.1290678>.
- [2] Mandy Davis. *Nearly 2.5 Million Active Sites running FreeBSD*. Netcraft Ltd. 2004. URL: http://news.netcraft.com/archives/2004/06/07/nearly_25_million_active_sites_running_freebsd.html.
- [3] Netcraft Ltd. *Operating System Software used at Sites in All Locations January 2009*. 2009. URL: https://ssl.netcraft.com/ssl-sample-report//CMatch/oscnt_all.
- [4] Nate Lawson. *ACPI and FreeBSD (Part 1)*. Bay Area FreeBSD Users Group. 2006. URL: http://www.root.org/talks/ACPI_20060503.pdf.
- [5] Nate Lawson. *ACPI and FreeBSD (Part 2)*. Bay Area FreeBSD Users Group. 2006. URL: http://www.root.org/talks/ACPI_20060906.pdf.
- [6] Eric Anholt. *PowerTOP for FreeBSD ?* freebsd-acpi mailing list. 2007. URL: <http://lists.freebsd.org/pipermail/freebsd-acpi/2007-May/003712.html>.
- [7] *Saving power on Intel systems with Linux*. URL: <http://www.lesswatts.org/results/server/index.php>.
- [8] Prashant Vaibhav. *Reworking the call-out API: towards a tickless kernel*. 2009. URL: <http://wiki.freebsd.org/SOC2009PrashantVaibhav>.
- [9] Alexander Motin. *Tuning Power Consumption*. 2010. URL: <http://wiki.freebsd.org/TuningPowerConsumption>.
- [10] George Neville-Neil. *Tuning SCHED_ULE on FreeBSD*. 2009. URL: http://www.bsdcn.org/2009/schedule/attachments/101_sched_tuning.pdf.

A Configuration Details

This section discusses details about the specific test system used, measurement methods, configuration gotchas with FreeBSD, code details, and adds support for decisions made when working



Figure 7: Graphical representation of the FreeBSD modules relevant to this project.

on the improved algorithm.

It is intended to be helpful for anyone wishing to extend this work but is not critical reading in order to understand the improvements presented in Section 3, *Improved Algorithm*.

There are three FreeBSD modules which are most relevant to our studies: `cpufreq`, `acpi_cpu`, and `powerd`. `cpufreq` contains the specific CPU or architecture-dependent CPU speed-changing code which MUST be in the kernel, `acpi_cpu` controls CPU sleep states, and `powerd` is a userland daemon which makes CPU speed transition decisions and uses the interface exposed by `cpufreq` to carry them out. A visual representation is shown in Figure 7.

A.1 Test System

The main test system for this project is an IBM R51 laptop with an Intel Pentium M processor. Its CPU supports speeds of 1700 MHz, 1400 MHz, 1200 MHz, 1000 MHz, 800 MHz, and 600 MHz, and sleep states C1, C2, and C3. Anywhere measurements are shown in this paper, you should assume they were taken on this system.

FreeBSD version 8.1-RELEASE-p1 was used. This was the latest stable version at the time of this writing.

Although improvements in measured performance on one system is a good indication that the algorithm has been improved, it would be prudent to perform measurements on additional types of hardware before the code is updated in FreeBSD's repository. This issue and related others are described in Section 4, *Future Work*.

A.2 Basic Configuration Changes

By default FreeBSD is not able to make use of C3-type states because they take so long to recover from, it would mess up kernel timing. Until FreeBSD moves to a tickless kernel[8] you may be able to take advantage of them by decreasing the `kern.hz` value to 100 or less.

For this project, `kern.hz` was set to 100. The default value of 1000 causes CPU temperature at idle to rise significantly compared to 100.

I also recommend disabling the older ACPI throttling mechanism (T states). For some reason these are enabled by default on new systems (probably a mistake). They create a set of pseudo speeds which don't actually exist. It is preferable to only use real speed states and manage CPU sleep through the newer C-state mechanisms.

You can disable the throttling with `hint.acpi_throttle.0.disabled=1` in the tunables file `/boot/loader.conf` [9].

A.3 Kernel Build Tips

The worst part of working on large C programs is waiting for the compiler to finish just to find out whether you had a syntax error or something. Using FreeBSD's default kernel build options will quickly make you go insane when actively making changes to the kernel because it takes nearly 30 minutes to rebuild after every single change. Using `ccache` and omitting non-essential kernel modules got the kernel build process down to about 2 minutes on our test machine which is much more manageable.

```
# Allow ccache to work with the kernel build process
.if (!empty(.CURDIR:M/usr/src*) || !empty(.CURDIR:M/usr/obj*)) && !defined(NOCCACHE)
CC=/usr/local/libexec/ccache/world-cc
CXX=/usr/local/libexec/ccache/world-c++
.endif

KERNCONF=POWERMOD

NO_MODULES=true
```

Listing 1: Suggested additions to `/etc/make.conf`.

To speed this process up, first, install `ccache` from ports and add something like Listing 1 to `/etc/make.conf`. Using `ccache` and disabling building modules should reduce kernel build

time to a minute or so which is much more manageable. Before you do this you have to make sure your kernel configuration (`/usr/src/sys/i386/conf/POWERM0D` in this example) has everything you need statically included so there is no need for any kernel modules to be loaded at runtime.

Then, to build the experimental kernel, install it into a staging area, and reboot the system with it (but automatically revert to the previous kernel on the boot after that in case there is a problem), you can use the short script given in Listing 2. The script will abort if there is a failure building the kernel so you can fix whatever syntax error you made. Automating all of the above steps makes it much easier to test changes to the kernel quickly, and got the build/install/reboot cycle down to just under 3 minutes for our test machine.

```
#!/bin/sh
set -e # abort on error

cd /usr/src
make buildkernel
make installkernel KODIR=/boot/kernel.TEST
nextboot -k kernel.TEST
shutdown -r now
```

Listing 2: Suggested script to build kernel and reboot into it automatically.

A.4 Instrumentation and Measurement

One goal of this project is to be able to demonstrate through measurement the superiority of the new algorithm. In order to do that, we want to be able to see the behavior of the old and new algorithms as clearly as possible. Showing the ways in which the new algorithm is better will also help convince others that they should switch to using the new code.

There are several methods we can potentially use to measure power consumption. Each method of measurement has its own benefits and drawbacks:

1. Install a watt-meter between computer's plug and wall-socket

Specialized equipment could potentially get you a lot of accuracy this way, but we don't have such equipment available. Also, you are measuring the entire system's power con-

sumption, not just the CPU.

2. Software calculations based on manufacturer's power figures for speed/sleep states

Often, CPU manufacturers publish average or maximum power consumption figures for each speed and sleep state. If we have this information and record the state the processor is in over an interval of time, we can compute an "expected" power consumption figure.

3. ACPI CPU Temperature Sensor

This is a sensor on the CPU die.

CPU temperature does correspond to power usage in the CPU, although it is more of a measurement of activity over the previous few seconds/minutes than a measurement of current consumption. Can be confounded by temperatures of other components in the system.

This is easy to check on FreeBSD. There are various ways to get it, but one command that will work is `sysctl hw.acpi.thermal.tz0.temperature`. Other BSDs and linux distributions should have similar commands.

4. ACPI Battery Sensors

Most of the behavior of `powerd` can be monitored by launching it with the `-v` switch, although we modified `/usr/src/usr/sbin/powerd/powerd.c` to add timestamps and some other instrumentation.

To enable ACPI debugging output for the CPU, the kernel can be recompiled with the `ACPI_DEBUG` option although we didn't find anything useful for our purposes in the debugging output.

A.4.1 Timing

Timing can be difficult to measure at the precision we need for this project. Older operating systems' clocks were often maintained by counters incremented during timing interrupts generated by the main processor. These interrupts couldn't happen more than 1000 times a second or so to avoid taking up too many resources, which made sub-millisecond timing precision difficult to achieve on such systems.

		Destination Frequency					
		1700 MHz	1400 MHz	1200 MHz	1000 MHz	800 MHz	600 MHz
Start Frequency	1700 MHz	-	1254 μ s	1458 μ s	1745 μ s	2175 μ s	2892 μ s
	1400 MHz	892 μ s	-	1211 μ s	1447 μ s	1802 μ s	2394 μ s
	1200 MHz	818 μ s	912 μ s	-	1249 μ s	1554 μ s	2062 μ s
	1000 MHz	776 μ s	813 μ s	901 μ s	-	1307 μ s	1732 μ s
	800 MHz	760 μ s	747 μ s	792 μ s	878 μ s	-	1403 μ s
	600 MHz	735 μ s	680 μ s	692 μ s	727 μ s	832 μ s	-

Table 2: Measured delay when transitioning from *Start Frequency* to *Destination Frequency* (as measured by userland utility from Section B.2).

Luckily, the hardware we’re using has support for an ACPI timer which runs independently from the main CPU, and in fact continues to increment even when the main CPU is in a sleep state. This timer source runs at 3.579 545 MHz and is fully supported by FreeBSD, so we can use standard clock calls to get approximately 300 ns resolution.

B Speed State Transition Delay Measurement

Changing the CPU speed state requires a certain amount of time. This time is a deadweight loss because power will be expended, but the system is not accomplishing any useful work unrelated to the speed state transition. The delay has two components:

1. Time spent running software algorithms related to the transition
2. Time spent by CPU adjusting itself electrically after the OS has issued the transition command (no instructions executed during this period)

To get a sense of the magnitude of these delays, we measured them for all possible combinations of speed states on our test system. The averages are shown in Table 2.

Although for the most part these values are either always increasing or always decreasing with the change in frequency, there are some exceptions: For example, the transition 600 MHz \rightarrow 1400 MHz is faster than both the transition from 600 MHz \rightarrow 1700 MHz and the transition from 600 MHz \rightarrow 1200 MHz. These aren’t flukes: the transition times are surprisingly consistent from test to test, occasionally varying by as much as 30 μ s but typically within 1 or 2 μ s of the previous run.

It would be interesting to learn how much of this delay is due to kernel scheduling and how much of it is due to the CPU actually switching speeds. The kernel may be taking longer to finish cleanup work and perform a task switch back to the measurement application when the CPU has been switched to certain frequencies. The way to detect this would be to perform measurement from inside the kernel rather than in a userspace application.

It's worth noting that since `powerd` is also a userspace application, the time spent doing any cleanup work represents a real cost that must be paid every time the CPU speed is changed through that mechanism.

The code written to make these measurements is in Section [B.2](#), [Source Code](#).

B.1 Comparison to Time Slice Duration

By default, FreeBSD ticks every millisecond and the default process time slice is 13 ticks[10]. Thus, compute-bound processes should run for 13 ms on average before being interrupted.

Is task-switch time a good time to perform speed state transitions? Maybe, but our longest observed transition time is 1700 MHz → 600 MHz at 2.9 ms. At 22% of the time slice duration, switching CPU speeds during every task switch would clearly be excessive and wasteful. If task-switch time is used for transition evaluation, something should be done to ensure the speed does not transition during too many consecutive task switches.

B.2 Source Code

This is the source code for the application written to measure speed state transition delay time. It should work on any system that is supported by the `cpu_freq` module.

```
// state_delay_check.c          vim: set ts=3 tw=74 shiftwidth=3 expandtab:
//
// Utility application to record time spent switching between CPU speed
// states
//
// Copyright 2010 Brandon Thomson
// Licence: BSD
//
// Some code derived from /usr/src/usr/sbin/powerd/powerd.c (by Colin
// Percival and Nate Lawson)
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#include <sys/time.h>
#include <sys/sysctl.h>

static int freq_mib[4];
static int levels_mib[4];

static int read_freqs(int *numfreqs, int **freqs);

/* freq should be a number like 1700 or 600 */
static int set_freq(int freq);

static int
read_freqs(int *numfreqs, int **freqs) {
    char *freqstr, *p, *q;
    int i;
    size_t len = 0;

    if (sysctl(levels_mib, 4, NULL, &len, NULL, 0))
        return (-1);
    if ((freqstr = malloc(len)) == NULL)
        return (-1);
    if (sysctl(levels_mib, 4, freqstr, &len, NULL, 0))
        return (-1);

    /* count number of frequencies */
    *numfreqs = 1;
    for (p = freqstr; *p != '\0'; p++) {
        if (*p == ',') {
            (*numfreqs)++;
        }
    }

    if ((*freqs = malloc(*numfreqs * sizeof(int))) == NULL) {
        free(freqstr);
        return (-1);
    }

    /* extract frequencies from string */
    for (i = 0, p = freqstr; i < *numfreqs; i++) {
        q = strchr(p, ',');
        if (q != NULL) {
            *q = '\0';
        }
        if (sscanf(p, "%d/%d", &(*freqs)[i]) != 2) {
            free(freqstr);
            free(*freqs);
            return (-1);
        }
        p = q + 1;
    }
}
```

```
    }

    free(freqstr);
    return (0);
}

static int
set_freq(int freq) {
    if (sysctl(freq_mib, 4, NULL, NULL, &freq, sizeof(freq))) {
        if (errno != EPERM)
            return (-1);
    }

    return (0);
}

int main() {
    size_t len = 4;

    struct timeval tp_start, tp_finish;
    struct timezone tzp;

    /* Will average result over this many attempts */
    const int NUM_TRIALS = 80;

    char *freqstr, *p, *q;
    int start_freq_idx, dest_freq_idx, start_freq, dest_freq, numfreqs,
        *freqs, i;
    double result_sum;

    if (sysctlbyname("dev.cpu.0.freq", freq_mib, &len))
        err(1, "freq_mib");
    if (sysctlbyname("dev.cpu.0.freq_levels", levels_mib, &len))
        err(1, "lookup freq_levels");

    if (read_freqs(&numfreqs, &freqs))
        err(1, "error reading supported CPU frequencies");

    for (start_freq_idx = 0; start_freq_idx < numfreqs; start_freq_idx++) {
        start_freq = freqs[start_freq_idx];
        for (dest_freq_idx = 0; dest_freq_idx < numfreqs; dest_freq_idx++) {
            if (start_freq_idx == dest_freq_idx) {
                continue;
            }

            dest_freq = freqs[dest_freq_idx];
            printf("C %d -> %d\n", start_freq, dest_freq);

            result_sum = 0.0;

            for (i = 0; i < NUM_TRIALS; i++) {
                set_freq(start_freq);

                gettimeofday(&tp_start, &tzp);
```

```

    set_freq(dest_freq);
    gettimeofday(&tp_finish, &tzp);

    /* Usually around 2-5 us overhead from gettimeofday commands,
     * but its not critical */
    int diff = tp_finish.tv_usec - tp_start.tv_usec;

    /* Handle rollover */
    if (diff < 0) {
        diff = 1000000 + diff;
    }

    result_sum += diff;
    //printf("%d\n", diff);
}
printf("%.1f\n", result_sum / (double)NUM_TRIALS);
}
}

return 0;
}

```

C Kernel Patches

These are the patches to the kernel source which implement the algorithm described in Section 3, *Improved Algorithm*. These should apply cleanly to 8.1-RELEASE-p1 and possibly other versions as well.

C.1 /usr/src/sys/kern/kern_clock.c

kern_clock.c is where the detection that the CPU is busy or idle is performed.

This is not a particularly brilliant file to put all of this code in, but it was convenient for now.

```

diff --git a/sys/kern/kern_clock.c b/sys/kern/kern_clock.c
index 071e0df..a2be623 100644
--- a/sys/kern/kern_clock.c
+++ b/sys/kern/kern_clock.c
@@ -68,6 +68,11 @@ __FBSDID("$FreeBSD: src/sys/kern/kern_clock.c,v 1.211.2.3.2.1 2010/06/14 02:09:0
#include <sys/limits.h>
#include <sys/timetc.h>

+/* bthomson */
+#include <sys/cpu.h>
+#include <sys/malloc.h>
+/* ***** */
+

```

```

#ifdef GPROF
#include <sys/gmon.h>
#endif
@@ -86,6 +91,21 @@ SYSINIT(clocks, SI_SUB_CLOCKS, SI_ORDER_FIRST, initclocks, NULL);
/* Spin-lock protecting profiling statistics. */
static struct mtx time_lock;

/* bthomson */
+
+/* consecutive busy information */
+static int busy_count = 0;
+static int idle_count = 0;
+
+static int count = 10;
+
+/* externs from kern_cpu.c */
+extern int cf_get_method(device_t dev, struct cf_level *level);
+extern int cf_set_method(device_t dev, const struct cf_level *level, int priority);
+extern int cf_levels_method(device_t dev, struct cf_level *levels, int *count);
+extern device_t cf_hack_device; /* extremely lame and extremely effective */
+/* ***** */
+
+static int
sysctl_kern_cp_time(SYSCTL_HANDLER_ARGS)
{
@@ -583,6 +603,92 @@ stopprofclock(p)
}

+static int cpufreq_increase(void);
+static int cpufreq_decrease(void);
+static void busy(void);
+static void not_busy(void);
+
+static int cpu_level_idx = 0; /* start at max speed */
+
+static void busy(void) {
+    idle_count = 0;
+    busy_count++;
+    //printf("btbusy: %5d %4d", p->p_pid, busy_count);
+    if (busy_count > 3) {
+        busy_count = 0;
+        cpufreq_increase();
+    }
+}
+
+static void not_busy(void) {
+    busy_count = 0;
+    idle_count++;
+    if (idle_count > 3) {
+        idle_count = 0;
+        cpufreq_decrease();
+    }
+}
+
+static int cpufreq_increase(void) {
+    int error = 0;
+    int dest_idx = 0;
+

```

```

+     struct cf_level *levels;
+
+     /* can't change anything if we don't have a device yet */
+     if (!cf_hack_device)
+         return (1);
+
+     if (cpu_level_idx > 0) {
+         levels = malloc(count * sizeof(*levels), M_TEMP, M_NOWAIT);
+         if (levels == NULL)
+             return (ENOMEM);
+         error = cf_levels_method(cf_hack_device, levels, &count);
+         if (error) {
+             if (error == E2BIG)
+                 printf("cpufreq: need to increase CF_MAX_LEVELS\n");
+             free(levels, M_TEMP);
+             return (error);
+         }
+         printf("btcpufreq: %d -> %d (%d)\n", cpu_level_idx, dest_idx, count);
+         cpu_level_idx = dest_idx;
+         error = cf_set_method(cf_hack_device, &levels[cpu_level_idx], 1000000);
+         free(levels, M_TEMP);
+     }
+
+     return (error);
+}
+
+static int cpufreq_decrease(void) {
+     int error = 0;
+
+     struct cf_level *levels;
+
+     /* can't change anything if we don't have a device yet */
+     if (!cf_hack_device)
+         return (1);
+
+     /* XXX beware, count must be set before */
+     if (cpu_level_idx < count - 1) {
+         levels = malloc(count * sizeof(*levels), M_TEMP, M_NOWAIT);
+         if (levels == NULL)
+             return (ENOMEM);
+         error = cf_levels_method(cf_hack_device, levels, &count);
+         if (error) {
+             if (error == E2BIG)
+                 printf("cpufreq: need to increase CF_MAX_LEVELS\n");
+             free(levels, M_TEMP);
+             return (error);
+         }
+         printf("btcpufreq: %d -> %d (%d)\n", cpu_level_idx, cpu_level_idx + 1, count);
+         cpu_level_idx++;
+         error = cf_set_method(cf_hack_device, &levels[cpu_level_idx], 1000000);
+         free(levels, M_TEMP);
+     }
+
+     return (error);
+}
+
+/*
+ * Statistics clock. Updates rusage information and calls the scheduler
+ * to adjust priorities of the active thread.

```

```

@@ -612,6 +718,7 @@ statclock(int usermode)
        cp_time[CP_NICE]++;
        else
        cp_time[CP_USER]++;
+
        busy();
    } else {
        /*
        * Came from kernel mode, so we were:
@@ -629,13 +736,18 @@ statclock(int usermode)
        td->td_intr_nesting_level >= 2) {
            td->td_iticks++;
            cp_time[CP_INTR]++;
+
            busy();
        } else {
            td->td_pticks++;
            td->td_sticks++;
-            if (!TD_IS_IDLETHREAD(td))
+            if (!TD_IS_IDLETHREAD(td)) {
                cp_time[CP_SYS]++;
-            else
+
                busy();
+
            }
+
            else {
                cp_time[CP_IDLE]++;
                not_busy();
+
            }
        }
    }
}

```

C.2 /usr/src/sys/kern/kern_cpu.c

kern_cpu.c contains an abstraction layer for CPU frequency modification. Normally you'd change the frequency using sysctl from a userland application like powerd does, but by making some modifications in here we are able to call these functions directly from inside the kernel.

I don't understand this code as well as I'd like to, so I probably didn't choose the best way of modifying it, but it works.

```

diff --git a/sys/kern/kern_cpu.c b/sys/kern/kern_cpu.c
index c4ad576..1fefec2 100644
--- a/sys/kern/kern_cpu.c
+++ b/sys/kern/kern_cpu.c
@@ -99,10 +99,10 @@ TAILQ_HEAD(cf_setting_lst, cf_setting_array);
    static int      cpufreq_attach(device_t dev);
    static void     cpufreq_startup_task(void *ctx, int pending);
    static int      cpufreq_detach(device_t dev);
-static int      cf_set_method(device_t dev, const struct cf_level *level,
+int             cf_set_method(device_t dev, const struct cf_level *level,
                             int priority);
-static int      cf_get_method(device_t dev, struct cf_level *level);
-static int      cf_levels_method(device_t dev, struct cf_level *levels,

```

```

+int      cf_get_method(device_t dev, struct cf_level *level);
+int      cf_levels_method(device_t dev, struct cf_level *levels,
                          int *count);
static int      cpufreq_insert_abs(struct cpufreq_softc *sc,
                          struct cf_setting *sets, int count);
@@ -140,6 +140,8 @@ SYSCTL_INT(_debug_cpufreq, OID_AUTO, lowest, CTLFLAG_RW, &cf_lowest_freq, 1,
SYSCTL_INT(_debug_cpufreq, OID_AUTO, verbose, CTLFLAG_RW, &cf_verbose, 1,
        "Print verbose debugging messages");

+device_t cf_hack_device = NULL;
+
static int
cpufreq_attach(device_t dev)
{
@@ -149,6 +151,9 @@ cpufreq_attach(device_t dev)
        uint64_t rate;
        int numdevs;

+        /*bthomson*/
+        cf_hack_device = dev;
+
        CF_DEBUG("initializing %s\n", device_get_nameunit(dev));
        sc = device_get_softc(dev);
        parent = device_get_parent(dev);
@@ -232,7 +237,7 @@ cpufreq_detach(device_t dev)
        return (0);
}

-static int
+int
cf_set_method(device_t dev, const struct cf_level *level, int priority)
{
        struct cpufreq_softc *sc;
@@ -405,7 +410,7 @@ out:
        return (error);
}

-static int
+int
cf_get_method(device_t dev, struct cf_level *level)
{
        struct cpufreq_softc *sc;
@@ -508,7 +513,7 @@ out:
        return (error);
}

-static int
+int
cf_levels_method(device_t dev, struct cf_level *levels, int *count)
{
        struct cf_setting_array *set_arr;

```