NRG-Loops: Adjusting Power from within Applications

Melanie Kambadur Martha A. Kim

Columbia University, New York, NY, USA

[melanie,martha]@cs.columbia.edu

Abstract

NRG-Loops are source-level abstractions that allow an application to dynamically manage its power and energy through adjustments to functionality, performance, and accuracy. The adjustments, which come in the form of truncated, adapted, or perforated loops, are conditionally enabled as runtime power and energy constraints dictate. NRG-Loops are portable across different hardware platforms and operating systems and are complementary to existing system-level efficiency techniques, such as DVFS and idle states. Using a prototype C library supported by commodity hardware energy meters (and with no modifications to the compiler or operating system), this paper demonstrates four NRG-Loop applications that in 2-6 lines of source code changes can save up to 55% power and 90% energy, resulting in up to 12X better energy efficiency than system-level techniques.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

Keywords Energy Efficiency, Power Management

1. Introduction

As computers overconsume power, threatening significant financial and environmental consequences [6], energy and power awareness has slowly crept up the stack from the traditional hardware and OS-levels to the application. In 2011, 70% of returned Motorola devices were due to battery life complaints attributable to applications [10], and new power monitors — such as the OS X Battery Status Menu — allow end-users to see which applications are to blame when power consumption is high.

While power efficiency is important, it does not yet trump runtime efficiency. Unfortunately, runtime performance and power efficiency are often at odds, and balancing the two needs simultaneously is a challenge. Until recently, a choice had to be made at hardware design time between optimizing for higher performance or for lower power. Now, instead of preselecting the tradeoffs, computer architects build in the option to dynamically tune hardware resources at runtime. Hardware may now switch between performance-aggressive and power-saving states, or to various states in the middle of those two goals. The different balances of power and performance can be adjusted through a wide menu of power management controls that include changing processor or memory voltage and frequency (DVFS), temporarily overclocking or putting processors into idle states, and choosing from one of multiple asymmetric multicores.

These *system-level knobs* are controlled by operating systems, runtime software, and compilers. A key downside of system-level knobs is that they must be tuned conservatively to avoid disturbing the performance and accuracy of overlying programs. In typical uses, even mildly power-saving states are entered only when the hardware is completely unused by overlying applications resulting in limited power and energy savings for active systems. Another issue with system-level power techniques is that they must be applied to hardware constructs — for example, voltage must be scaled for an entire socket or at least an entire processor core — which is not suitable when multiple applications share a hardware context through multiprogramming or simultaneous multithreading.

To achieve more aggressive energy savings, programs must introduce their own *application-level knobs*. Tough decisions about when it is appropriate to trade performance, accuracy, or functionality for power and energy savings cannot be made by compilers or operating systems. They need to be internal to applications, and made on a case-by-case basis by developers, as different applications may want to make syntactically and semantically varied changes. For example, one application might choose to save power by adjusting its caching strategies, while another might reduce thread counts, and another may choose to scale down data structure sizes. To make power and energy efficiency trades, programmers need two types of support not presently available. First, they need runtime power and energy usage statistics that are accessible to the program's source code, in order

CGO'16, March 12–18, 2016, Barcelona, Spain
ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854045

to evaluate when adaptations are needed, or even whether they are needed at all for a particular program execution. Second, they need language support that helps them incorporate adaptations into source code simply but flexibly, without making assumptions about the underlying platform so that applications remain portable.

This paper provides for both of these needs through a set of language extensions called *NRG-Loops* (pronounced "Energy Loops"). NRG-Loops let applications set run-time evaluated *NRG-Conditions* that dictate whether and when to make changes. NRG-Conditions are used within annotated `for` loops, and ensure that applications make adjustments only in the case that runtime power or energy use meets a specified budget. This budget can be expressed in absolute Watts or Joules, or alternatively set relative to other parts of the program (e.g., function `foo` is allocated 50% of the energy of function `bar`), or relative to the system (e.g. 80% of the maximum system power). At runtime, the budgets are compared to the accumulated energy or average power across loop iterations using measurements abstracted from system hardware counters. When the budget is met, the program dynamically *truncates*, *adapts*, or *perforates* the loop to begin reducing power or energy use. NRG-Loop adjustments can be made in arbitrary application-specific ways, the resulting code is portable to multiple systems, the savings are complementary to system-level energy management techniques, and no compiler or operating system modifications are required.

The primary contributions of this paper are:

- A specification of NRG-Loops, a platform-independent C or C++ language extension that lets applications trade performance, accuracy, or functionality only as dynamic power and energy use necessitates (Section 2).
- A prototype implementation of NRG-Loops, called *NRG-RAPL*, that utilizes hardware power counters to implement the NRG-Loop syntax and semantics for a Linux/Intel platform (Section 3).
- Four case studies that demonstrate 10-90% energy savings and up to 55% power savings by changing just 2-6 lines of source code per program (Section 4).

2. NRG-Loops

NRG-Loops provide a simple, flexible interface for applications to modify their own performance, functionality, and accuracy for power and energy savings. The syntax of NRG-Loops is purposefully brief to avoid a steep learning curve for users. This first version of NRG-Loops extends C or C++ programs, but a similar extension could be developed for other languages. This section describes the syntax and semantics of NRG-Loops, which consists of several abstractions: *NRG-Conditions*, four types of `for`-loop directives, and helper data structures and functions. To increase portability, NRG-Loops abstract away the underlying measurements or models that collect power and energy usage statistics. Later, Section 3 discusses one implementation option for populating this information into the appropriate syntactic structures.

NRG-Conditions

```
NRG_TOT_E <= <float> // in Joules
```

```
NRG_AVG_P <= <float> // in Watts
```

NRG-Loops

```
NRG_TRUNC_for (<loop bounds> &&
               <NRG condition>) {
  // body
}
```

```
NRG_ADAPT_for (<loop bounds> &&
             <NRG condition>) {
  // original body
} NRG_ALTERNATE {
  // alternate body
}
```

```
NRG_PROB_PERF_for (<loop bounds> &&
                   <NRG condition>;
                   PROB_SKIP=<float>) {
  // body
}
```

```
NRG_AUTO_PERF_for (<restricted bound> &&
                   NRG_TOT_E <= <float>) {
  // body
}
```

NRG-Helpers

```
SYS_MAX_POWER
```

```
SYS_MIN_POWER
```

```
struct NRG_USAGE_INFO {
    float energy;
    float average_power;
    float wall_time;
}
```

```
NRG_AUDIT {
  // arbitrary code
} NRG_USAGE(NRG_USAGE_INFO* foo);
```

Figure 1. The NRG-Loops Syntax.

2.1 NRG-Conditions

NRG-Conditions enable programs to monitor accumulated energy or average power across loop iterations, and then to react intelligently to these measurements at runtime. There are two types of NRG-Conditions, as illustrated in Figure 1. The first type of condition, `NRG_TOT_E`, checks if the total accumulated energy across loop iterations is less than or equal to the given value of Joules. The right-hand side can be any expression that evaluates to a floating point value. The second type of condition, `NRG_AVG_P`, checks if the average power across loop iterations is less than or equal to the specified value in Watts, again expressed as a floating point expression. An example NRG-Condition that limits power to 50 Watts is: `NRG_AVG_P <= 50.0`.

Instead of setting NRG-Condition values in terms of absolute Watts or Joules, the power or energy limits for one piece of code can alternatively be set relative to the amount consumed by a different part of source code at runtime. For example, a user could write an NRG-Condition that ensures a loop body in function `bar()` consumes at most the energy that function foo() consumed: NRG_TOT_E <= foo_energy.

The value of foo_energy could be predetermined at development time, but more likely, users will want to dynamically import such a value, because energy varies across platforms, inputs, and even different executions. Dynamic energy measurement is easy with the help of NRG_AUDITs, which enclose an arbitrary region of code in a pair of curly braces, as shown in Figure 1. The enclosed code may perform any computation including spawning threads and calling functions — even calling more NRG_AUDITS. The audits record the energy (in Joules), the average power (in Watts) and the wall time (in seconds) of the enclosed region (and any child threads or functions) and deposit the information into a named NRG_USAGE_INFO structure. For example, foo_energy can be obtained as follows:

```
NRG_AUDIT {
  foo();
} NRG_USAGE (NRG_USAGE_INFO* foo_usage);
float foo_energy = foo_usage->energy;
```

2.2 Truncate Loops

NRG-Conditions serve as a secondary for loop bound (concatenated to the original loop bound) for different types of NRG-Loops. The first type of NRG-Loop is called an NRG_TRUNCATE_for. Continuing our foo() example, an NRG_TRUNCATE_for can be expressed as follows:

```
NRG_TRUNCATE_for (int i=0; i<N; ++i &&
                  NRG_TOT_E <= foo_energy) {
  // do work until foo_energy exceeded
}
```

This directive tells the loop body to execute while both the original loop bound (int i=0; i<N; ++i) and NRG-Condition (NRG_TOT_E <= foo_energy) hold, and to stop execution otherwise. Like a regular loop bound, the NRG-Condition is checked only at the beginning of a loop iteration, and thus will not stop a loop in the middle of an iteration, even if the power limit or energy budget has already been exceeded. The user rather than the NRG_TRUNCATE_for is responsible for any clean-up (e.g. releasing a lock, freeing memory, closing files) that may be required as a result of exiting the loop early.

2.3 Adapt Loops

The next type of loop directive is an NRG_ADAPT_for. Like the truncate directive, it concatenates an NRG-Condition to the original loop bound. It also has the user add a second, alternate loop body that directly follows the first, is wrapped in brackets, and is preceded by the NRG_ALTERNATE keyword as shown in Figure 1 and the snippet below:

```
NRG_ADAPT_for (int i=0; i<N; ++i &&
                  NRG_TOT_E <= foo_energy) {
  // do work until foo_energy exceeded
} NRG_ALTERNATE {
  // do more energy efficient work
}
```

Execution of the original loop body continues while both the original loop bound and the NRG-Condition hold. If the NRG-Condition breaks before the original bound, then execution transfers to the alternate loop body. After the transfer, the alternate body continues to execute until the original bound is met. Note that the original bound state (e.g., loop index value or i in then running example) is *not* reset upon transfer to the alternate body. As with the truncate loop, NRG-Conditions are checked only at the beginning of loop iterations and control is never transferred in the middle of an iteration. Again, any required clean up relating to loop body transfer must be handled by the user.

In the case of an NRG_AVG_P condition, loop execution may transfer back to the original loop body if the average power goes back below the specified limit after the alternate body is executed for a time. This will not happen with an NRG_TOT_E condition, because total energy increases monotonically.

2.4 Perforate Loops

Finally, there are two types of NRG-Loops that allow applications to perforate, or skip select loop iterations. The first type, NRG_PROB_PERF_for, executes normally until the NRG-Condition bound is exceeded, then probabilistically skips iterations with a probability of the specified PROB_SKIP, which should be a floating point number between 0 and 1. For example, if the user specifies PROB_SKIP = 0.1 as in the following example, once the NRG-Condition has been exceeded, 1 out of 10 future loop iterations will be skipped.

```
NRG_PROB_PERF_FOR (int i=0; i<N; ++i &&
                   NRG_TOT_E <= foo_energy;
                   PROB_SKIP = 0.1) {
  // once NRG-Condition met, do work 9/10 times
}
```

The second type of perforation is NRG_AUTO_PERF_for. This type of NRG-Loop automatically decides how many loop iterations to skip in order to meet a user-specified energy budget. Unlike the other types of NRG-Loops, it does not support NRG_AVG_P and it restricts the original loop bound to be of the form (int i=0; i<=N; ++i). For example:

```
NRG_AUTO_PERF_FOR (int i=0; i<N; ++i &&
                   NRG_TOT_E <= foo_energy) {
  // do work, skipping an estimated number of
  // iterations to exactly match foo_energy
}
```

These two restrictions allow the loop increment to be modified (e.g., change ++i to i=i+2 or i=i+3, etc.) to keep the loop on target to consume no more energy than specified in the NRG_TOT_E condition.

2.5 NRG Helpers

NRG-Loops also provides two helpers to assist users in choosing platform-independent NRG-Condition values. The first helper, SYS_MAX_POWER, is a global floating point value that holds the maximum power (in Watts) that the platform can achieve with all hardware threads active. Similarly, SYS_MIN_POWER is a global floating point value that holds the minimum power (in Watts) of the system when running essential services only (i.e., an idle operating system). These two constants can be used within NRG-Conditions to further abstract NRG-Loop code from a particular platform, for example, NRG_AVG_P <= 0.8*SYS_MAX_POWER.

3. NRG-RAPL

The first implementation of the NRG-Loops interface is a C library called *NRG-RAPL*. It is named for its use of Intel's *Running Average Power Limit (RAPL)* [14] counters to profile energy. NRG-RAPL is portable and lightweight, utilizing commodity hardware and requiring no operating system extensions or middleware. This section describes the library implementation, including how NRG-Loops syntax is translated at the preprocessor level into pure C (Section 3.1), how energy is profiled (Section 3.2), and how we attribute the hardware-level measurements to software constructs (Section 3.3). The tool usage (Section 3.4) and limitations (Section 3.5) are also discussed.

3.1 Translating NRG-Loops

Since the NRG-Loops syntax is highly abstracted to minimize programmer effort, NRG-RAPL's first job is to translate the code into a pure C intermediate representation. Translation involves (1) accumulating energy or recording average power across loop iterations as required, (2) checking this usage against the specified limits, and (3) adjusting the source code as necessary. To collect power and energy data, the intermediate representation uses the previously described NRG_AUDIT helper and its NRG_USAGE_INFO structure.

Each kind of loop directive and type of NRG-Condition has its own intermediate representation, so there are eight types of translations in total. Space prevents us from sharing all eight, but as an example, here is the intermediate representation of an Adapt NRG-Loop when an energy condition is used:

```
float NRG_BUDGET = 0.0;
for (int i = 0; i<N; ++i) {
  if (NRG_BUDGET <= <float>) {
    NRG_AUDIT {
      // original loop body
    } NRG_USAGE(NRG_USAGE_INFO *use);
    NRG_BUDGET += use->energy;
  } else {
    // alternate loop body
  }
}
```

The translated code starts the loop with its original bounds, inserting a check at the top of each iteration to see if energy use has exceeded or met the user-specified budget. If it has, control transfers to the alternate body through the use of an else statement. If the budget has not been met, the original loop body runs — within an NRG_AUDIT. The recorded NRG_USAGE_INFO is accessed to update the budget based on the energy consumed by the loop body.

3.2 Profiling Energy and Power

Any power profiler or model could be used to collect energy and power within the audit calls. There is some debate about the best way to monitor power and energy today, because techniques vary widely in terms of implementation complexity, precision, accuracy, portability/availability, and scope of coverage (i.e., is only processor power being measured, or the whole system's power draw including any peripherals such as monitors?). Unfortunately, no existing power measurement technique fares well in all of these categories. For this implementation, we chose to use a combination of hardware power meters and operating system usage statistics that performs at least reasonably in each of the categories and has been shown to be among the most accurate techniques for measuring power [29], and has been used in several previous works (e.g., [32] and [35]).

Hardware meters supplement power measurements with event-based linear models that are periodically re-calibrated. Meters are conveniently found in widely available server SoCs such as the Intel SandyBridge and the IBM POWER7, and have recently been introduced to mobile devices. The meters can cover a large portion of computation, accounting for processor, interconnect, cache, and DRAM power and energy. However, with update frequencies on the order of 1ms, they are not terribly precise and have been shown to have occasional modeling errors [22]. Other drawbacks are that the meters are small and overflow frequently, and that they currently exist only at the granularity of a whole socket, which can contain multiple CPUs running many concurrent processes, making it a challenge to attribute power measurements to individual processes and threads.

3.3 Energy Accounting in NRG-RAPL

NRG-RAPL's energy accounting fixes the overflow and hardware meter granularity problems, and reduces the overheads involved in dynamic profiling by combining meter reads across multiple concurrently running audit functions. When the application has one or more audits open, NRG_RAPL spawns a single monitoring thread (regardless of the number of active audits) to periodically sample:

- E_{sys}: a system-wide energy reading obtained from RAPL counters for all sockets
- U_{sys}: system-wide CPU time from /proc/stat
- U_{tid}: CPU time for each application thread tid from /proc/<pid>/tasks/<tid>/stat

The frequency of these readings is configurable, but RAPL samples must be taken frequently enough to provide good precision and to detect overflow — the RAPL counters overflowed roughly every 10-20 seconds on our experimental machine — yet not so often that profiling results in excessive overhead. Sampling at 100 Hz strikes a good balance between these constraints on our machine, yielding reasonable precision with negligible time or power overhead above the unmonitored application.

As every i^{th} sample is recorded, NRG-RAPL decomposes it into individual measurements for every active thread tid according to the following equation:

$$E_{tid,i} = \frac{U_{tid,i} - U_{tid,i-1}}{U_{sys,i} - U_{sys,i-1}} \times (E_{sys,i} - E_{sys,i-1})$$

This divides the measured energy values amongst running threads according to CPU utilization. Thus, should another thread or co-running application run up E_{sys}, that usage will not be charged to tid.

In addition to the profiling samples, NRG-RAPL maintains an application thread tree by interposing on calls to pthreads which create or destroy threads. It also maintains information about the nested structure of the audits created per thread by reading the opening and closing brackets surrounding audits in the intermediate translations of NRG-Loops. Combining the active thread and audit information with $E_{tid,i}$ measurements, NRG-RAPL fills in usage records of any open audits of the thread tid and its ancestors.

3.4 Usage Logistics

After a user adds NRG-Loop directives to their application, he or she must link against NRG-RAPL (i.e., `-lnrgrapl`), and ensure that the NRG-RAPL shared object file is loaded first (i.e., via `LD_PRELOAD` or `LD_LIBRARY_PATH`). The only other requirement of the current implementation is that, once compiled, the application be executed by a sudoer. This is because Linux does not currently expose even read-only access of the RAPL registers to non-sudoers. This work is just one example of why it would be beneficial for Linux to do so in the future.

3.5 Limitations

Measurement precision. RAPL registers are updated every 1ms and the CPU usage every 10ms (at 100 jiffies/sec), so NRG-RAPL cannot audit sub-10ms windows of execution. However, since the application must adjust loops that represent large chunks of execution to make a difference in energy consumption, this may not be a practical limitation. In the case studies presented in the next section we had no problems with precision.

System coverage. The specifics of the power model remain, by design, orthogonal to the NRG-Loop interface, with the expectation that as power models improve and energy meters become more pervasive, the availability and quality of power information provided through the NRG-Loop interface will only improve. For example, in smartphones, many non-processor resources such as the network and backlight account for a significant portion of smartphone power draw. To the extent that their energy usage is exposed, it can be incorporated into improved implementations of the NRG-Loop interface.

Library Preemption. With a user-level library like NRG-RAPL, it is possible that the monitor thread could be preempted or delayed, thus creating irregular profiling samples. However, Linux never stalled the monitor in any of our experiments, including stress tests with more than 100 busy application threads. On extremely busy systems, it may be necessary for monitor threads to be granted a higher priority so that they are not preempted.

4. Case Studies

This section demonstrates the potential of NRG-Loops and the immediate utility of NRG-RAPL with four case studies that: maintain an energy budget by trading-off video quality (Section 4.1), respect a power cap by reducing parallelism (Section 4.2), save power by approximating a mathematical algorithm (Section 4.3), and keep a third-party advertisement's power use in check (Section 4.4). The NRG-RAPL instrumentation adds minimal energy, power, and runtime overhead to these applications (Section 4.5).

The experimental platform is a dual socket Dell PowerEdge R420 server with Intel Sandybridge E5-2430 processors, each with 6 cores, 12 hardware threads, and 24GB of DRAM (for a machine total of 12 cores, 24 threads, and 48GB DRAM). The machine runs Linux kernel version 3.9.11 and Ubuntu 12.04.2. Intel sleep states [25], Turbo Boost [4], and the ondemand frequency governor [26] are turned *on* for all the experiments to demonstrate that NRG-Loops complement and extend the savings of existing system-level energy management techniques.

4.1 Perforate: Bodytrack

Sometimes the best strategy to meet high computational demands under strict energy budgets is to reduce the accuracy of application services provided. To demonstrate real-world, application-level accuracy tradeoffs, we augmented the `bodytrack` application from the Parsec benchmark suite [2]. `Bodytrack` tracks the poses of a person recorded on multiple video cameras; the majority of this work occurs in a for loop within the main function, which processes frames one at a time. We perforated this loop using both types of NRG perforation loops. For the first type of perforation, we varied the probabilistic percentage of loops a user might choose to skip from 0 to 75% by modifying the `MY_PROB` variable in the code below. Note that the NRG-Condition is somewhat of a no-op in this example, telling the code to start perforating when energy use is greater than or equal to zero; we did this to make the numbers comparable to our next experiment.

```
NRG_PROB_PERF_for (int i=0; i < frames; ++i &&
  NRG_TOT_E <=0; PROB_SKIP = MY_PROB) {
    // DO FRAME PROCESSING
}
```

Varying the probability of perforation (to 0.05, 0.1, 0.2, 0.5, and 0.75) produces the whole program energy savings shown to the left of Figure 2 when `bodytrack` is run with an input size of native and the `-O3` compiler flag is enabled.

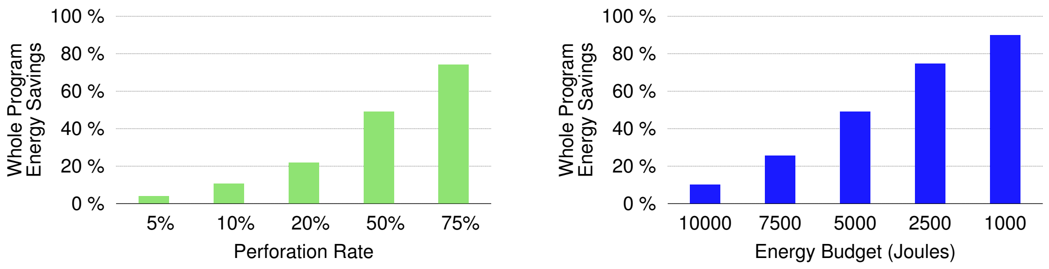

Figure 2. An NRG Perforate Loop augments `bodytrack` to (left) drop different specified percentages of frames to save energy, or (right) maximize quality without exceeding various allocated energy budgets.

Unsurprisingly, as more frames are skipped more energy is saved, ranging from 3.9% savings when 5% of frames are dropped to 74% savings when 75% of frames are unexecuted. Of course, in most applications skipping 75% of the work is not reasonable. Developers will have to decide how many frames it makes sense to trade for energy, but at least with NRG-Loops they can now reason about the energy values of these tradeoffs and easily affect the changes they want to make at runtime, incorporating dynamic conditions such as changing inputs into their decision.

The second way to initiate perforation changes in a program using NRG-Loops is to set a total energy budget. This may be preferable in situations where the developer knows exactly how much energy they need to save, and is flexible about the number of frames skipped. Adding this kind of NRG-Loop to a program is as easy as choosing the `BUDGET` to spend, connecting the library, and modifying a single line of source code:

```
NRG_AUTO_PERF_for (int i=0; i < frames; ++i &&
  NRG_TOT_E <= BUDGET) {
    // DO FRAME PROCESSING
}
```

To find appropriate possible `BUDGET`s, we first used an `NRG_AUDIT` helper to determine the overall energy consumed by the program, which is approximately 9600 Joules on our machine. The right side of Figure 2 shows the percentage of energy saved when the `BUDGET` is set to 10,000, 7500, 5000, 2500, and 1000 Joules, and the framerate is adjusted automatically by the NRG Perforate Loop to meet these budgets. Even though 10,000 is above the un-budgeted, original program's energy, there is still a little savings because the NRG-Loop drops a few early frames before it is sure that the target budget will be met. For lower budgets, even more frames are dropped throughout the loop's iterations, and the energy savings are accordingly higher — up to 89.8% when the budget is set to 1000 Joules.

4.2 Adapt: Parallel Substring Search

For safety reasons and to prevent overheating, hardware enforces a *hard power cap*. Called a TDP, or thermal design point, this upper bound for power varies by architecture, and in practice may never be reached thanks to efficient cooling strategies. For example, on our experimental machine, which has a 190W TDP, we never observed a peak power of more than 120W even with all 12 cores fully utilized. Despite this, there are numerous reasons one may wish to cap an application's power below the TDP, for example to make datacenter energy expenses more predictable and affordable, to allow more headroom under the TDP for applications sharing a machine, or to throttle usage when the power supply is intermittent or variable as with RFID harvesting [7], or solar [33] power sources. We call these sub-TDP power caps *soft power caps*.

Several existing tools tune hardware and operating system resources to enforce user-specified soft caps, using techniques such as DVFS, thread mapping, and asymmetric hardware [5, 15, 31]. The soft caps provided by these tools are hardware-centric, and thus applied to specific hardware components such as a single core, a socket, or a whole machine. In contrast, NRG-Loops enable soft caps on software entities, specifically those encased by `for` loops.

Software-centric power caps can be simply implemented using the `NRG_ADAPT_for` directive. To compare system-based soft caps to NRG Adapt Loop-based soft caps, we implemented a C++ benchmark that searches many long base strings for a particular substring match. Substring searching is used in many important real world applications, such as genomic analysis and satellite image processing. For faster throughput, base strings can be searched in parallel by multiple software threads. And, since parallelism is highly correlated with power [16], we opted to use dynamic adjustments to the degree of parallelism to regulate the application's power and enforce the soft cap.

The code below shows how NRG-Loop annotations added to the benchmark can enforce a user-specified cap at `SOFT_CAP` Watts.

```
NRG_ADAPT_for (int i=0; i<STRINGS_TO_CHECK; ++i &&
               NRG_AVG_P <= SOFT_CAP) {
  if (num_threads < MAX) num_threads += 2;
  // num_threads search concurrently for substring
} NRG_ALTERNATE {
  num_threads -= 2;
  if (num_threads < MIN) num_threads = MIN;
  // num_threads search concurrently for substring
}
```

As the benchmark searches base strings with `num_threads` threads, an `NRG_ADAPT_for` checks that average power stays below the `SOFT_CAP`. If it does not, control shifts to the alternate loop body, where the thread count is decreased by 2, and a check for over decrementing is performed to ensure that `num_threads` is at least a `MIN` number of threads (in our experiments, 2). Afterwards, the smaller `num_threads` search for the substring just as in the original loop body. Finally, in case the average loop power happens to go back above the `SOFT_CAP`, execution will automatically return to the original loop body, so we also added a line of code to the original body that increments the threads back up by 2 to increase search speed.

In the code snippet, `SOFT_CAP` is expressed as an absolute value, but it could also be expressed relative to the system maximum using the helper described in Section 2.5, for example: `#define SOFT_CAP 0.5*SYS_MAX_POWER`.

We compare the NRG-Loop solution against Intel's RAPL and DVFS-based power capping tool, Intel Power Governor [13], which is an example of system-only soft caps. The tool lets users limit power across three domains per socket: *power plane 0 (PP0)* which includes cores and private caches, the *package (PKG)* which includes all PP0 elements as well as alternate processing units such as GPUs and shared caches, and *DRAM*. The user selects one or more of these three domains to cap, then gives the Power Governor interface a specific power value in Watts to limit each domain to, as well as a time window in milliseconds over which the running average power must not exceed the cap. As an example, if the time window is 100ms and the cap is 30W, RAPL promises that for every 100ms, the average power of the target RAPL domain will not exceed 30W. Because RAPL caps are enforced in multiple domains, getting our benchmark to respect an overall soft cap required tuning all the individual settings. For example, to get the program to run within a soft cap of 55 Watts, the RAPL PKG cap had to be set to 45 Watts, and the other two domains had to remain uncapped. To get the program to run within a soft cap of 40 Watts, we had to set the PKG cap to 15 Watts and the DRAM to 15 Watts and disable Turbo Boost. Such device and system-specific tuning is probably more than most software developers will wish to take on, particularly as the settings must be re-tuned for each soft cap, application, device, and, potentially, input.

Figure 3 shows the energy impact on the substring search benchmark when NRG-Loops and the Power Governor enforce a range of soft caps between 30 and 65 Watts. In each experiment, the maximum thread count is set to 12, the number of base strings to 12 thousand, and the length of each base string to 5 million. Each bar reports the program energy at a particular cap relative to the energy of the uncapped program, which maximizes performance by running continuously with 12 threads. For the NRG-Loop soft caps, the reduced power was always almost exactly offset by the increase in runtime due to decreased parallel processing. Thus, the total energy was roughly equivalent to the peak performance energy, regardless of what cap was set.

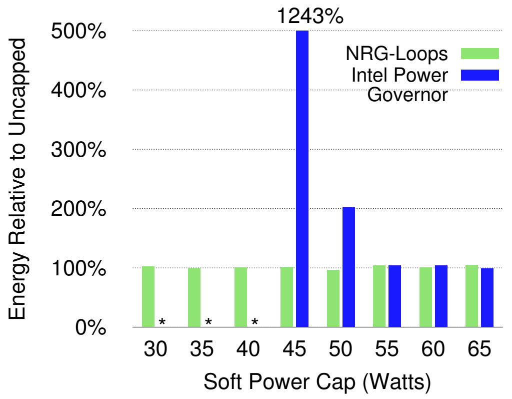

Figure 3. NRG Adapt Loops can meet a preset power budget by adjusting application-internal thread count, analogously to the Intel Power Governor tuning DVFS. For the string matching application shown, NRG-Loops can set a broader range of caps (Power Governor caps could not be used below 45 Watts), and required up to 12X less energy to enforce them.

The same was not true for the Power Governor caps. First, no Power Governor cap setting could produce a soft cap below 45 Watts. Even at 45 Watts, the Power Governor struggled to maintain the soft cap, significantly increasing runtime so that the energy consumption was more than 12 times both the uncapped program and the NRG Loop-based cap. For the 50 Watt Power Governor soft cap, the energy consumed was still 2X the uncapped energy. Only at the 60 Watt cap, as the program neared its uncapped power of 63 Watts, were Power Governor caps finally able to keep energy within 4% of uncapped.

At least for this application, NRG-Loops worked better than system capping in two ways. First, once we added the 7 lines of code above to the program, setting a new cap was as simple as passing a new value for `SOFT_CAP`); far preferable to the complicated tuning required to set the Power Governor caps. Second, the NRG-Loop based caps offered a broader range of viable power caps than Power Governor, often at a significantly lower energy consumption.

4.3 Truncate: Streamcluster

Algorithms sometimes spend valuable energy converging to a perfect solution when an approximate solution is good enough. The `streamcluster` application from the Parsec benchmark suite [2] is one example of this. Streamcluster is a data mining/pattern recognition application that solves the online clustering problem, assigning a stream of input points to their nearest *center* [24]. Misailovic et al. identified a loop within the `pFL` function that can be approximated: if given fewer iterations, the number of centers that the program considers clustering the data around will be decreased, possibly without detriment to a final solution [23].

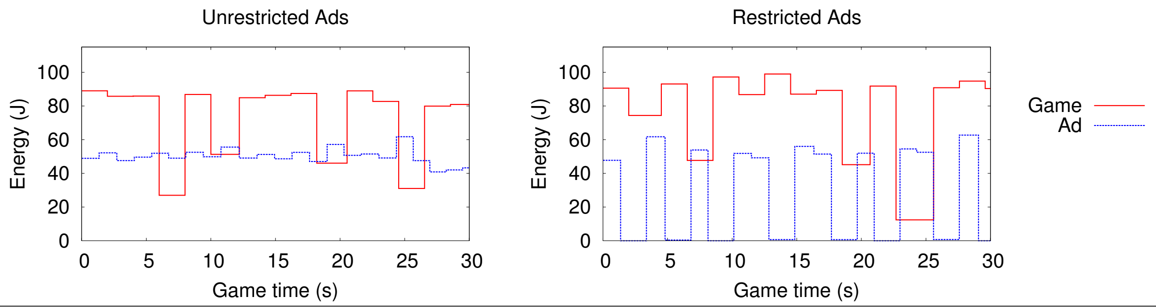

Figure 5. A minesweeper game uses NRG-Adapt Loops to prioritize game power over third-party advertisements. Run unchecked, the ads sometimes consumes more power than the game, but NRG-Loops can force the ads to occasionally pause, decreasing net game+ad energy.

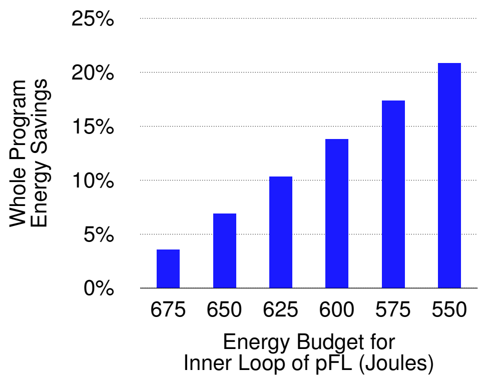

Figure 4. NRG Truncate Loops estimate a mathematical clustering algorithm within `streamcluster` to save various amounts of whole program energy depending on the degree of approximation.

In the Misailovic et al. work, the authors tried to guarantee minimal disturbance to an ideal solution, so they conservatively perforated the loop. In a scenario where overall energy savings is more important that guaranteeing a near-perfect outcome of the algorithm, a truncating NRG-Loop may be appropriate. Truncating the inner-loop of `pFL` with NRG-Loops is as simple as including the NRG-RAPL header file and modifying the original `for` loop to include an extra NRG-Condition:

```
float pFL (<args>) {
  NRG_TRUNCATE_for (i=0; i<iter; ++i &&
    NRG_TOT_E <= BUDGET) {
    // COMPUTE CLUSTER VALUE
  }
}
```

To determine the potential whole program energy savings of truncating the loop, we experimented with different values assigned to the `BUDGET` variable in the NRG-Condition. To find a baseline for our experiments, we first used an `NRG_AUDIT` and discovered that, when run with native inputs, the inner loop consumes anywhere from 600 to 733 Joules to complete all of its iterations. With a `BUDGET` value of 675 Joules (which is in the range of un-truncated consumption and therefore most likely affects program accuracy minimally), NRG-Loops provides a whole program energy savings of 3.5%, as shown in Figure 4. Reducing the energy of the inner loop further — down to 550 Joules — results in a whole program energy savings of over 20%.

4.4 Adapt: Minesweeper and Advertisement

Free applications comprise 91% of the mobile marketplace [11], and 77% of the top free applications in the Google Play store are advertisement supported [18]. Moreover, third-party ads may consume as much as 65-75% of the total mobile application energy [27]. These numbers indicate that developers and mobile providers alike have an incentive to ensure that mobile ads consume only their fair share of energy.

While operating systems could be tasked with moderating ad energy use, that would take valuable control away from developers. Instead, NRG-Loops make it simple for developers to moderate ads on their own terms using familiar programming techniques even if the advertisements are written by third-parties and have unpredictable demands. To demonstrate this, we introduced NRG-Loops into a text-based minesweeper game [21] that calls a simulated advertisement. The advertisement in this experiment is a separate pthread that performs computationally and I/O intensive busywork. The minesweeper game has potentially long rounds of play with varying durations depending on live user interactions. To keep the advertisement's power in check, we converted the code that calls the advertisement into an Adapt NRG-Loop that pauses the ad for `PAUSE_TIME` microseconds if it consumes more than a given `POWER_LIMIT`:

```
NRG_ADAPT_for (int i=0; i<MAX_ADS; ++i &&
               NRG_AVG_P <= POWER_LIMIT) {
  // run ad normally
} NRG_ALTERNATE {
  usleep(PAUSE_TIME);
}
```

Figure 5 shows resource measurements of two versions of the game being played by a real user. The first version (at left) does not restrict ad energy, while the second version (at right) restricts advertisements with the `PAUSE_TIME` set 2 seconds and the `POWER_LIMIT` to 20 Watts. Both graphs show the energy in Joules consumed by the minesweeper game in two second intervals over 30 total seconds of play, and the energy of the advertisement recorded at each iteration of the above `for` loop. Fluctuations in the game energy (solid red series) are a result of dynamic user interactions. In the unrestricted ads version, the advertisement energy (dashed blue series) regularly consumes more than half of the energy of the game, and sometimes even exceeds the game's energy. In the restricted version of the game, the ads are periodically paused to respect the power limit. Together the unrestricted game and ads consume an average of 72 Watts, while the restricted game and ads consume only an average 54 Watts together.

4.5 Overheads

To find the overheads of NRG-RAPL, we compared the applications' runtime, power, and energy with no NRG-Loops annotations versus with annotations added but dynamic modifications disabled (i.e. the frame skip probabilities were set to zero, and the energy budgets and power caps were set to unattainable levels). We then used a standalone RAPL monitor thread to measure 10 trials of the programs running on the opposite chip socket so that the power and energy consumption of the monitor itself would not be counted. Across trials and benchmarks, the maximum energy increase was 0.6%, the maximum power increase was 0.1%, and the maximum runtime increase was 1.4% (on average, -0.4%), indicating that NRG-RAPL has a very limited affect on program performance. These overheads do not include the minesweeper game, because the live user interaction makes the energy usage and runtime highly variable.

5. Related Work

Energy management is an established and crowded field. This section contextualizes NRG-Loops within this large body of work, relating it to examples of different categories of efficiency techniques: *system-only management* techniques, those that use *application hints* or both *application hints and exposed knobs*.

System-Only Management. Most energy managers operate entirely at the system level, with the system both offering the energy conserving *knobs* and initiating the action to tune them. Projects that fall into this category including Linux's `cpufreq` and `cpuidle` governors [3, 25], Pack & Cap [5], computational sprinting [28], and a tool for optimizing dynamic backlight scaling [19]. Modern systems offer many knobs, including DVFS, overclocking, idle states, adjustable DRAM refresh rates, asymmetric multicores, and configurable floating point widths. These knobs let the system avoid using and paying for any more resources than necessary to maintain performance and accuracy, but must be conservatively tuned to avoid performance or accuracy hits to overlying applications.

Application Hints. Another category of energy management tools adds application hints to help manage system resources. These tools use annotations or new languages to denote regions of code for which it is safe to optimize power while potentially reducing performance or accuracy. To make the adjustments or approximations, they rely on the same set of system-level knobs as tools in the previous category, though those knobs may now be tuned more aggressively. For example, EnerJ [30] is a language where the type system indicates which program values can tolerate imprecision for subsequent approximation by the runtime system. Some of the other techniques in this category include architecture support for disciplined approximate programming [8], Flikker [20], Eon [33], and the Latency, Accuracy, Battery abstraction [17].

Application Hints & Exposed Knobs. A third category of work lets the application expose internal knobs (most commonly, loop perforation), but ultimately relies on the system to decide when and by how much to tune those knobs. Examples include PowerScope used with the Odyssey OS [9], GRACE-2 [34], PowerDial's Dynamic Knobs [12], the Green framework [1].

Application-Only Management. NRG-Loops comprise a new category of management solutions, that allow applications to tune their own knobs from within themselves. There are several important benefits to forgoing system involvement. First, application-only management provides transparency and control to the programmer, which beats hoping for a system's "best effort" of efficiency. Additionally, application-only management does not require modifying the operating system, meaning that updating new application knobs is simpler. Finally, application-only management is portable — the source code is not tied to specialized systems, and thus program annotations do not need to be revised to execute on new platforms.

6. Conclusion

In previous work, the operating system or specialized hardware takes sole responsibility for power and energy efficiency. In contrast, NRG-Loops enable *applications* to manage their own power and energy, resulting in more control for programmers and potentially more aggressive tradeoffs and therefore greater efficiency. This work showed that application-only management is not only complementary to system-level management, but that is also portable, simple, and effective — saving up to 55% of whole system power and up to 90% of system energy with just a few lines of source code modifications. All of the benefits of NRG-Loops are immediately available through a software-only C library (NRG-RAPL) which runs on commodity hardware and does not require changes to the operating system or a new runtime system.

7. Acknowledgements

This work is partially supported by the DARPA PERFECT program (HR0011-13-C-0003) and the National Science Foundation (CCF-1253772).

References

[1] W. Baek and T. M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*.

[2] C. Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[3] D. Brodowski and N. Golde. CPU frequency and voltage scaling code in the Linux kernel: CPUFreq Governors.

[4] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. 2009. Evaluation of the Intel Core i7 Turbo Boost feature. In *IISWC*.

[5] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. 2011. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *MICRO*.

[6] G. Cook, T. Dowdall, D. Pomerantz, and Y. Wang. 2014. Clicking Clean: How Companies are Creating the Green Internet. greenpeace.org.

[7] D. De Donno, L. Catarinucci, and L. Tarricone. 2013. An UHF RFID Energy-Harvesting System Enhanced by a DC-DC Charge Pump in Silicon-on-Insulator Technology. *IEEE Microwave and Wireless Components Letters* 23, 6 (2013).

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012. Architecture support for disciplined approximate programming. In *ASPLOS*.

[9] J. Flinn and M. Satyanarayanan. 2004. Managing Battery Lifetime with Energy-aware Adaptation. *TOCS* 22, 2 (2004).

[10] N. Gohring. 2011. Motorola CEO: Open Android Store Leads to Quality Issues. Computer World.

[11] M. Hamblen. 2013. Mobile app download tally will soar above 102B this year. Computer World.

[12] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *ASPLOS*.

[13] Intel Corporation. 2012. Intel®Power Governor. `https://software.intel.com/en-us/articles/intel-power-governor`.

[14] Intel Corporation. 2015. Intel 64 and IA-32 Architectures Software Developer's Manual.

[15] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. 2006. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *MICRO*.

[16] M. Kambadur and M. A. Kim. 2014. An Experimental Survey of Energy Management Across the Stack. In *OOPSLA*.

[17] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. 2013. The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing. In *OOPSLA*.

[18] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. 2012. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Workshop on Mobile Computing Systems & Applications*. 2.

[19] C.-H. Lin, P.-C. Hsiu, and C.-K. Hsieh. 2014. Dynamic Backlight Scaling Optimization: A Cloud-Based Energy-Saving Service for Mobile Streaming Applications. *IEEE Transactions on Computers* 63, 2 (2014).

[20] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. 2011. Flikker: saving DRAM refresh-power through critical data partitioning. In *ASPLOS*.

[21] M. Marczykowski and K. Sachanowicz. 2013. The Saper Project (a minesweeper game). Version X.0.14.

[22] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. 2011. Evaluating the Effectiveness of Model-based Power Characterization. In *USENIX*.

[23] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. 2010. Quality of Service Profiling. In *ICSE*.

[24] L. OCallaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. 2002. High-performance clustering of streams and large data sets. In *International Conference on Data Engineering*.

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

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

[27] A. Pathak, Y. C. Hu, and M. Zhang. 2012. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *EUROSYS*.

[28] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. 2012. Computational sprinting.

[29] S. Rivoire, P. Ranganathan, and C. Kozyrakis. 2008. A Comparison of High-level Full-system Power Models. In *Hot-Power*.

[30] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *PLDI*.

[31] H. Sasaki, S. Imamura, and K. Inoue. 2013. Coordinated power-performance optimization in manycores. In *PACT*.

[32] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. 2013. Power containers: an OS facility for fine-grained power and energy management on multicore servers. In *ASPLOS*.

[33] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. 2007. Eon: a language and runtime system for perpetual systems. In *SenSys*.

[34] V. Vardhan, W. Yuan, A. F. Harris, S. V. Adve, R. Kravets, K. Nahrstedt, D. Sachs, and D. Jones. 2009. GRACE-2: Integrating fine-grained application adaptation with global adaptation for saving energy. *International Journal of Embedded Systems* 4, 2 (2009).

[35] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. 2002. ECOSystem: managing energy as a first class operating system resource. In *ASPLOS*.