An Experimental Survey of Energy Management Across the Stack

Melanie Kambadur
Columbia University
melanie@cs.columbia.edu

Martha A. Kim
Columbia University
martha@cs.columbia.edu

Abstract

Modern demand for energy-efficient computation has spurred research at all levels of the stack, from devices to microarchitecture, operating systems, compilers, and languages. Unfortunately, this breadth has resulted in a disjointed space, with technologies at different levels of the system stack rarely compared, let alone coordinated.

This work begins to remedy the problem, conducting an experimental survey of the present state of energy management across the stack. Focusing on settings that are exposed to software, we measure the total energy, average power, and execution time of 41 benchmark applications in 220 configurations, across a total of 200,000 program executions.

Some of the more important findings of the survey include that effective parallelization and compiler optimizations have the potential to save far more energy than Linux's frequency tuning algorithms; that certain non-complementary energy strategies can undercut each other's savings by half when combined; and that while the power impacts of most strategies remain constant across applications, the runtime impacts vary, resulting in inconsistent energy impacts.

1. Introduction

Modern computational needs and resource constraints have promoted energy efficiency to a first order design goal, precipitating a wide array of energy conservation techniques from the circuit to the user and everywhere in between. Despite marked advances in energy efficiency, the anticipated constraints of future domains such as wearable or implanted computers necessitate continued advances.

The fragmentation of work between different communities is one obstacle to progress. Individual research papers tend to compare a new technique against the next closest, which rarely extends into other layers of the system stack. When the energy savings of a new technique are not compared to existing techniques at multiple levels of the stack, it is hard to evaluate the new idea's broader impact to energy research. Since experimental methods vary widely, using different hardware, versions of the OS, compilers and flags, languages, and benchmarks, comparing results across research papers is rarely a viable option. For example, some studies report power while others report energy, some measure power while others model it, and some report usage for the entire package while others report usage only for the cores. Accurately comparing a new technique to old techniques requires normalized experimental evaluation methodologies on similar software and architectural platforms.

Understanding how a research project fits into the quantitative landscape of existing work enables researchers to evaluate the new work's energy savings and tradeoffs in the proper context. For example, if one strategy decreases energy consumption by 50% but requires new hardware, it might be less desirable than an alternative that saves only 40% but uses commodity hardware. Or, a language extension that saves 200% of the energy of existing system level strategies may be more readily adopted into the language standard than one that saves only 20%. It is also important to understand how techniques combine, both in deployment and when discerning the most promising future research directions. For example, if a compiler level energy optimization complements an operating system level technique, both techniques merit further investigation regardless of which saves more in isolation. However, if one eclipses or eliminates the impact of the other, the lower saver may be less valuable.

To restore a broad context for software energy research, this work measures the relative power, performance, and energy effects of a range of energy management strategies. While most of the strategies we study have been previously evaluated in some context, this is the first time that all of the results can be compared, because our experiments have standardized the architecture, OS, measurement tools, and benchmarks. We examine each technique in isolation

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660196

as well as in combination with other techniques at different parts of the system. Examining 220 experimental configurations of 41 applications totaling more than 200,000 trial runs, we juxtapose the energy impacts of frequency scaling, sleep states, parallelism, compiler optimizations, application-specific power caps, and source-level optimizations. These are some of our key findings.

There is only so much room to save power in software (Section 3). We found that the lowest system baseline power (i.e., the operating system running with no user applications) consumed 60% of serial application power, and 35% of the power of a well parallelized application. Moreover, single-threaded power varies relatively little across programs.

Linux does not provide energy-efficient frequency tuning algorithms (Sections 4.1, and 4.5). Add us to the chorus [39] noticing that Linux's energy-efficient frequency scaling algorithm, *ondemand*, is not great at its purported job. Particularly when applications were parallelized, ondemand often increased energy rather than saving it. The aptly named *powersave* algorithm does save some power but at great cost to performance, so it is also an energy loser.

Overclocking has little to no effect on energy (Sections 4.1 and 4.5). While overclocking saves runtime, it eats away a commensurate amount of power, resulting in no net effect on energy for most applications. At increased thread counts (e.g., 16 threads), overclocking's power increases begin to outstrip its runtime savings, meaning overclocking reduces energy by a small amount.

Parallelization can save so much energy relative to other strategies that energy-conscious software developers must embrace it (Section 4.3). Most desktop, server, and mobile chips have multiple cores, each of which costs power even when unused. When these cores are utilized, the performance gains more than offset their power costs. For example, increasing parallelization from 1 to 16 threads saved energy for all the applications we tested — even the poorly scaling applications — for an average of 55% energy savings across applications.

Good compilation beats most other energy management techniques (Section 4.4). Performance-oriented optimizations (e.g., gcc's `-O3`) offer significant energy savings, with `-O3` optimized software consuming less than 43% of `-O0` optimized. As for power-oriented optimizations, despite research proposals dating back 20 years [67], modern compilers still do not explicitly optimize for, or significantly impact power.

Java programs require special energy attention, but they don't make it easy (Sections 4.4 and 5.1). Optimizing Java for energy is even more important than optimizing native languages. Not surprisingly, interpreted Java costs nearly 8X the energy of compiled Java. Additionally, prior work has found that Java is particularly prone to source-level inefficiencies, possibly in part from the development tools used to produce it [15]. Despite this, we observed that Java is challenging to manually optimize for energy.

Power-oriented source code optimizations are probably not worth the average programmer's time (Section 5.1). Source-level power tuning suggested by previous research [44] may be effective for tiny embedded programs but is challenging in larger programs. Despite hundreds of micro-optimizations across eight selected benchmarks, we were unable to produce significant power savings for any of the applications.

Idle states are very complementary to other techniques (Sections 4.5 and 5.4). Processor idle or sleep states saved energy — up to 19% — with nearly all of the energy management strategies we combined it with.

Non-complementary conservation strategies can undercut one another by half (Sections 5.3, 4.5 5.4). Not all of the management techniques play well together and their benefits are absolutely not additive. For example, the 19% idle state savings can be cut in half when frequency is tuned to lower levels. However, none of the management strategies interfere so badly that they completely negate another strategy's effects when combined.

2. Background on Energy Management

To set the context for the techniques that this work measures, this section provides a short primer on energy management strategies. For a more complete survey, we refer the reader elsewhere [54, 72, 73, 75]. Although these techniques span many fields of computer science, they all boil down to two broad strategies: *reduce a computation's resource requirements* and *use no more than the required resources*.

Circuit One popular energy conservation technique is to turn off or turn down underutilized components. This is usually accomplished by reducing or stopping the clock and/or supply voltage. An integrated circuit's power consumption is the sum of the active ($P_{active} = \alpha \cdot C \cdot V_{dd}^2 \cdot f$) and leakage ($P_{leak} = V_{dd} \cdot I_{leak}$) power, where α is an activity factor determined by the dynamic switching activity in the circuit, C is the circuit's capacitative load, V_{dd} is the supply voltage, f is the clock frequency, and I_{leak} is the amount of leakage current. *Frequency scaling* reduces the clock for a linear reduction in active power, while *clock gating* stops it entirely. *Power gating* turns off current to idle components, while *dynamic frequency and voltage scaling (DVFS)* reduces supply voltage and frequency together. Targeting supply voltage is particularly effective as it reduces both active and leakage power, the latter of which accounts for up to 50% of total power today [1].

When applied to an idle or near-idle circuit (e.g., a processor executing a memory-bound workload) these techniques

save power while minimally impacting application runtime, ultimately saving energy. The control policies to manage these settings is an active area of research, particularly with respect to emerging integrated voltage regulators [65], which are improving the spatial and temporal resolution of DVFS. These controls are increasingly being exposed to software, however it remains to be seen what type of control policy is best.

Architecture Above the circuit, there is a huge volume of work in energy-oriented microarchitecture including cache tuning [37], on-chip networks [38], memory compression [7], and instruction speculation control [36]. The research community is also embracing heterogeneity in the form of specialized accelerators [23, 28] and asymmetric designs [14] such as ARM's big.LITTLE. Even the now mainstream chip multiprocessors originated out of a need to scale performance without increasing power density, so the software parallelization it forced could be considered part of the power-conservation landscape.

Platform Off-chip, there are numerous other strategies. DC to AC conversion, which consumes 0.9 Watts for every compute Watt [68], is unsurprisingly a focus of datacenter energy efficiency. Cooling, which incurs similar overheads, has also received significant attention (e.g., [51]). On laptops and mobile devices, reducing screen brightness and duty cycling for services such as GPS are other proven energy savers [4, 16].

Operating System Operating systems get involved by explicitly treating energy as another hardware resource to be managed [46, 70]. To save energy, they control software's interactions with lower level resources, for example adjusting DVFS on the fly [47], mapping processes to cores to keep total power below a cap [5, 56], or strategically offloading computation to achieve battery lifetime goals [71].

Compiler and Runtime Via static analysis, feedback directed compilation, or JIT compilation, compilers can analyze applications to insert hints about when to change frequencies [62], rearrange computation to create longer idle periods [3], and place instructions and data into memory in a more energy efficient manner – either by reorganizing instructions in the register file [59] or by creating a compiler-managed scratchpad [32]. There is also research on offloading compilation to a remote machine [42] to save energy and on power-saving hybrid garbage collection schemes [24].

Source and Language At the source level, energy optimization strategies range from micro-optimizations such as manual loop unrolling [19] to macro solutions like updating software development environments to encourage programmers to be more energy friendly [15]. Additionally, language extensions (such as EnerJ, which recruits programmer assistance in finding opportunities for power-accuracy tradeoffs [55]) and new languages (such as Eon, which has programmers identify high and low power energy regions at the source level [63]) have been proposed to improve energy efficiency.

Suite	Applications Used
Parsec 3.0	`blackscholes`*, `bodytrack`, `canneal`, `dedup`, `ferret`, `fluidanimate`*, `raytrace`, `swaptions`, `streamcluster`, `x264`
SPLASH-2X	`barnes`, `fft`, `fmm`, `ocean_cp`*, `radix`* `water_spatial`
Spec CPU 2006	`bzip2`, `gcc`, `mcf`, `hmmer`, `sjeng`, `milc`, `gromacs`, `cactusADM`, `astar`*, `lbm`*, `wrf`, `sphinx3`, `tonto`, `povray`, `GemsFDTD`, `gamess`, `omnetpp`
DaCapo 9.12	`avrora`, `h2`, `jython`, `luindex`, `lusearch`*, `pmd`*, `sunflow`
Spec JBB 2013	`pjbb2005` with 8 warehouses and 100,000 transactions.

* benchmark chosen for application-specific experiments

Table 1: Experimental benchmarks, chosen to represent a range of languages, programming styles, and application domains.

3. Experimental Design and Methodology

Good experimental design and methodology were crucial for this survey. This section describes and justifies the design choices we made.

Experimental System All the experiments in this paper use a single, dedicated Dell PowerEdge R420 server. The server is dual socket with Intel Sandybridge E5-2430 chips, each with six cores and two-way hyper-threading for a total of 24 hardware contexts. The system has 24GB of DRAM and runs Ubuntu 12.04.2 with the 3.9.11 version of the Linux kernel, the latest release at the time of our first data collections. To allow the operating system and userspace to adjust certain controls such as frequency tuning, we switched the Dell BIOS settings to 'operating system control'. The machine runs gcc Version 4.6.3 compiler and Java HotSpot 64-bit server VM with JRE 2, build number 1.5.0.

Power Measurements For all of the power and energy measurements, we use Intel's Running Average Power Limit, or RAPL, interface [29]. RAPL uses non-architectural, model-specific registers (MSRs) that indicate the amount of energy consumed by different parts of the system (e.g., package, cores, DRAM). We sample all the energy counters every 50ms over the course of each program's run and then combine the values to compute total energy. Dividing this value by the total runtime produces the average power during a program's execution.

Benchmark Applications and Inputs Our experiments use 41 benchmarks from five different suites, each commonly used in previous energy management research. The applications represent a breadth of languages, design paradigms, and application domains. Table 1 lists the applications. The first ten come from the Parsec Benchmark Suite [9] which contains multi-threaded programs written in C and C++. We ran each of these programs with the 'simlarge' inputs. The next six applications are from the Splash-2 Benchmark

	Benchmark Suite				
	Parsc	*SpecCPU*	*Splash2X*	*DaCapo*	*SpecJBB*
System (Sec. 4)					
Processor Frequency Tuning	✓	✓	✓	✓	✓
Overclocking (Turbo Boost)	✓	✓	✓	✓	✓
Processor Sleep States	✓	✓	✓	✓	✓
Parallelism	✓		✓		✓
Compiler Opt. Sets	✓	✓	✓		
Interpreted v. Compiled				✓	✓
Application Specific (Sec. 5)					
Source Code Tuning	*	*	*	*	
Per App. Frequencies	✓	✓	✓	✓	✓
Per App. Power Caps	✓	✓	✓	✓	✓

✓= full set of applications, * = select applications only

Table 2: A summary of the energy efficiency techniques explored in this experimental survey.

Suite [76], which are also multi-threaded and written in C. In contrast to Parsec's benchmarks, many of Splash's benchmarks come from high-performance computing and graphics. As prior characterizations demonstrate, these two suites are also fundamentally different with respect to their memory usage and communication patterns [10]. We use the Splash2x variant of the suite that is distributed with the latest version of Parsec in order to have access to the 'simlarge' input sets. The next benchmark suite is SPEC CPU2006 [26], which includes single-threaded, CPU-intensive workloads in C, C++, and Fortran, of which we use 17 benchmarks and the test input sizes. The fourth suite is DaCapo [12], a multi-threaded Java benchmark collection with applications from a variety of real-world domains. We benchmark seven programs using the 'default' input size. Although DaCapo is multi-threaded, it does not allow the user to set the target thread count, so we leave the 'external' thread count setting at one (see the usage documentation [18]) and exclude DaCapo from the parallel experiments. The final benchmark used is SPECjbb2005 [64], which is a client/server system designed to test the performance of Java servers. As packaged, SPECjbb always tries to complete in a fixed amount of time. This makes it hard to compare energy across trials, so we use the pjbb2005 patch [11], a variant of the benchmark that fixes the workload size instead of the runtime. The workload size in pjbb is set via two inputs, a transaction and a warehouse count (see [64] for details). Exploratory experiments on our machine showed the most scalable configuration to be 100,000 transactions and 8 warehouses, so these are the settings we chose. Without constraints, pjbb uses all the hardware threads. To adjust parallelism to a discrete thread count, we used the `taskset` unix command.

Energy Management Technique Selection The energy management techniques cited in Section 2 represent just a fraction of work in the area. To narrow down the large pool, this study focuses on techniques that are software-controllable, as opposed to those that require changes to the underlying architecture, circuitry, or hardware devices. Because hardware energy savings are already well studied (e.g., [20].), it made sense to cut the space this way.

We culled the remaining space by choosing a representative set of techniques that are broadly applicable to a variety of workloads and systems and that span multiple levels of the software stack. We omitted techniques that were infeasible to replicate on our own machine including those requiring complex toolchains, architectural simulation, specialized hardware, or homegrown compiler or operating systems. Table 2 summarizes the nine power management techniques we chose to study. More detailed explanations of the techniques, including pointers to relevant prior work, are presented alongside the experimental results. Section 4 measures the individual and combined effects of six *generic system techniques*: processor frequency scaling, overclocking, use of idle states (all in the OS), compiler optimization flags, interpretation versus compilation, and the effects of parallel thread counts. Section 5 presents the results of three *application-specific* experiments, namely power-oriented source code transformations, per-application processor frequency tuning, and per-application power capping. Section 5 also compares and contrasts application-specific strategies with generic system strategies.

Experimental Rigor Given the breadth of this study, we took particular care to gather accurate, precise, and well organized results. This strengthens our own conclusions and enables other investigators to analyze and build on our data, which we have provided at: `www.arcade.cs.columbia.edu/energy-study`. Automated scripts managed all aspects of the experiment setup, data collection and labelling, thus ensuring repeatability. In addition to the raw energy and runtime data, we gathered supplemental data, such as frequency readings via the i7z tool [33], to confirm that each configuration was successfully applied and implemented as expected. Each benchmark was run a minimum of 20 times at each configuration, and as many times as necessary for the 95% confidence interval to come within 2% of each application's energy, runtime, and power means. In rare cases, this required over 100 program runs of an application for a sin-

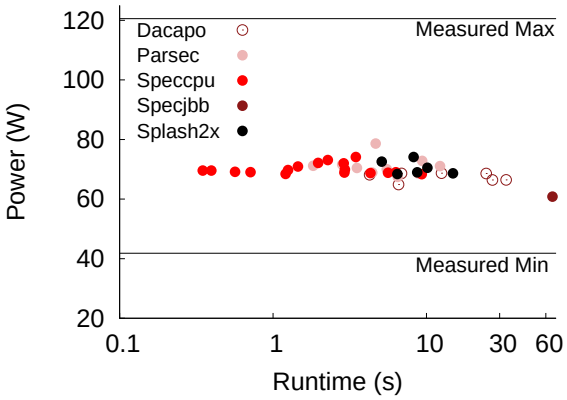

Figure 1: **Baseline Performance and Power.** The 41 benchmark applications exhibited more variation in runtime than in power when run at our *baseline* configuration of a single thread utilizing a processor set to maximum frequency, and with compiler/JVM optimizations and processor idle states all enabled.

gle configuration. In total, the measurements represent over 200,000 application runs across the 220 individual and combined energy management configurations. Using averaging (with geometric means for any pre-normalized data [22]) and normalization we compress this vast amount of data into easy to understand results.

Baseline Power and Performance For clarity and to aid inter-study comparisons, nearly all experimental data is reported relative to a single *baseline* configuration. This baseline, which is our system's default, maximizes processor frequency (2200 MHz), enables Turbo Boost and idle states, maximizes compiler optimizations (`-O3` and `-funroll-loops` for `gcc`, compiled for the JVM), and runs each application with one thread. Getting the Java benchmarks to run with one thread required using the `taskset` command to force the virtual machine onto a single thread. When not taskset, we observed that the Virtual Machine might use any number of hardware threads even if the application is offered only a single thread.

Figure 1 shows the measured runtime and power of the 41 benchmarks on this baseline configuration. Each point on the plot represents the average across as many runs as required to reach our statistical standards. The runtimes (from 0.4 to 66 seconds) showed a greater range than the power consumption (from 61 to 79 Watts). Primarily a result of the range in runtime, energy also ranged widely from 24 to 4036 Joules.

This initial data corroborates existing work from Esmaeilzadeh et al. [20], showing that power is not necessarily related to the thermal-design point, or TDP, of the CPU. While the TDP of our machine is 190 Watts across both sockets, a multithreaded microbenchmark designed to generate large amounts of busywork consumed only 120 Watts. The fact that we never near TDP even at peak system usage could be a symptom of a good cooling system, though this theory has not been tested. We have marked the busywork micromenchmark as "Measured Max" power on Figure 1. We also record a "Measured Min" at 43 Watts, which is is the machine power when nothing other than system utilities and our power profiler were running. Note that this background power is significant, accounting for an average of 60% of the single-threaded benchmark power and for 35% of the multithreaded busywork program.

4. System-Level Results

Here, we present the system-level measurements of frequency tuning, overclocking, processor idle states, parallelism, and compiler flags. We first examine the impact of each setting in isolation and then examine how the five techniques combine. Section 5 presents the remaining application-specific techniques listed in Table 2.

4.1 Frequency Tuning and Overclocking

A huge body of prior work uses dynamic frequency scaling to improve energy efficiency. The key insight is that lower processor frequencies consume less power, so energy can be conserved if processor frequencies are reduced during periods of low work. The challenge of frequency tuning is to figure out when and by how much to reduce frequency without causing performance losses significant enough to negate the power savings. Operating systems are often tasked with this, because they can measure application performance and then reactively set the clock frequency via software exposed registers in the CPU [13]. Linux provides several algorithms, called *cpufreq governors*, to manage this process. The available algorithms depend on the machine architecture and version of Linux, so we measure three commonly available ones:

- The *performance* governor sets frequency to its maximum, 2200 MHz on our test machine. We call this setting **perf w/ Turbo** because, as described below, it also includes Turbo Boosting. It is the baseline described in Section 3.
- The **powersave** governor also uses a constant frequency, but at the system minimum, which is 1200 MHz on our machine.
- The **ondemand** governor increases or decreases frequency, reportedly per processor, when a (tunable) threshold of dynamically measured CPU utilization is reached [47]. We leave all tunables at their default settings, for example leaving the utilization threshold at 95%.

In addition to frequency, power governors also have limited influence on *overclocking*, which means temporarily raising frequency above the processor manufacturers' recommend level for sustained computation. Both Intel and AMD offer dynamic overclocking called Turbo Boost [30] and Turbo CORE [2] respectively. Overclocking may or may

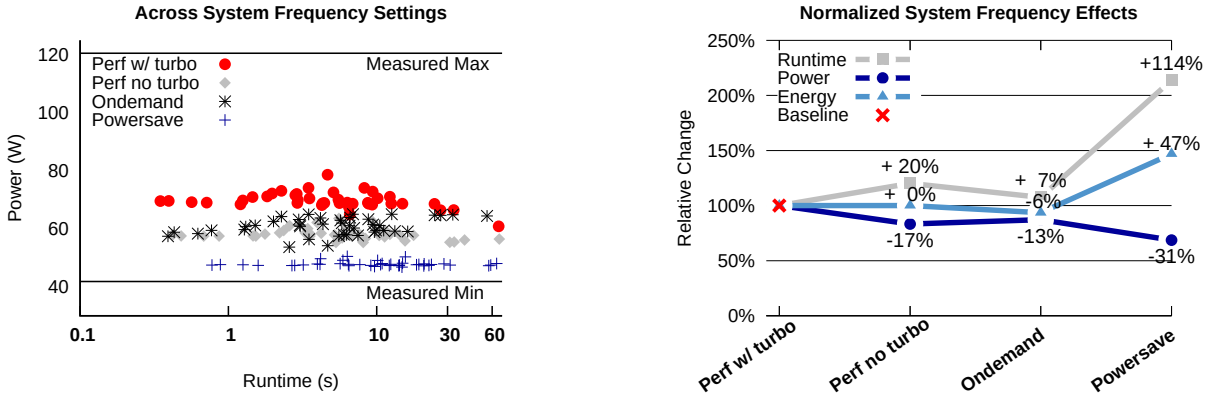

Figure 2: **System frequency tuning algorithms**, such as *ondemand* save at most 6% of energy across applications versus the system baseline of maximum frequency with Turbo Boost enabled (*perf w/ turbo*). Other frequency tuning options include disabling Turbo Boost for decreased runtime but no net energy savings (*perf no turbo*) or a powersave option that saves an average of 31% of the power, but with great costs to runtime (*powersave*).

not have significant bearing on energy; while it reduces compute time, it also causes the system to run hotter and dissipate more power. For safety reasons, hardware has ultimate control over when and for how long overclocking can occur, however, the operating system does have the option to disable overclocking all together. By default, the ondemand and performance governors permit overclocking, which kicks in only when the processor frequency has reached the maximum rating. Using an Intel-supported driver, we were able to create a fourth governor that isolates the effects of Turbo Boosting:

- The **performance no Turbo** governor sets the CPU frequency to its maximum, but disables Turbo Boost (i.e., no dynamic over clocking).

On our system, disabling overclocking for the ondemand algorithm is not an option. Disabling overclocking for the powersave algorithm would not make sense because by the algorithm's definition, frequency is always set to minimum.

Experiments show that these four frequency management strategies yield a range of power-performance tradeoffs. The left panel of Figure 2 plots the individual application runtimes and power consumption at each of these four settings, while the right panel summarizes the impact of these settings across all applications.

Disabling Turbo Boost and removing the machine's ability to ramp up frequency for short periods of time resulted in a runtime *increase* of 20% across applications. We did not monitor the frequency changes across all of our experiments, but observed using the i7z tool [33] that Turbo Boost almost always increases frequency (up to 2700 MHz, or 500 MHz above the normal maximum frequency) when a single procesor is working at 100% utilization but the remaining processes are idle, as was the case for most of the experiments in Figure 2. In many cases, the frequency was allowed to remain at 2600-2700 Mhz for the duration of the application's execution provided the other cores remained idle, which explains the significant performance differential.

Conversely, disabling Turbo Boost *decreased* power by an average of 17% across applications, a direct consequence of the lower average processing frequency. The nearly equivalent increase in runtime and decrease in power meant that across applications, disabling Turbo Boost produced no net change in energy versus Turbo Boost enabled. Individual applications saw some minor energy shifts with Turbo Boost disabled versus enabled: at most a 9% increase and a 6% decrease with 17 applications increasing in energy consumption and 24 decreasing.

Similarly, the out-of-the-box ondemand algorithm affects energy by only a small amount, with 6% average savings across applications. Most of the individual applications (38 out of 41) saved a little energy, but only six saved more than 10% relative to the baseline. This limited savings may be unsurprising to some in the operating systems community, who have questioned the efficacy of the ondemand frequency tuning algorithm [39] as well as frequency tuning's potential to save energy at all on modern processors [40]. Powersave is a big energy loser, with an average increase of 47% versus the baseline and with not a single individual application saving energy. From these results, it is clear that powersave, or any similar strategy that reduces frequencies to a minimum, is not a desirable policy for active processors.

4.2 Idle States

Most computers spend a significant amount of time underutilized, for example while serving I/O. Datacenter servers reportedly use only 10-50% of their processors at a time; the remainder are idling [6]. Idleness can be costly in terms of power draw with under-utilized servers still drawing more than 50% of their peak power [6]. In recent years processor vendors have offered a rich menu of processor idle states, that send the processor to increasingly deep levels of 'sleep'

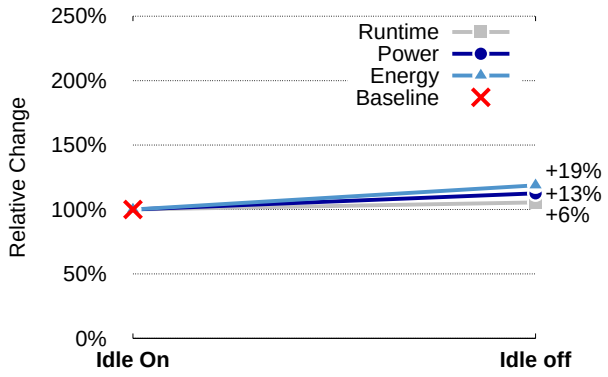

Figure 3: **Processor idle states** enable 19% energy savings relative to the mode that prevents cores from entering these power-saving sleep modes.

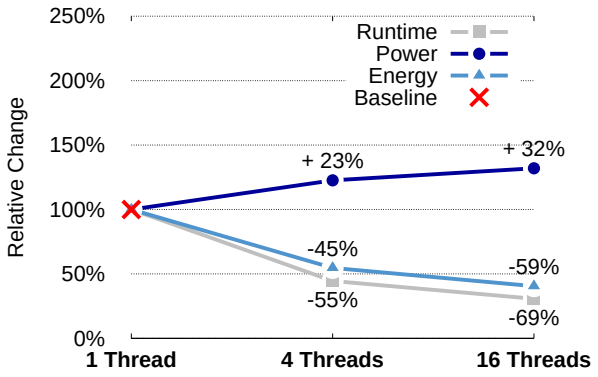

Figure 4: **Parallelization** increases energy savings for all applications tested. For our 12 core, 24 hyperthread server, running 16 application threads consumed just 45% of the energy of the serial execution.

for increasing power savings. The specifics vary from vendor to vendor, but as an example, a first level of sleep might be to stop the CPU clocks, a second to turn down CPU voltage, and a third to reduce the voltage further and stop refreshing cache [69]. The reason for multiple levels of idleness, sometimes called *c-states*, is that each deepening state comes at an added transition cost, taking increasingly more time for the processor to switch back to active. If a processor is sent into a deep idle state immediately before an application requests its resources, the application will experience runtime delays. Thus, the main challenge to managing idle states is to figure out when to idle, how deeply to idle, and when to wake up.

As with frequency scaling, the operating system has been tasked with observing application behavior and managing idle states accordingly. Linux provides a *cpuidle* idle state manager [48], which is analogous to its *cpufreq* frequency algorithms. The cpuidle manager monitors the dynamic use of all the system processors and uses this information to determine the appropriate depth of sleep. It is also possible to force a processor to use a specific idle state (see instructions in [25]) rather than allowing the automated manager to control sleep depth. According to the documentation [25], manual settings are helpful for reducing system latency but not likely to save more power than the *cpuidle* manager, so we limit our experiments to the managed algorithm rather than manual settings. This narrows our idle state exploration to just two settings:

- The **idle on** data is measured with the *perf w/ turbo* frequency tuning, per application thread count of one, and `gcc-O3` or the default `javacc` options (i.e., the baseline).
- The **idle off** is the same configuration but with cpuidle disabled (i.e., the cores are not allowed to sleep).

Figure 3 plots the comparison, which reveals a 19% energy difference between idle on and off across all 41 applications. For individual applications, the differences range from 11 to 25%, with all applications seeing a net energy decrease when idle states are enabled. Also in line with expectations, power is on average 13% higher when idle states are turned off, at most 24%, and at least 8%. Unexpectedly, all of the applications run faster by an average of 6% when idle states are enabled. We found that this runtime difference reverses when Turbo Boost is disabled, and we suspect that with idle states enabled, the core used by the single-threaded application is able to take advantage of the lower overall system power and turn on Turbo Boost more frequently than when idle states are disabled, resulting in the shorter runtimes.

4.3 Parallelism

Although the idea of parallelism has been around since the first computers [74], multicores became mainstream roughly a decade ago, when AMD and Intel started selling dual core processors for desktops. This revolution was driven largely by energy and power concerns. The increasing clock speeds and transistor counts that drove performance higher for fifty years also drove power density to unsustainable levels. Computer architects reacted by simplifying processor cores and offering more of them, which kept heat levels under control while allowing performance to continue to grow. The catch is that to effectively increase performance, software must actually *use multiple cores*.

As core counts grow, software engineers are left to deal with the difficult challenges of writing well parallelized code to improve performance on future generations of chips [43]. One could argue (as Urz Holzle, SVP at Google, did [27]) that it is the responsibility of computer architects to keep serial processing efficient so that software is not forced into parallelism. However, for better or worse, chip-multiprocessors now dominate the desktop market, and core counts in the mobile market are also creeping up [61]. In addition to runtime efficiency, prior research has shown that

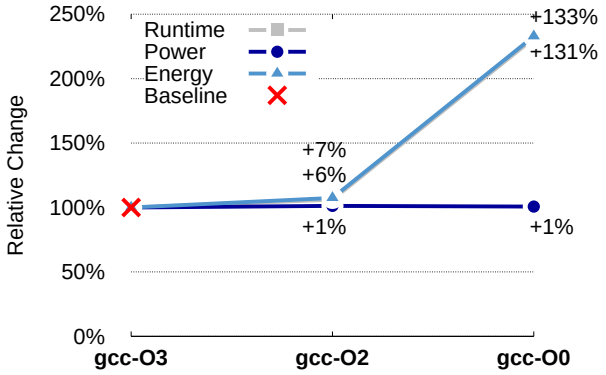

Figure 5: **Standard compiler optimization sets** save energy, but largely through runtime reductions not power reductions. Applications without optimization take 133% more energy and 131% more time than fully optimized applications.

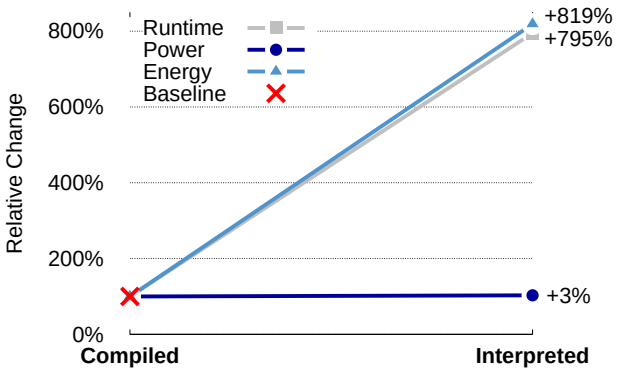

Figure 6: **Java compilation** saves substantial energy versus interpreted code, which consumes 8X the energy, but again these savings are due to runtime, not power.

energy efficiency is similarly reliant on effective parallelization [52], so energy conscious programmers must also deal with this reality.

We measured the interplay of performance, energy, and parallelism on our 12 core, 24 hyperthread machine; Figure 4 summarizes the results. The baseline is the same as before, with the data showing average changes in runtime, power, and energy for 4-thread and 16-thread program runs versus single-threaded runs. The results are averaged across multiple runs of the 17 benchmarks from the Parsec, Splash2x, and SPECjbb suites, the three suites that supported discrete thread count settings. From the runtime values, it is evident that some of the applications scale poorly: on average the applications show only a 2X speedup over serial with four threads, and a 3X times speedup with 16 threads. The most scalable application tested, `radix`, saw only an 8X speedup at 16 threads. Jumping from 4 to 16 threads caused `radix`'s power to increase by 50%, thanks to increased core activity reducing the opportunity to exploit idle states. This power increase tempered the runtime savings, so that `radix`'s energy at 16 threads was about 20% of its single-threaded energy. In other applications, a similar phenomenon occurred: speedups provided by added parallelism were offset by the power increases resulting from more concurrently active threads. However the power increases *did not* exceed the runtime savings for any of the applications we tested, meaning all of the applications saved energy. Even the most poorly scaling application, `raytrace`, whose runtime at 16 threads decreased only 19% versus one thread saved a non-negligible amount of energy at 13%. On average, the applications saved 55% of the single-threaded energy when run with 16 threads.

4.4 Compiler optimizations

Most existing work on energy efficient compilation focuses on the power and energy impacts of performance optimizations. They typically find that these optimizations reduce runtime much more than they increase power, resulting in a net decrease in energy. In attempts to isolate which optimizations are the most power efficient, a number of studies apply individual optimizations such as function in-lining, loop unrolling, and loop vectorization to benchmarks (e.g., [58]). Other research has constructed new optimization sets for energy rather than performance (e.g., [49]). The conclusion of all prior studies seems to be the same: when it comes to compilation, what is best for performance is best for energy.

This is not a surprising conclusion when the optimizations tested affect performance almost exclusively (and not power). The community has proposed a few power-optimizations, such as reordering instructions or memory operands or reassigning registers to reduce control path switching. A good overview of these techniques is presented in a 1994 paper by Tiwari et al. [67]. Twenty years later it seems none of these techniques have made it into mainstream compilers, so quantitative data on their ability to improve power and energy is sparse. Given the lack of power-specific optimizations in commercial compilers, we measure the energy effects of standard sets of compiler optimizations. While we are far from the first to take these measurements, we include them to provide a quantitative comparison point for the other measurements in this paper.

Figure 5 shows the energy effects of the standard `gcc` compiler optimization sets for applications in the three native benchmark suites. On average, the applications ran in 131% less time for `gcc-O3` versus `gcc-O0`, meaning the optimized code took 43% of the time of the unoptimized code to run. The change in power was negligible, about 1% on average, so the energy effects track the runtime, with `gcc-O0` taking 233% of `gcc-O3`'s energy. The per application energy savings of turning on optimizations ranged wildly, from less than 1% to nearly 700%, likely a reflection of how optimized the original source code was. In terms of energy and runtime

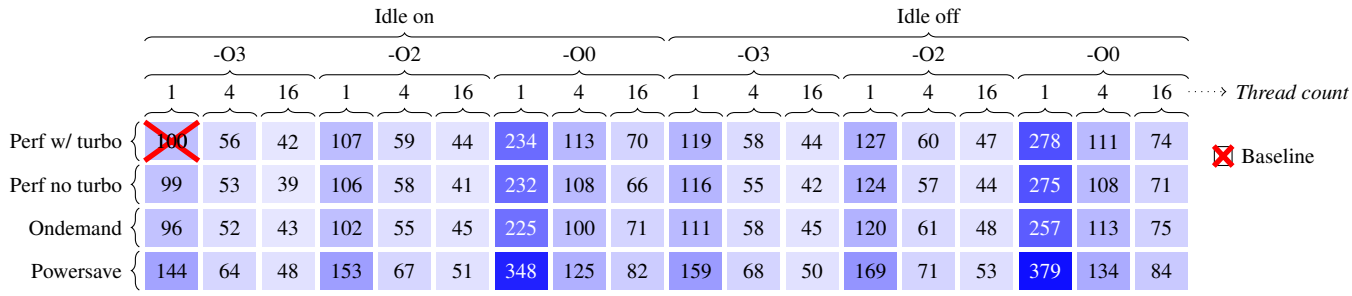

Figure 7: **Energy effects of combining multiple configurations.** This table shows cross-level energy interactions of the five energy configurations discussed so far, as a percentage of the baseline, (i.e., baseline = 100%). Note that the matrix includes data from only the parallel, native benchmark suites: Parsec, Splash2x.

the `-O2` optimizations were very similar to `-O3` on average, with 8 of the 33 applications actually saving more energy with `-O2` than with `-O3`. These numbers emphasize compilers' important contributions to energy savings, but confirm that all the savings come in the form of reduced runtime.

For the eight Java benchmarks, we measured the energy of interpreting rather than compiling. On average, the cost of interpreting was huge, consuming 818% more energy on average than compilation, which is roughly in line with the runtime impact of 795%. Again, the energy changes varied between applications, from an energy savings of 23% for `pmd` (the only application to save energy, and purely a result of runtime savings), to an increase of over 2600% for `sunflow`. Average power increases were barely significant at just 3%, and varied less between applications (-0.5% to 6%).

4.5 Cross-Layer Energy Effects

The preceding sections explore the impact of each optimization in isolation. We wanted to know whether turning on multiple techniques resulted in additive, negative, or synergistic interference, so we ran experiments that combine all of the techniques presented so far. The heatmap matrix in Figure 7 shows all of these combinatorial effects as a percentage of the baseline. The data in the matrix comes from the 16 applications in the two parallelizable, native benchmark suites, Splash2x and Parsec. The rows represent different system frequency algorithms, while the columns cover all of the idle states, compiler options, and parallelism configurations previously discussed.

A number of insights could be drawn from these comparative experiments. Most notably, the energy savings of one strategy can be cut by half depending on what other strategies are in use (e.g., enabling idle states saves 19% at the baseline frequency, but only 10.4% when the powersave algorithm is used.) Similarly, the ondemand frequency algorithm saves less energy at 16 threads than with one thread. In fact, at 16 threads, ondemand actually increases energy regardless of compiler optimization or idle state configuration. Compiler optimizations follow this pattern as well, saving less energy at 16 threads (about 40% across configurations) than at one thread (57%).

Several techniques were a win across the board. For all 18 configurations, disabling Turbo Boost saved energy because the runtime savings from Turbo Boost's increased frequency were more than offset by corresponding power increases. However, unlike the other techniques, disabling Turbo Boost saves *more* energy for 16 threaded trials than serial trials. Idle states also provided nearly universal energy savings, with 34 out of 36 configurations showing energy decreases when they were enabled. Increasing parallelism, even without perfect performance scaling, was also a relatively large energy winner, with energy decreasing from 1 to 4 to 16 threads for all configurations.

Ultimately, the best energy configuration was with idle states on, the `-O3` optimization set, 16 threads, and the performance no turbo frequency tuning. Note that this does not match our baseline and the system default, which enables Turbo Boost. The worst configuration was essentially the opposite: `-O0`, idle off, one thread, and the powersave algorithm. The difference between these two configurations is a whopping 10.3X.

5. Application-Level Energy Management

This section presents the measurements of three application-specific energy management techniques: source code tuning, custom frequency scaling, and power capping. It then links these techniques to system-level techniques in a combination study.

5.1 Source Code Tuning

A number of recent and older works suggest that optimizing source code for power savings can have significant impact [8, 15, 44, 67]. Surveying these works and others, we found eight kinds of source-level transformations purported to save power or energy, and applied these transformations to the eight benchmarks marked with a * in Table 1. The transformations aim to:

1. reduce temporary variables,
2. eliminate common subexpressions (e.g. consolidate duplicate computations or lookups in complex structures),
3. postpone variable declarations until needed,
4. use operator= instead of the operator alone and use prefix instead of postfix operators (but only for complex types),
5. use direct assignments of variables rather than initializations followed by assignment,
6. replace multiply and divide operations with shifts or addition when possible,
7. optimize loops with unrolling and unswitching (moving a conditional from inside to outside of a loop)
8. reduce the number of arguments passed to functions.

To focus our efforts, we looked for opportunities to apply the first six optimizations within loops or within functions called inside loops. In total, we made 688 changes across the eight applications, ranging from 5 to 292 changes per application. Figure 8 shows the exact number of changes per application, in square brackets above each triplet of bars. It was simpler to make changes in the less optimized applications of the Splash2x and Parsec benchmarks, and conversely more difficult to improve the already-optimized SPECCPU benchmarks. The DaCapo benchmarks were especially challenging to transform. This is partly because they are already well optimized, and partly because it is difficult to track the scope of objects whose instantiation may be far removed from use, as opposed to variables in native benchmarks, whose scope often lasts only one function. When the scope of an object was unclear, it was necessary to be more conservative about deletion or modification.

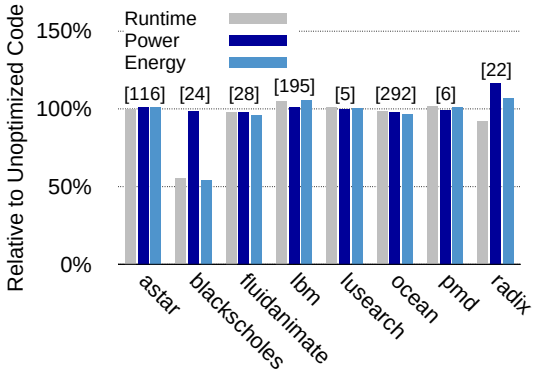

Figure 8: **Source code tuning** methods from prior embedded systems research were not very effective energy savers for for our complex and already well-optimized benchmarks running on servers.

Figure 8 shows the power, performance, and energy effects of our transformations relative to the unoptimized programs. The data represents multiple trials, all utilizing the same baseline as previous experiments, with `gcc` optimization level `O3` and the `-funroll-loops` options enabled for native programs, and the compiled virtual machine used for the Java programs. Only one application saw a significant reduction in energy — `blackscholes` — while four others' energy was slightly reduced by our transformations. The effective transformations in `blackscholes` were common subexpression elimination, the reduction of temporary variables, and direct assignment, all within a 'hot' function, and missed by the optimizing compiler due to the complex objects involved in the computation. The transformations reduced power for five of the eight applications, but none of these measured reductions were outside of our 2% confidence interval range, so they should be considered statistically insignificant. One application, `radix`, experienced a significant power increase at 17%, likely due to additional loop unrolls that the compiler would normally not perform.

This study could be considered a failure given that the optimizations did not result in significant power or energy savings, but we still felt it important to include the negative results. They demonstrate that, at least for previously optimized applications run on servers, micro-optimizations for power and energy are challenging. Given that the results were poor and the source-level transformations require disproportionately more effort than other energy saving techniques, we suggest that *power-specific source transformations are not worth the average programmer's time once the code has been optimized for performance*.

5.2 Application Tuned Frequencies

As previously shown, the ondemand frequency governor provides only small energy savings. In part this is because it is conservative (optimizing for performance) and reactive (waiting to measure processor utilization before adjusting frequency). We also observed (using the i7z frequency monitoring tool [33]) that even when only one core is utilized by an application, ondemand tends to unnecessarily ramp up the frequency of the entire socket. All of these behaviors limit ondemand's ability to conserve energy.

A number of researchers have noticed that reactive measurements coupled with the high latencies of switching frequencies through the OS may result in less than optimal frequency tuning, and have proposed alternate methods. For example, a recent paper by Rangan et al. [53] proposes setting individual core frequencies to different static values, then migrating application threads to improve both energy and throughput. Hints from compilers [77], static analysis tools [60], and even software developers [63] have also been proposed to tune frequencies more effectively. None of these papers distribute open-source code, so in lieu of reimplementing their work, we contextualize it by running individual applications at discrete, constant frequency levels. This obviously does not replicate techniques that continually switch applications between frequency levels, but it at least gives us an idea of the range of power-performance tradeoffs involved in application-specific frequency tuning.

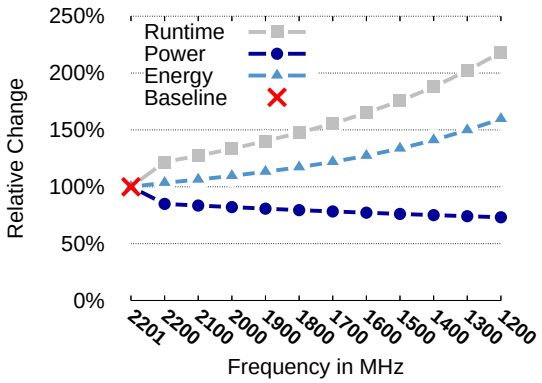

Figure 9: **Application-specific frequency tuning**, or running an application at a single discrete frequency, allows power-performance tradeoffs to be flexibly manipulated.

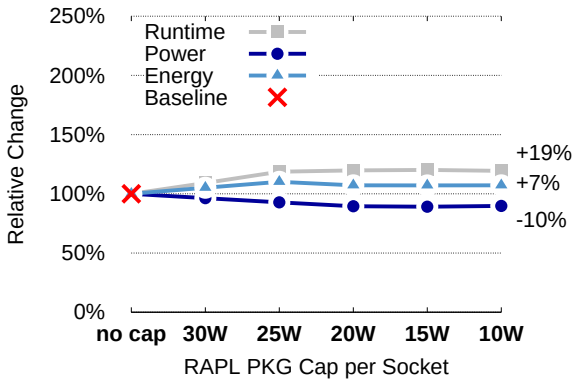

Figure 10: **RAPL power caps**, which limit the amount of power a part of the chip is allowed to consume over a given time window, yield a more limited power-performance tradeoff range.

Linux provides a mechanism for root to change individual processor frequencies through a *userspace* governor. On the machine used for these experiments, frequencies can be set to 11 distinct levels, from 1200 MHz through 2200 MHz at 100 MHz steps. A twelfth option is to set the frequency to maximum (2200 MHz) and enable Turbo Boost. Figure 9 shows how all 41 benchmarks perform, on average, at different frequency levels. So that all the applications could be used, these results show single-threaded runs: the unused 11 cores (22 hyperthreads) were set to minimum frequency while the occupied processor's frequency was varied. Idle states are turned on for all of these experiments. In the average case, none of the frequency configurations saves energy relative to 2200 MHz with Turbo Boost; instead, there is a smooth power-performance tradeoff curve with runtime increases always slightly over-shadowing power decreases. Looking at individual applications, three of the 41 save a negligible amount of energy when Turbo Boost is disabled but frequency remains set to 2200 MHz. Moving down the frequency scale to 2100 MHz none of the applications save any significant amounts of energy. Power decreases are relatively uniform across applications, sinking a little more at each frequency. The corresponding runtime increases, however, vary significantly between applications. For example, at 1700 MHz, runtime may increase as little as 38% or as much as 74% relative to the baseline, resulting in relative energy losses of 9 to 35%. Future algorithms should be sure to account for this highly application-specific response to frequency tuning.

5.3 Per Application Power Caps

While hardware ensures that on-chip power levels do not exceed the TDP, sometimes there is a need to cap power at a lower level. For example, in datacenters, enforcing a strict power cap somewhere below the TDP could make energy expenses more predictable and affordable. Several research projects have addressed this need via power-attentive thread to core scheduling and DVFS [17, 31, 56]. A couple of industrial tools exist as well, for example, Intel's RAPL Power Caps [29]. Since our machine contains Intel processors, we experiment with this particular implementation.

RAPL allows a user with sufficient privileges to limit power across multiple domains per socket: *power plane 0* which includes cores and private caches, *power plane 1* which includes alternate processing units such as GPUs, the *package* which includes both power planes as well as shared caches, and finally *DRAM*. The user selects a domain to cap, then gives the RAPL interface a specific power value to limit that domain, as well as a time window. The time window specifies periods during which average power levels must meet the cap. For example, if the given window is 100ms and the cap is 30W, RAPL promises that every 100ms, the average power of the specified domain will not exceed 30W. RAPL documentation is unclear about how these power limits are maintained, but our reverse engineering shows that frequency scaling is at least part of their strategy. RAPL capping overrides system frequency algorithms, but does allow idle states to be enabled.

Figure 10 shows the results of capping both sockets' package power at various levels. Initial experiments showed that useful package capping values fell between 30 and 10 Watts. Only the 8 applications marked with a * in Table 1 were used for these results. Across applications, capping traded modest (up to 11%) decreases in power for modest but slightly larger (up to 20%) increases in runtime, resulting in a slight net energy *increase*. The graph indicates that the power increases and runtime decreases were not quite monotonic as power caps were lowered, but the slight up and down fluctuations are less than our error range at 1%, and thus should not be considered statistically significant.

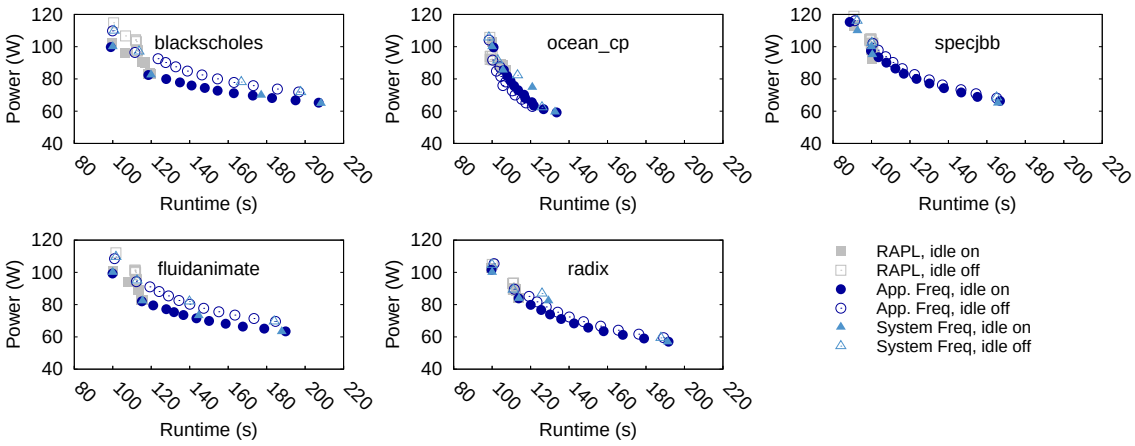

Figure 11: **Application-specific strategies versus system level strategies for frequency tuning.** RAPL caps, application-specific frequency tuning, and system frequency governors could not be combined with each other, so we compared their power performance effects instead. All three could be combined with idle states, however, which when enabled saved energy across all of the different frequency configurations.

5.4 Comparing Application-Specific to System-Level Strategies

We wrap up our study by showing how system-level strategies (frequency governors and idle states) compare with application-level techniques (application-specific frequency tuning and RAPL caps). The frequency governors, application-specific frequencies, and RAPL caps all control the same knob of processor frequency, so these three strategies cannot be combined and should be considered mutually exclusive. However, idle states can be combined with all three forms of frequency tuning, and we noticed that toggling Turbo Boost on or off has some effect on RAPL capping performance. For the five parallelizable applications marked with * in Table 1, we set the compiler optimization level to `-O3`, and the thread counts to 16 (note this deviation from the baseline of one thread; we return to unnormalized data here) then ran multiple trials of the following configurations:

- 6 levels of RAPL caps $\times$ 2 idle settings $\times$ 2 turbo settings
- 12 application-specific frequency values $\times$ 2 idle settings
- 4 system frequency algorithms $\times$ 2 idle settings

This comes to a total of 56 configurations per application, each of which is plotted in the power-performance graphs of Figure 11. A few insights immediately jump out. First, the fastest configurations tend to take the most power, and overall, the majority of configurations seem to make a strict trade of increased runtime for decreases in power. Second, `ocean_cp` trades off much less performance for power savings than the other three applications. Third, regardless of the configuration, turning idle states on has a positive effect on power. Though it may not jump out of the plots immediately, idle states also save energy for the majority of configurations.

Relative to the system default (i.e., the baseline), the number of configurations that save energy varies significantly per application. For example, 50 of `ocean_cp`'s 56 configurations save energy, with the best saving 24%. However, for `fluidanimate`, just 12 configurations save energy with the best saving only 5%. The lowest energy configuration for each application varies as well: for `ocean_cp` the best is to turn off idle states and minimize frequency, while for `fluidanimate` and `blackscholes` it is idle states on and maximized frequency, and for both `radix` and `SPECjbb` it is idle states on with RAPL caps set to 10W and no Turbo Boost. These results show that while the power savings trends of each strategy may hold across applications, the performance trade-offs can vary, resulting in unpredictable energy effects.

6. Related Work

In the most comprehensive prior study of energy efficiency techniques, Esmaeilzadeh et al. [20] examined the power-performance tradeoffs of different microarchitectural features including clock frequencies, memory hierarchy configuration, and hardware parallelism. While the two studies overlap in some dimensions (parallelism, frequency tuning), ours explores a wider variety of software techniques, for the first time allowing direct quantitative comparisons between energy efficiency solutions at different layers of the stack.

Several other studies also compare multiple hardware-level energy efficiency techniques. Patki et al. take an HPC perspective, examining how overprovisioning techniques (such as overclocking) and power capping can help improve supercomputers' efficiency [50]. Subramaniam and Feng combine RAPL capping with a variety of server load inputs to see how well RAPL can provide energy proportionality (i.e., similarly efficient execution for different levels of server utilization) [66]. Le Sueur and Heiser examine the effects of DVFS and idle states across multiple processor generations [40], finding that newer processors see smaller

energy benefits from frequency scaling. None of these works compare the hardware-level techniques to higher level software techniques as we do.

Like ours, Schone et al.'s experiment space includes processor level DVFS, different degrees of parallelism, and overclocking [57], but their study measures the impact of these techniques on memory and last level cache bandwidth only. The relative energy savings of multiple software-level techniques are compared in just a few prior works. Most are either qualitative (e.g., [21]), or focused exclusively on compiler or application-level energy management strategies [34, 45]), without linking those techniques to the system-level as this study has.

7. Discussion and Conclusions

Energy management has become a large field in recent years, with work spanning all levels of the stack. Unfortunately, the broad interest in energy-efficiency has caused fragmentation: most management strategies are not compared against each other – especially those at different levels of the stack – and most research papers do not quantitatively or even qualitatively address how their work will combine with existing strategies.

As a first step to bridging these discontinuities, this experimental survey directly compared and combined nine existing but previously uncontrasted energy management strategies. The work prompts a number of suggestions and directions for future energy research, particularly for software-controlled energy management.

Suggestions for the community. The research and industrial community should take concerted steps to reduce the amount of fragmentation in future work. First, research papers should compare their work to commonly available baselines. For example, OS DVFS strategies might compare their energy savings to the widely accessible Linux frequency scaling governors. Second, researchers should attempt to combine their energy saving strategies with existing energy strategies to uncover interference effects. Failing that, they should qualitatively discuss their work in the context of existing work, noting which strategies could combine with their work versus which strategies it is meant to compete with. Future work should address inter-application differences; we found that particularly with respect to performance tradeoffs, individual applications varied significantly. Future work should also consider parallelism, given that our work showed parallelization in particular has a large effect on other energy management strategies, and given that CMPs are the new normal. To help others contextualize their work, researchers should try to open source their energy management tools, preferably with as standard a setup as possible (i.e., commodity languages, compilers, OS, and architectures). Finally, outside of research, we need continued and increased communication with industry. Conversations with several industrial computer scientists lead us to believe that industrial energy management is in some ways more advanced and less fragmented than public research. If existing industry knowledge is not adequately shared, public domain research will continue to chase and potentially clash with industrial progress.

Research areas in need of attention. Our survey highlighted several promising paths forward for software-level energy research. Development aids that help programmers write energy efficient code could overcome multiple issues including the difficulty of manual power optimization (Section 5.1), the needs of object-oriented programs [8], and the inflation in energy caused by IDE programming [15]. There is also a need for power aware compiler optimizations, since present optimizations have minimal effect on power. Machine-specific power optimizations may be especially timely, with mobile devices like Android moving to ahead-of-time byte code compilation [41]. Our work indicates that increasing parallelism as a solution for energy efficiency is another promising research direction. The proliferation of CMPs also means that the wide scale effects of inter-application power interference should be studied, perhaps using methods analogous to existing application performance interference measurements [35]. Finally, the majority of current energy solutions exploit performance tradeoffs. To exploit others, such as trading program functionality for energy and power savings, there is a need for library or language-level energy management tools.

8. Acknowledgements

This research was supported by grants from the National Science Foundation (CCF-1253772) and DARPA (PERFECT Program).

References

[1] The intenternational technlogy roadmap for semiconductors, 2009. http:public.itrs.net/.

[2] AMD. AMD Phenom™II key architectural features. http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx.

[3] J. Ayala, A. Veidenbaum, and M. Lpez-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Programming*, 31(6), 2003.

[4] N. Banerjee, A. Rahmati, M. D. Corner, S. Rollins, and L. Zhong. Ubiquitous Computing. *Lecture Notes in Computer Science*, 4717, 2007.

[5] M. Banikazemi, D. Poff, and B. Abali. PAM: A novel performance/power aware meta-scheduler for multi-core systems. In *International Conference on Supercomputing (SC)*, 2008.

[6] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[7] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems

with embedded processors. In *Design, Automation, and Test in Europe (DATE)*, 2002.

[8] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta. Does lean imply green?: A study of the power performance implications of java runtime bloat. In *ACM SIGMETRICS*, 2012.

[9] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *International Symposium on Workload Characterization (IISWC)*, 2008.

[11] S. Blackburn, M. Hirzel, R. Garner, and D. Stefanovic. pjbb2005, 2006. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005.

[12] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Annual Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2006.

[13] D. Brodowski and N. Golde. CPU frequency and voltage scaling code in the Linux kernel: CPUFreq governors. http://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[14] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[15] E. Capra, C. Francalanci, and S. A. Slaughter. Is software "green"? application development environments and energy efficiency in open source applications. *Information and Software Technology*, 54(1), 2012.

[16] Y. Chon, E. Talipov, H. Shin, and H. Cha. Mobility prediction-based smartphone energy optimization for everyday location monitoring. In *Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

[17] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[18] DaCapo Project. The DaCapo benchmark suite usage documentation, 2009. http://www.dacapobench.org/.

[19] V. Dalal and C. P. Ravikumar. Software power optimizations in an embedded system. In *International Conference on VLSI Design*, 2001.

[20] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[21] F. Fakhar, B. Javed, R. ur Rasool, O. Malik, and K. Zulfiqar. Software level green computing for large scale systems. *Journal of Cloud Computing*, 1(1), 2012.

[22] P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3), 1986.

[23] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[24] P. Griffin, W. Srisa-an, and J. M. Chang. An energy efficient garbage collector for java embedded devices. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

[25] S. Hayes. Controlling processor C-State usage in Linux, 2013. http://en.community.dell.com/cfs-file.ashx/__key/telligent-evolution-components-attachments/13-4491-00-00-20-22-77-64/Controlling_5F00_Processor_5F00_C_2D00_State_5F00_Usage_5F00_in_5F00_Linux_5F00_v1.1_5F00_Nov2013.pdf.

[26] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.

[27] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[28] S. Hong and H. Kim. An integrated GPU power and performance model. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[29] Intel Corporation. Intel 64®and IA-32 architectures software developer's manual. http://download.intel.com/products/processor/manual/253669.pdf.

[30] Intel Corporation. Intel® Turbo Boost Technology 2.0, 2014. http://www.intel.com/technology/turboboost/.

[31] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *International Symposium on Computer Architecture (ISCA)*, 2006.

[32] Y. Ishitobi, T. Ishihara, and H. Yasuura. Code and data placement for embedded processors with scratchpad and cache memories. *Journal of Signal Processing Systems*, 60(2), 2010.

[33] A. Jaiantilal. i7z, 2013. http://code.google.com/p/i7z/.

[34] R. Jain, D. Molnar, and Z. Ramzan. Towards understanding algorithmic factors affecting energy consumption: Switching complexity, randomness, and preliminary experiments. In *Joint Workshop on Foundations of Mobile Computing*, 2005.

[35] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *International Conference on Supercomputing (SC)*, 2012.

[36] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[37] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Annual International Symposium on Microarchitecture (MICRO)*, 1997.

[38] N. Kirman and J. F. Martínez. A power-efficient all-optical on-chip interconnect using wavelength-based oblivious rout-

ing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[39] M. Larabel. Benchmarking the Intel P-State, CPUfreq changes. Phoronix Media, 2013. http://www.phoronix.com/scan.php?page=news_item&px=MTM3NTI.

[40] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Conference on Power-Aware Computing and Systems (HotPower)*, 2010.

[41] J. Levi. Dalvik vs. ART: Android virtual machines and the battle for better performance, 2013. http://pocketnow.com/2013/11/13/dalvik-vs-art.

[42] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[43] A. Marowka. Back to thin-core massively parallel processors. *Computer*, 44(12), 2011.

[44] H. Mehta, R. Owens, M. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Low Power Electronics and Design*, 1997.

[45] K. Naik and D. S. L. Wei. Software implementation strategies for power-conscious systems. *Mobile Networks and Applications*, 6(3), 2001.

[46] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2001.

[47] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Linux Symposium*, volume 2, 2006.

[48] V. Pallipadi, S. Li, and A. Belay. cpuidle: Do nothing, efficiently. In *Linux Symposium*, volume 2, 2007.

[49] J. Pallister, S. J. Hollis, and J. Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 2013.

[50] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *International Conference on Supercomputing (ICS)*, 2013.

[51] M. Patterson. The effect of data center temperature on energy efficiency. In *Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM)*, 2008.

[52] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. Computational sprinting. In *Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[53] K. Rangan, G. Wei, and D. Brooks. Thread motion: Fine-grained power management for multi-core systems. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[54] P. Ranganathan. Recipe for efficiency: principles of power-aware computing. *Communications of the ACM (CACM)*, 53 (4):60–67, 2010.

[55] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[56] H. Sasaki, S. Imamura, and K. Inoue. Coordinated power-performance optimization in manycores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[57] R. Schöne, D. Hackenberg, and D. Molka. Memory performance at reduced CPU clock speeds: An analysis of current x86 64 processors. In *Conference on Power-Aware Computing and Systems (HotPower)*, 2012.

[58] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. In *Workshop on Interaction Between Compilers and Computer Architectures*, 2003.

[59] D. She, Y. He, B. Mesman, and H. Corporaal. Scheduling for register file energy minimization in explicit datapath architectures. In *Design, Automation, and Test in Europe (DATE)*, 2012.

[60] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Design Automation Conference (DAC)*, 2001.

[61] E. Slivka. Apple's A8 chip production for iPhone 6 underway at TSMC, 2014. http://www.macrumors.com/2014/03/05/a8-chip-underway-tsmc/.

[62] S. W. Son, G. Chen, O. Ozturk, M. Kandemir, and A. Choudhary. Compiler-directed energy optimization for parallel disk based systems. *Parallel and Distributed Systems*, 18(9), 2007.

[63] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[64] Standard Performance Evaluation Corporation. SPECjbb2005, 2013. http://www.spec.org/jbb2005/.

[65] N. Sturcken, M. Petracca, S. Warren, P. Mantovani, L. Carloni, A. Peterchev, and K. Shepard. A 2.5D integrated voltage regulator using coupled magnetic core inductors on silicon interposer delivering 10.8A/mm^2. In *International Solid-State Circuits Conference (ISSCC)*, 2012.

[66] B. Subramaniam and W.-c. Feng. Towards energy-proportional computing for enterprise-class server workloads. In *International Conference on Performance Engineering (ICPE)*, 2013.

[67] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Low Power Electronics*, 1994.

[68] M. Ton, B. Fortenbery, and W. Tschudi. DC power for improved data center efficiency. 2008.

[69] G. Torres. Everything you need to know about the CPU c-states power saving modes, 2008. http://www.hardwaresecrets.com/article/611.

[70] A. Vahdat, A. Lebeck, and C. S. Ellis. Every joule is precious: the case for revisiting operating system design for energy efficiency. In *ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, 2000.

[71] N. Vallina-Rodriguez and J. Crowcroft. ErdOS: achieving energy savings in mobile OS. In *International Workshop on MobiArch*, 2011.

[72] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, PP(99), 2012.

[73] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3), 2005.

[74] J. von Neumann. First draft of a report on EDVAC. Technical report, Univ. of Pennsylvania, 1945.

[75] G. F. Welch. A survey of power management techniques in mobile computing operating systems. *SIGOPS Operating Systems Review*, 29(4), 1995.

[76] S. C. Woo, M. Oharat, E. Torriet, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*, 1995.

[77] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Annual International Symposium on Microarchitecture (MICRO)*, 2005.