Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs

Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest

Abstract—Data centers account for a significant fraction of global energy consumption and represent a growing business cost. Most current approaches to reducing energy use in data centers treat it as a hardware, compiler, or scheduling problem. This article focuses instead on the software level, showing how to reduce the energy used by programs when they execute. By combining insights from search-based software engineering, mutational robustness, profile-guided optimization, and approximate computing, the Producing Green Applications Using Genetic Exploration (POWERGAUGE) algorithm finds variants of individual programs that use less energy than the original. We apply hardware, software, and statistical techniques to manage the complexity of accurately assigning physical energy measurements to particular processes. In addition, our approach allows, but does not require, relaxing output quality requirements to achieve greater non-functional improvements. POWERGAUGE optimizations are validated using physical performance measurements. Experimental results on PARSEC benchmarks and two larger programs show average energy reductions of 14% when requiring the preservation of original output quality and 41% when allowing for human-acceptable levels of error.

Index Terms—power optimization; search-based software engineering; genetic algorithms; profile-guided optimization; optimizing noisy functions; accurate energy measurement.

1 INTRODUCTION

THE use of data centers has expanded in recent years to support a growing spectrum of applications. At these scales, *non-functional* [1] properties such as energy consumption can have significant economic and environmental impact. This article addresses the application side of this problem, reducing the energy requirements of programs running on server-class hardware. We discuss the problem of partitioning the energy usage of a server among its hardware and software components, and present hardware, system configuration, and algorithmic techniques for managing it. We then combine insights from search-based software engineering (SBSE), mutational robustness, profile-guided optimization, and approximate computing to modify non-functional properties after compiler optimizations have been applied.

Over the past few years, data center energy usage has grown to over 1% of global energy consumption [2] and is projected to cost American businesses $13 billion per year by 2020 [3]. The mechanical and electrical systems (such as lighting, cooling, air circulation, and uninterruptible power supplies) required to support warehouse-scale computation can quadruple the power required by the computation itself [4]. Because the load on many of these support systems grows with the computational load, computational efficiency is a significant determinant of the economic and environmental costs of data centers. In this setting, even modest reductions in energy consumption or heat generation can, through the scale of deployment, produce significant savings.

Researchers have approached energy reduction from several perspectives, including hardware (e.g., providing for voltage scaling and resource hibernation [5]), scheduling (e.g., predicting the interaction between workloads running on shared systems [6]), compilation (e.g., using instruction scheduling [7] to lower the switching activity between consecutive instructions [8]), and the API (e.g., selecting low-energy API implementations [9]). These perspectives are largely complementary—reducing the scheduling interference of applications running on low-power hardware may reduce energy costs beyond what either approach could achieve alone. In this article, we focus on the software itself, separate from the additional costs of the hardware or operating system.

One of the difficulties in managing energy usage at the software level is lack of visibility into how implementation decisions relate to energy use [9]. Indeed, the success of approaches at so many different levels (hardware architecture, operating systems, compilers, and API selection) is an artifact of the large number of variables that influence energy consumption. We address this difficulty using stochastic search to modify compiled assembly programs and measure the energy required to execute an indicative workload, allowing us to identify beneficial modifications. We observe that common hardware-based techniques for measuring energy consumption have significant limitations; we therefore propose and evaluate techniques to mitigate this issue. Our method, called POWERGAUGE, provides energy reductions that retain required functionality, and we investigate more aggressive reductions for applications that can tolerate slight reductions in output quality [10]. We achieve this via a multi-dimensional search algorithm.

We evaluate POWERGAUGE on the PARSEC [11] bench-

- *Dorn, Lacomis, and Weimer were with the Department of Computer Science at the University of Virginia, Charlottesville, VA, 22904 when this work was done.*
  Email: {dorn,lacomis,weimer}@virginia.edu
- *Forrest is with Arizona State University, Tempe, AZ and the Santa Fe Institute, Santa Fe, NM 87287.*
  Email: stephanie.forrest@asu.edu

Manuscript revised Nov 9, 2017

mark programs to allow comparison to earlier work. These benchmarks were designed to mimic the behavior of data center applications. However, modern real-world data center applications are much larger than these benchmark applications. For example, Kanev et al. report that binaries running in Google's data centers frequently exceed 100 MB [12]. We therefore selected two larger (millions of assembly lines) programs to include in our evaluation.

These larger applications are challenging for search-based methods such as POWERGAUGE because the effective search space grows rapidly with the length of the program (discussed in Section 5.4). We consider two approaches to address search space size, using two insights about mutated assembly programs. First, many such programs fail to assemble (e.g., because of duplicate labels [13]), and second, many mutated programs are functionally equivalent [14]. We remove both invalid and duplicate programs before they are evaluated on test cases, allowing scalability to larger problems in some instances.

Accurately measuring the energy use of a program poses an additional challenge because the energy usage triggered by a program is not restricted to the CPU. Although stochastic search often performs well on noisy problems [15], [16], [17], Haraldsson and Woodward highlight some difficulties encountered with models that estimate energy in our problem setting [18]. They observed that energy models do not always accurately capture all components (e.g., memory, disk drives, network cards, fans, etc.) affected by software. Model accuracy can also diminish on programs containing unintuitive optimizations such as those found by a stochastic search, leading to both false positives and false negatives. We therefore address the issue of developing accurate measurement tools and techniques for use in the fitness function of the genetic algorithm (GA) search.

The main contributions of this article are as follows:

- POWERGAUGE, a post-compiler method for optimizing non-functional properties of assembly programs via an explicit multi-objective search that considers both functional fidelity and energy consumption.
- An empirical evaluation of POWERGAUGE using large- and small-scale applications representative of data center applications. We find that our technique reduces energy consumption by 14% over and beyond `gcc -O3` for fixed output quality and 41% when relaxing the output quality.
- An exploration of techniques for managing search space explosion due to large program sizes.
- A comparison of POWERGAUGE to existing software techniques for reducing energy consumption.

The work reported here builds on Schulte et al.'s earlier work that used a GA to find post-compiler optimizations for assembly programs [19]. Here we extend that work to include explicit use of multi-objective search, additional search-space reduction and noise mitigation techniques, and improved energy measurements.

The rest of the article is organized as follows. Section 2 places POWERGAUGE in the context of related work. Section 3 presents a motivating example that describes an ideal solution for post-compiler energy reduction and illustrates the difficulties associated with physically measuring whole-system energy consumption. Next, Section 4 describes our proposed power measurement system that addresses these difficulties. In Section 5 we detail our multi-objective GA, highlighting several algorithmic features that improve its efficiency. Section 6 describes our research questions and experimental setup. This includes an explanation of our benchmarks and our process for determining human acceptability. Section 7 reports our results, including a quantitative exploration of tradeoffs. We also present a qualitative analysis of some of the successful optimizations that POWERGAUGE found and discuss some opportunities for improving the search based on the structure of certain benchmark programs. Section 8 discusses the remaining benchmarks, the optimization of energy vs. runtime, and threats to validity. Finally, Section 9 concludes the article.

2 Background and Related Work

In this section we discuss three broad approaches to power improvement: GA-based techniques, semantics-preserving techniques, and approximate computing.

2.1 Genetic Algorithms for Power Improvement.

Researchers have recently begun investigating the potential of GAs and related methods for reducing software energy consumption [19], [20], [21]. In general, these approaches are given a program to optimize. They then generate an initial *population* of programs by *mutating* the original, that is, by making small random modifications to it. The algorithm evaluates the *fitness* of every *individual* in the population using a *fitness function*, which assesses the individual according to the optimization goals. For complex properties or those that are time consuming to measure, such as program energy use on all workloads, a model or approximation is frequently used. The main loop of the GA selects individuals with high fitness, applies additional mutations and recombinations, and adds these new individuals to the population. The algorithm may replace random individuals or those with poor fitness immediately (steady-state GA), or wait to replace the entire population at once (generational GA).

Schulte et al. introduced the use of GAs for post-compiler optimizations of non-functional software properties [19]. This work was evaluated on relatively small benchmarks, used an energy model to estimate fitness, and followed up with a wall socket energy reading of the best individual found. Our work extends this technique to production-scale programs, such as `libav`[1] and `blender`,[2] by introducing search-space reductions and profiling. In addition, we directly address the concerns raised by Haraldsson and Woodward [18] regarding accuracy of software energy measurement in the context of stochastic search. By using specialized hardware to measure whole-system power we gain confidence that our results represent real-world reductions in energy consumption.

Bruce et al. [20] also use genetic improvement in the context of reducing energy consumption. Their technique, however, relies on estimated energy consumption provided

1. https://www.libav.org/
2. https://www.blender.org/

by the Intel Power Gadget API. This estimate includes only the CPU and does not model the energy consumed by other components of a system, such as memory or I/O devices. Our work uses whole-system power measurements and the search is not restricted to CPU-specific energy optimizations. Additionally, while their technique modifies software at the source code level, ours operates on assembly output from the compiler. This more easily supports combining energy optimizations with standard compiler optimizations.

Other work used GAs to reduce the energy consumption of GUIs in Android applications [21] but is specific to mobile devices that use Organic Light-Emitting Diode (OLED) screens. This work modeled energy consumption as a function of the color components of pixels on OLED screens, which provides an accurate estimate of energy consumption for some applications. However, it is domain-specific and does not generalize to other use cases.

2.2 Semantics-Preserving Techniques

While GA-based approaches typically apply random transformations to the program and assess their functional correctness via test cases, semantics-preserving approaches are designed by construction to retain required functionality.

2.2.1 Superoptimization

Superoptimization techniques [22], [23] check large numbers of sequences of assembly instructions to find an optimal execution sequence. These techniques are similar to ours in that both may change the implementation. However, superoptimization techniques scale only to short sequences of assembly instructions, while our approach operates on entire programs. Superoptimization and our approach are both assembly-to-assembly transformations, but they are independent and could be composed together in any order.

2.2.2 Profiling and Profile Guided Optimization

Profiling has been used in the past to guide optimizations [24], [25], [26]. These techniques use profiles collected from previous runs of a program when applying optimization to create more efficient code in the context of a typical program execution. These methods are typically semantics preserving. By relaxing semantics, we can use profiling similarly to guide the search to locations in the code most likely to have a large impact on energy consumption (i.e., are executed most often) and create novel optimizations that are not possible with strict semantics-preserving transformations.

2.2.3 Hardware Techniques

Hardware techniques for power reduction such as voltage scaling and resource hibernation [5] or clock gating [27] do not modify the subject program and are independent of source optimization techniques. By combining hardware optimization and POWERGAUGE or other optimization techniques, it should be possible in practice to create an even more efficient system.

2.3 Approximate Computing

Approximate computing has emerged as an alternative approach to decreasing runtime and energy consumption [28], [29]. By trading off some computational accuracy, approximate computing allows for reduced runtime or energy consumption, similar to how lossy compression trades off quality for space efficiency.

2.3.1 Task Skipping

Task skipping [30], [31] uses models that characterize the tradeoff between accuracy and performance to skip or halt the execution of some tasks. In task skipping, the developer manually partitions the program into task blocks that the execution environment can terminate to reduce accuracy while increasing performance. Although task skipping can be effective, decomposing a problem into tasks requires manual intervention, domain-specific knowledge, and it may not always be possible [32].

2.3.2 Loop Perforation

In loop perforation, individual loop iterations are skipped to reduce computational load with the goal of reducing computation time, energy consumption, or responding to faults. Loop perforation is particularly effective in algorithms that use looping to iteratively improve the accuracy of a calculation [10], [33]. Truncating or skipping iterations of these loops can allow a program to calculate an approximate answer with reduced computation. Loop perforation does not require the same domain-specific knowledge as task skipping, but not all programs can be optimized with loop perforation. In some cases, loop perforation leads to decreased performance if, for example, a loop is used to filter out data before an expensive evaluation step [33]. We compare our technique explicitly against state-of-the-art loop perforation techniques in Section 7.3.1, but also note that both techniques could be used together.

2.3.3 Precision Scaling

Precision scaling improves efficiency by altering arithmetic precision [34], [35]. Adjusting variable precision can expose optimizations or make more efficient use of hardware. For example, rounding floating point values near one can completely optimize out an expensive floating point multiplication. Additionally, scaling the precision of data can change memory layout and lead to better cache performance as alignments change.

2.3.4 Approximate Hardware Techniques

Hardware-based approximate techniques, such as approximate adders [36], [37] or multipliers [38], also aim to trade off accuracy for reduced power consumption or computation time. By allowing for error in computations, hardware designers are able to reduce the number of transistors in circuits, reducing power dissipation and gate delays. Like semantics-preserving hardware optimizations, approximate hardware optimization techniques are independent of software-based techniques such as POWERGAUGE.

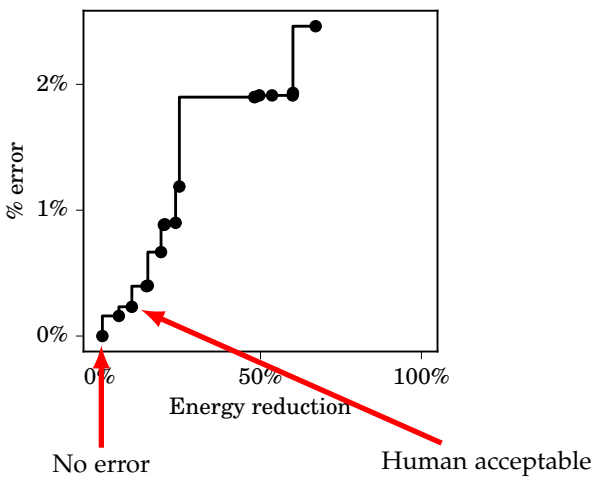

Fig. 1: Pareto frontier for the `blender (car)` benchmark. The X axis indicates percentage energy reduction and the Y axis indicates percentage error. The point in the lower-left has no error and corresponds to a 1% energy savings. The point in the lower-right was judged to be human acceptable and corresponds to a 10% energy savings.

3 Motivation

Energy consumption is a significant cost for data center scale computing. With power for American data centers projected to reach an annual cost of tens of billions of dollars [3] in the coming years, companies have already begun taking steps to reduce energy expense. A recent example is Google's $2.5 billion investment in wind and solar farms near their data centers.[3]

In addition to hardware and compiler techniques for reducing energy consumption, there is a need for software modifications to further reduce these costs. In this article, we present PowerGauge, a mostly-automated technique and prototype implementation for exploiting opportunities for relaxing output quality to reduce energy consumption. PowerGauge takes as input compiled assembly code and an existing test suite, and produces an optimized program with the same behavior but reduced energy requirements. In this scenario, PowerGauge imposes little additional burden on developers, since they may reuse existing tests and need only adapt their build process to produce assembly files and provide a mechanism to measure energy consumption. (We discuss one possible mechanism in Section 4.)

To achieve even greater energy reductions, the test suite may be augmented with a metric that estimates the quality of the output (instead of merely *pass* or *fail*), enabling PowerGauge to search for programs that optimize for energy consumption while allowing for small differences in output. We give some examples of simple, yet effective metrics in Section 6.1. With such an augmented test suite, PowerGauge produces a list of Pareto-optimal programs that trade off energy consumption and error. In this case, the developer or end user may select the program that provides the most desirable balance for their particular use case.

3. https://www.google.com/green/energy/, Dec. 2016

We present an example of the Pareto-optimal output from PowerGauge in Figure 1. To create this figure, we applied PowerGauge to `blender`, a 3D computer graphics application, and plotted the error vs. energy reduction of the Pareto-optimal programs. Each point in the figure represents an optimized program. The programs associated with the points in the lower left minimize differences from the output of the original program and demonstrate correspondingly conservative energy reductions. For example, the image generated by the `blender` version associated with the point labeled "No error" is identical to the image produced by the original program, but this program version has a 1% energy reduction over the original. The points in the upper right of the figure represent programs with both greater energy reductions and larger impact on the output quality.

The primary manual involvement required from PowerGauge users is to select the program that shows the largest reduction in energy use while maintaining an acceptable level of quality. In this example, the image generated by the program at the point labeled "Human acceptable" was subjectively judged to contain an allowable level of error. This program incorporates optimizations that reduce energy consumption by 10% over the original program. Since PowerGauge creates a sequence of modified programs, software engineers or end users can choose their own tolerance for error and deploy the corresponding binary.

PowerGauge operates on assembly programs and representative test inputs, artifacts that are often already available or are easily introduced to many development processes. It also requires two mechanisms that may be less commonly available: one for measuring energy and another for estimating output quality. We discuss the latter in Section 6.1. We present our approach to the former in the following section.

4 Power Measurement

To optimize the energy usage of a program, our search requires a fitness function to estimate the energy consumption of each individual. To achieve this, we require a mechanism capable of measuring whole-system energy of individual servers without requiring hardware modifications. This apparatus must also have suitably fine-grained time and energy resolution and a reporting rate that does not greatly increase our search times. Additionally, to minimize noise due to overhead on the system under test, we require that the device be entirely self-contained without relying on monitoring software running on the same system. As a practical matter, we also require it to be sufficiently cost effective to run several experiments in parallel, allowing one to take full advantage of the independence of fitness evaluations in a GA.

4.1 Existing Approaches

Previous work discussed in Section 2 relied on energy models to guide the search. For example, Schulte et al. [19] used a linear combination of performance counters for the fitness function, measuring power empirically only at the end of the search. However, performance counters (e.g., instruction

counts) cannot account for energy differences that arise from optimizations such as instruction reordering [39]. In a preliminary investigation, we found that these potentials for model inaccuracies can have a negative impact. Specifically, using the model suggested by Schulte, we identified a variant of `freqmine` with over 70% predicted energy improvement, but only 2% actual improvement. These discrepancies between predicted and measured benefit motivated us to find an improved power measurement system.

One common alternative to energy models is direct measurement. For example, many commercial devices, such as the Watts up? PRO meter, can inexpensively measure and report energy consumption. These off-the-shelf meters are simple to install and easy to use: the system under test is plugged into the device and energy consumption is reported over USB. However, these devices are typically designed for long-term monitoring and are not for capturing rapid changes such as those caused by relatively short program executions. POWERGAUGE typically compiles and evaluates tens of thousands of candidate programs during a search, and the limited (1 Hz) reporting rate of the Watts up? PRO creates delays that greatly increase the time required for a search (see Section 4.2).[4] Note that sampling rate is distinct from reporting rate; sampling rate refers to the rate at which a signal is measured, while reporting rate refers to the rate at which the samples are sent to the system monitoring the measurement device.

Although some specialized solutions for measuring energy exist, we were unable to find one suitable for our application. LEAP [40] and similar projects [41] use a specialized Android platform to measure energy consumed by mobile devices, and could not be directly adapted to monitor server systems. JetsonLeap [42] is designed to accurately measure power consumption to enable power-based compiler optimizations, however it requires that the system under test have a general-purpose I/O (GPIO) port. GPIO ports are common in System-on-a-Chip devices such as the Arduino or BeagleBone, but are rare on server systems. Similarly, the Monsoon Power Monitor[5] is only designed to measure power on mobile devices rated to 4.5 V and costs $771 to measure a single device. Other approaches require a separate measurement PC and either only monitor CPU power [43], or measure whole-system energy but require specialized boards to be installed on power lines inside the device under test [44].

The search technique used by POWERGAUGE is independent of the measurement device and could be adapted for use on mobile applications using the devices described above. However, for our domain, an ideal solution would be simple to install and capable of measuring energy consumption directly on an unmodified production server. To our knowledge, in 2017, there is no cost-effective commercial solution available that meets all of these needs.

4. Since we must wait for a report from the energy meter before we start a program under test and also wait to collect the first report after the program has terminated, the 1 Hz reporting rate of the Watts Up? PRO corresponds to a $0.5\,\text{s} \times 2 = 1\,\text{s}$ overhead per fitness evaluation. If our GA conducts 65,536 fitness evaluations, this overhead amounts to an additional 18 hours; a meter reporting at 0.1 Hz would produce just under 2 hours of overhead instead.

5. https://www.msoon.com/LabEquipment/PowerMonitor/

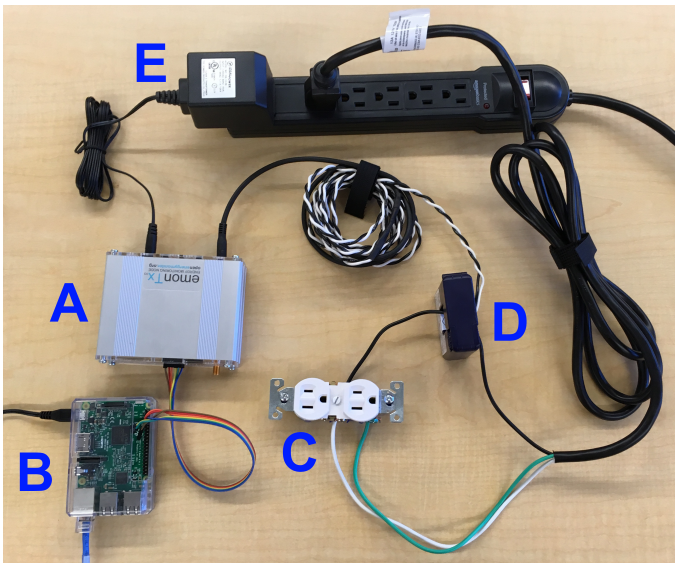

Fig. 2: Energy meter setup. A: emonTx V3 energy monitoring node. B: Raspberry Pi. The Raspberry Pi is connected to the emonTx by UART (multicolored cable), and reports via ethernet (bottom cable). C: Monitored receptacle. D: Accu-CT ACT-0750 5A current transformer. Note that the current transformer is around the hot (black) wire attached to the receptacle. Up to four current transformers can be monitored by the emonTx at once. E: AC-AC voltage adapter. The emonTx uses this voltage adapter both for power and for monitoring the voltage present at the receptacle.

4.2 Measurement Apparatus

We addressed this issue by designing a simple, inexpensive monitoring system based on available components that provides the resolution and the sampling rate required for a scalable genetic search. This device only needs to be constructed and calibrated once and can then be used to monitor the whole-system energy of any machine by simply plugging its power into the device and connecting the device to the network. In the remainder of this section, we describe our monitor along with the configuration and framework with which we made it scalable.

We based our design on the emonTx V3 energy monitoring node.[6] This open source design consists of an ATmega328 microcontroller with sockets to connect an AC-AC voltage adapter and up to four current transformers. The microcontroller is programmable using the Arduino API.[7] The current transformers read the varying amperage on up to four separate lines while the voltage adapter reads the varying voltage from the same power source (see Figure 2). We evaluated several different current transformers and chose Accu-CT ACT-0750 current transformers[8] rated for 5 A with the 1 V output option because our testing showed that these gave us the most precise measurements in the range of powers used by our systems (i.e., between 40 and 100 W).

6. https://openenergymonitor.org/emon/modules/emonTxV3
7. https://www.arduino.cc/en/Main/Software
8. https://ctlsys.com/act-0750_series_split-core_current_transformers/

Although this baseline hardware provides a cost-effective solution for high resolution time and energy measurements, we found that the default firmware needed to be completely rewritten to meet our requirements. Our software running on the microcontroller combines the signals from the current transformers with the voltage reading from the AC-AC voltage adapter to compute the real power on each line. This power is reported via the on board UART serial device. Our present prototype implementation is capable of reading inputs from the four current transformers and the voltage adapter at a sampling rate of about 1200 Hz, which is significantly faster than can be transmitted via the serial controller. We therefore aggregate a configurable number of measurements together and report the average power usage less frequently. For all experiments in this article, the microcontroller sampled at 1200 Hz and reported measurements on the serial bus at 10 Hz. This is ten times faster than the reporting rate that was possible with the Watts up? PRO energy meters, and supports measuring energy consumption at a rate that makes large-scale searches feasible.

The code to convert the integral sensor readings into floating point current and voltage readings requires coefficients to scale the values properly. We calibrated these using a Watts up? PRO device as a baseline. Although the Watts up? PRO is not suitable for fitness evaluations, as a commercially calibrated meter, it is suitable for use as a baseline for calibration, which can tolerate slower responses. Note that properly calibrating real power measurements requires a resistive load, such as a high-wattage light bulb or small heating element, so that real power and apparent power are equal [45, § 8.4]. We used a lamp with three 40-watt incandescent light bulbs to produce a large enough load for the limited power resolution of the Watts up? PRO to provide four significant digits. After calibration we collected 2500 readings of the resistive load to confirm that the power readings from the microcontroller were reliable. We confirmed that they were approximately normally distributed (Shapiro-Wilk normality test, $p > 0.1$: large p-values fail to reject the null hypothesis that the distribution is normal [46]) and showed a small standard deviation relative to the average value (about 0.7%).

To support running experiments in parallel and measuring the power usage of multiple machines, we designed our solution to make the multiple separate energy measurements from a single microcontroller available via ethernet. This allows the energy measurements to be dynamically distributed to different machines as necessary. We accomplished this by connecting the output of the microcontroller to the GPIO pins of a Raspberry Pi 3.[9] We wrote a simple tool to read the power measurements from the GPIO pins, multiply by the time since the last reported measurement to obtain energy, and make the result available via TCP/IP. Note that the Raspberry Pi is simply a convenient system to distribute the energy readings. It is straightforward to connect the microcontroller to any system with a USB port using a UART to USB cable, allowing that system to distribute measurements or simply consume them directly.

This setup permits the following procedure to measure the energy consumed by a process: On the machine that will run the process, (1) establish a TCP/IP connection to the Raspberry Pi to receive continuous energy measurements, (2) launch the process and record the energy while waiting for it to complete, and (3) close the TCP/IP connection. Although it seems that the overhead from networking could impact our measurements, Langdon et al. have shown that this effect is negligible at the energy and time scales of our benchmarks [47]. All the code for the microcontroller, the server process on the Raspberry Pi, and a simple script to measure the energy of a process are available from a GitHub repository.[10] The completed apparatus, with capacity to measure the energy consumption of 10 machines, can be seen in Figure 3.

As of 2017, the hardware required to monitor the power consumption of a single machine costs $244. However, a single emonTx v3 node can simultaneously measure four different current transformers. Thus, the additional cost of measuring up to three more machines is only $47 per current transformer. The final hardware cost to monitor four machines is $385, just under $100 per machine. The custom firmware for this hardware outputs data that is directly compatible with POWERGAUGE and no additional support hardware or software is required.

This system provides fast, reliable measurement of a constant load, showing less than 1% deviation in the measurement of reference light bulbs. However, the energy usage of a computer system is much more complicated. In the next section, we discuss measures to stabilize the system load.

4.3 Configuring the System to Minimize Noise

In addition to precise measurements of energy consumption, we also required a low level of noise in the energy consumed by the systems running the search. That is, the system should add minimal variation to the measured energy; the less variation that must be ascribed to the system, the more certain we can be that any variation in energy level is due to the optimization.

All systems used for these experiments were purchased at the same time, to the same specifications, from a single distributor. We used the Ubuntu Server 16.04.1 image, to which we added only the packages (mostly libraries) needed to compile our benchmarks. Using a relatively small base distribution and adding only minimal packages reduced the chance that an unexpected cron job or daemon service would run in the middle of a fitness evaluation, adding an unexpected additional energy cost to the individual being evaluated. Next, we disabled dynamic frequency scaling governors by adding the argument `intel_pstate=disable` to the kernel boot arguments. Frequency scaling is used by the system to adjust power consumption (see Section 2.2.3), which can cause POWERGAUGE to interpret scaling as an optimized individual. After the search, frequency scaling can be reenabled to take advantage of the additional power savings.

However, hardware variation can add significantly to the variability of energy readings between repeated executions of the same (deterministic) program. Over the course of

9. https://www.raspberrypi.org/

10. https://github.com/dornja/powergauge

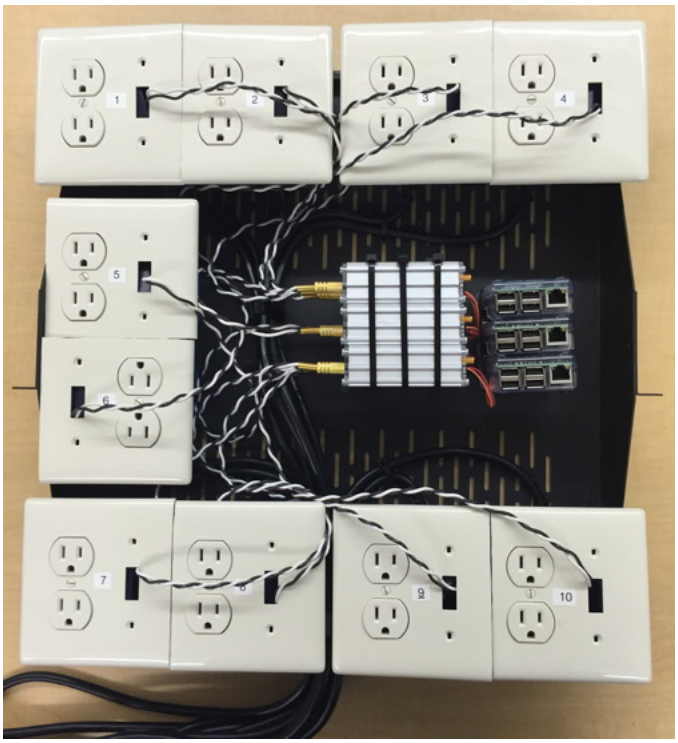

Fig. 3: Final assembly of 10 circuits to be read by three emonTx V3 devices, connected to three Raspberry Pis. Each wall socket housing contains a current transformer clipped around the hot wire and connected via the black and white twisted wires to an emonTx (center). Each emonTx measures the power usage of up to four connected wall sockets and is connected to a Raspberry Pi (center right). Each Raspberry Pi computes the energy usage for each channel of the connected emonTx and the readings are available via TCP/IP over ethernet.

preliminary runs, we detected unexpected variance that we were unable to attribute to our measurement equipment, the operating system, or frequency scaling. We narrowed down the problem to energy usage variations on different CPU cores. In particular, the average energy consumed while running the benchmark on one core could be significantly higher than when running the same benchmark on another core. For example, on one of our systems we found about 0.4 J difference between running the benchmark on cores 5 and 6 (we restricted the OS to schedule only normal tasks to core 2 in both cases). Since the measurements on the same cores showed a standard deviation around 0.2 J, allowing programs to be scheduled to either core would represent a 100% increase in the measurement uncertainty.

To mitigate this source of uncertainty, we restricted the scheduler to always assign the fitness evaluation to one core on a given system and to assign all other processes to a second core. To determine which cores to use for each experimental system we measured the energy used while running a particular benchmark under each possible allocation of two cores. We then selected the combination with the smallest variation and used it for all experiments on that system. Although we carefully restrict the scheduler and select cores during the search, the subsequent evaluation

Input: p : *Program*
Input: FITNESS : *Program* $\rightarrow \mathbb{R}^n$
Input: *MaxCount* : $\mathbb{N}$
Input: *PopSize* : $\mathbb{N}$
Output: *Front* : Pareto frontier of *Program*s

function POWERGAUGE(p, FITNESS, *MaxCount*, *PopSize*)
  $P \leftarrow \{\}$
  ADDTOPOP(P, p, FITNESS)
    ▷ Adds p to P and evaluates its fitness
  count $\leftarrow 1$
  while $|P| <$ *PopSize* **do**
    $q \leftarrow$ MUTATE(p)
    ADDTOPOP(P, q, FITNESS)
    count $\leftarrow$ *count* $+ 1$
  Ranks $\leftarrow$ NONDOMINATEDSORT(P)
  while *count* $<$ *MaxCount* **do**
    $Q \leftarrow \{\}$
    while *count* $<$ *MaxCount* $\wedge$ $|Q| <$ *PopSize* **do**
      $p_1, p_2 \leftarrow$ TOURNAMENT(P, *Ranks*, 2)
      $q_1, q_2 \leftarrow$ CROSSOVER(p_1, p_2)
      $r_1, r_2 \leftarrow$ MUTATE(q_1), MUTATE(q_2)
      ADDTOPOP(Q, r_1, FITNESS)
      ADDTOPOP(Q, r_2, FITNESS)
      count $\leftarrow$ *count* $+ 2$
    Ranks $\leftarrow$ NONDOMINATEDSORT($P \cup Q$)
    $P \leftarrow$ *Ranks*$[1 :$ *PopSize*$]$
  Front $\leftarrow \{\}$
  for all $q \in$ GETFRONTIER(P) **do**
    Front $\leftarrow$ *Front* $\cup$ MINIMIZE(q, FITNESS)
  return *Front*

Fig. 4: POWERGAUGE Optimization algorithm.

demonstrated that the discovered optimizations generalize to other cores and other machines with the same hardware.

4.4 Remaining Sources of Noise

The measures described above mitigate most of the noise we observe. However, they do not eliminate noise completely. Some remaining potential sources of noise include environmental factors such as ambient temperature, the physical limitations of the measuring device, periodic system maintenance tasks scheduled by the Linux kernel, scheduling delays between opening the TCP/IP connection and starting the subprocess, and communication delays on the TCP/IP or serial communication channels. This remaining noise level is low enough that the GA used by POWERGAUGE is able to effectively search for energy reductions (several works have shown that GAs can be effective even with noisy fitness functions [15], [16], [17]), but we still would like to gain confidence that the modified programs have a statistically significant energy savings. We discuss the sampling step used by POWERGAUGE in Section 5.3.

5 POWERGAUGE OPTIMIZATION ALGORITHM

POWERGAUGE extends the Genetic Optimization Algorithm (GOA) presented by Schulte et al. [19]. Like GOA,

POWERGAUGE operates on the compiled assembly program representation after compiler optimizations have been applied. We use this program to seed an initial population for the GA as described in Section 2.1. To evaluate the fitness of each individual in the population, we assemble it into a binary and execute the binary on a representative workload, measuring energy using the meter described in Section 4. Unlike GOA, which rejected any program that failed to produce identical output to the original, our approach recognizes that in some situations, trading slight differences in output for further decreases in energy usage may be desirable. We incorporate a multi-objective GA to optimize this tradeoff explicitly.

As discussed in the introduction, a significant motivation for our work is the optimization of data center applications. Although GOA was successful on smaller (100 kLOC) benchmarks, the algorithm as presented does not scale well to larger (10 MLOC) programs. For example, GOA running on the `blender` program described below was unable to construct the initial population before running out of memory on our 16 GB system. Therefore, we adopted a more memory-efficient representation for the experiments reported here.

The following subsections detail our program representation and mutation operators (Section 5.1) and the multi-objective GA (Section 5.2). In Section 5.3, we describe our algorithm for minimizing differences between the original and optimized programs while retaining the optimizations. Finally, we elaborate the techniques we developed to better manage the search space induced by large programs in Section 5.4 and Section 5.5.

5.1 Program Representation

We represent programs in POWERGAUGE as a list of edits to apply to the original program, similar to the "patch representation" used in population-based program repair techniques [48], [49]. Unlike the GOA representation, which maintains an in-memory copy of the complete source for each individual in the population, our representation requires only one shared copy of the original among the entire population. Edits are applied to the shared representation as each individual is serialized to disk for fitness evaluation. Since the number of edits applied to any individual is small relative to the size of the program, this results in significant memory savings.

Our technique also relaxes the assumption in GOA that programs consist of a single assembly file. Our implementation can handle multiple assembly files, greatly simplifying the process of preparing a program for optimization. Combining assembly files for a large project into a single file is non-trivial, e.g., the `gcc` "–combine" flag suggested for use with GOA is no longer available in recent versions of `gcc`. This change enables POWERGAUGE to store only in-memory copies of files for which an edit is present in the population; unmodified files can be reused directly on the disk, providing further memory savings.

Following Schulte et al., we consider three types of edits (mutations) to the program:

- *Delete* deletes a line in the program.
- *Swap* exchanges the positions of two lines in the program.
- *Copy* duplicates a line in the program and inserts it at a random location.

Our implementation uses one-point crossover on the genome (list of edits). A random point is selected in each of two individual genomes, and a new genome (child) is constructed by combining the edits from the first portion of the first representation with the second portion of the other. A second child is constructed similarly, using the first portion of the second representation and the second portion of the first.

5.2 Multi-Objective Search

Our GA is described in Lines 2 to 20 of Figure 4. We require as input the program p to optimize as well as an n-dimensional fitness function. In our experiments, $n = 2$ and the function computes the energy used by the program and the error, if any, in its output. Our algorithm also has two parameters to manage the size of the search: *MaxCount* indicates the total number of fitness evaluations to conduct and *PopSize* indicates the number of individuals to maintain in the population. In Lines 2 to 8, we initialize the population with the original and $PopSize - 1$ mutants and evaluate their fitness.

The main body of the GA consists of Lines 9 to 20. We optimize both energy and error simultaneously to identify those modifications to the original program that provide the best tradeoffs between them. That is, we compute a Pareto frontier, the set of modified programs such that every other program in the search has worse energy or error (or both). To this end, our multi-objective algorithm uses the non-dominated sort from NSGA-II [50]. This sorting defines a pair of values for each member of the population based on their fitness values. In the tournament selection on Line 13, these pairs are compared lexicographically to identify the tournament winner. At a high level, the first element of the pair indicates how close the individual is to the Pareto frontier while the second gives greater weight to individuals in less well-represented parts of the frontier.

After selecting two individuals using the above tournament selection procedure (two tournament rounds producing two winners), the crossover operator is applied with 50% probability. Next, the mutation operator is applied to each child, and finally the fitness of the mutated children is evaluated. Once *PopSize* new individuals have been generated, we then select the elements of the population the next generation. Our algorithm is elitist, considering individuals from both the previous and current generation together. Out of these $PopSize$ individuals, the best *PopSize* new or old individuals are selected, according to the NSGA-II ranking, which forms the next generation.

5.3 Edit Minimization

As was the case with GOA, optimized individuals likely include edits that do not impact the energy use or measured error of the program on the workloads used by the fitness function. We would also like to gain confidence that the energy reductions measured during the POWERGAUGE

search are true energy optimizations and not artifacts of noise sampled during the search. We therefore use the same strategy as GOA by including a final minimization step augmented with sampling for each individual on the Pareto frontier (Lines 21 to 23) to eliminate spurious edits that do not contribute to the fitness metrics.

Our minimization algorithm uses Delta Debugging [51], which takes as input a set of edits and identifies a 1-minimal subset of those edits that maintains the optimized performance as measured by the fitness function. The Delta Debugging algorithm is linear time in the original number of edits and requires evaluating the fitness of a new collection of edits at each step.

Because our energy measurements are stochastic, we collect several samples and apply a one-tailed Wilcoxon Rank-Sum Test [52] to determine whether the distribution of fitness values is worse than the distribution of values collected for the optimized variant. If the test for either objective indicates a significant difference between the distributions ($p < 0.05$), we treat that variant as "unoptimized." In all experiments described in this article, we collected at least 25 fitness samples for each Delta Debugging query, increasing this number as necessary to increase the power of the statistical test, always maintaining relative standard error below 0.01. We found that starting with 25 energy measurements and using the relative standard error threshold provided a good tradeoff between runtime and smaller minimized genomes. At the end of the Delta Debugging stage, we are left with a 1-minimal set of edits to the original program that results in a statistically significant reduction in energy consumption.

5.4 Search Space Reductions

The search space for our algorithm consists of all programs generated by a finite series of edits. Since *Copy* and *Swap* each involve two independently selected instructions, the search space for a program with n instructions using up to k edits contains $\mathcal{O}(n^{2k})$ programs. For example, the `libav` benchmark compiles into 23 M lines of assembly code. Since *Copy* can copy any line to any location, this means that there are 5.2×10^{14} possible copies to consider. Many of these copies can result in programs that fail to build (e.g., duplicate labels in a single file) or almost certainly have no or negative effect (e.g., copying an instruction into the data segment). Writing, assembling, and running programs that cannot have improved fitness is wasted effort for the search. We investigate several ways to avoid this cost.

- We remove duplicate instructions from the set of instructions that *Copy* can insert. The magnitude of this reduction can be seen in the "Unique Lines" and "% Unique columns" of the Table 1. It is most dramatic for the largest benchmarks, because redundancies increase with program size.
- We disallow the insertion of code directly after assembly directives or labels that are not used as jump targets. For example, inserting an instruction after line 1 in Figure 5 has the same effect as inserting it after lines 2 or 3 (although this may be difficult to determine manually given the alignment directives). This reduction considers only the insertions after line 1.
- We disallow copying or deleting labels. Either copying a label into a file where it already exists or deleting a label referenced by some instruction produces a program that fails to build. Copying a label into a file where it is never referenced produces no semantic change.
- We allow *Swap* to operate on labels only if the exchange is limited to a single file. Swapping a label into a different file would either have no effect (if it is swapped with an identical label in another file) or fail to build (because a needed label was removed from a file).

```
1       jle .L91
2       .p2align 4,,10
3       .p2align 3
4       movl    %r13d, (%r14)
```

Fig. 5: Sample of assembly from `blackscholes` benchmark. Copying an instruction after line 1 has the same effect as copying it after lines 2 or 3 instead.

5.5 Profiling

Our search space reductions remove from consideration programs that our simple static analyses show cannot improve fitness. However, they still allow edits to portions of a program not visited by the target workload. For example, inserting an instruction into a program region that is not executed is unlikely to affect energy usage. We therefore investigate the use of execution profiles to capture the runtime behavior of the program and increase the probability of making edits that impact fitness. The insight behind our use of execution profiles is that beneficial edits in more frequently-executed regions of the code are likely to have a greater impact on the fitness than beneficial edits in less-frequently executed regions.

There has been a significant amount of work using profiling information to guide software optimizations (e.g., [25], [26], [53]). GOA used sample-based profiling combined with Gaussian smoothing to collect approximate execution counts to guide the GA [54]. We observe, however, that a profile can be collected once and reused throughout the search, reducing the need for sampling. We used Pin [55] to measure the exact execution count of each basic block in the compiled binary and then extrapolated the execution count of each instruction. Although profiling with Pin does take time, we found this cost to be reasonable. Collecting profiles for each benchmark takes less than 300× the runtime of a normal execution. As can be seen in Table 1, our longest-running benchmark is `vips`, which normally executes in 18.1 seconds. We were able to collect its profile in approximately 90 minutes. To map between the instruction addresses from our Pin tool and the lines of assembly code, we used the output of GNU `objdump` to align instruction addresses with the contents of the assembly files first by function name, then by instruction mnemonic.

Benchmark	Assembly Lines	Unique Lines	Unique %	Executed Lines	Executed %	Tests (s)	Error Metric
blackscholes	12,437	3,504	28	637	5	2.7	RMSE
blender (car)	17,559,869	1,574,349	9	256,687	1	17.6	Lab distance
blender (planet)	-	-	-	221,397	1	10.6	Lab distance
bodytrack	198,462	62,544	32	23,746	12	3.3	RMSE
ferret	80,811	26,883	33	15,181	19	6.4	Kendall's τ
fluidanimate	7,511	4,436	59	3,828	51	2.7	Hamming distance
freqmine	26,281	12,115	46	10,404	40	7.4	RMSE
libav (mpeg4)	22,831,124	698,445	3	42,747	0	1.3	Lab distance
libav (prores)	-	-	-	34,634	0	2.7	Lab distance
swaptions	55,753	14,911	27	2,911	5	3.2	RMSE
vips	822,655	160,075	19	24,000	3	18.1	Lab distance
x264	205,801	58,754	29	41,063	20	5.7	Lab distance
Total lines	43,128,694	2,836,551		677,235			

TABLE 1: . Benchmarks used for our experiments. The "Assembly Lines" column shows the number of lines of compiled assembly for each benchmark. The "Unique" columns show the number and percentage of unique lines in these sources, while the "Executed" columns show the number and percentage of lines executed in each of the test cases. The "Tests (s)" column shows the runtime of a single test execution for each test input, while the "Error Metric" shows the error metric used for each test.

For the experiments described in Section 7.2 the probability that an instruction would be the target of a *Delete*, *Swap*, or *Copy* was simply the relative execution count of that instruction divided by the total number of instructions executed. (In the case of *Copy*, this weighting is used only to select the insertion location, not when selecting the line to be inserted.) This excludes unexecuted lines from the search space as well as directing the search towards more frequently executed lines. We show the number of instruction locations executed in each benchmark in Table 1. Overall, our profiling eliminated almost 99% of lines from consideration.

6 Experimental Setup

We investigate the following research questions:

- RQ1 Does PowerGAUGE discover energy reductions with human-acceptable levels of error?
- RQ2 Do the search-space reduction techniques (Section 5.4) and profiling (see Section 5.5) allow the search to find useful optimizations more effectively?
- RQ3 How do the optimizations found by PowerGAUGE compare to less-general techniques such as loop perforation?

As described in Section 5, PowerGAUGE contains several features designed to scale to larger programs. Our experiments investigate the optimizations it found and the time it took. To provide a meaningful comparison to previous work, we evaluate our technique on the PARSEC benchmark suite [11] used by loop perforation and by Schulte et al., and we evaluate on two larger applications.

6.1 Benchmarks

We chose `blender` and `libav` to investigate the scalability of PowerGAUGE. `blender` is a large 3D computer graphics application supporting a wide variety of tasks such as scene and character design as well as physics simulation and rendering and is used for visual effects in the movie industry.[11] `libav` is a collection of audio and video processing libraries for manipulating and encoding multimedia. It is a fork of the `FFmpeg` encoder, which is used as a backend for projects such as VLC and video streaming websites like YouTube.[12] Both rendering and encoding software is often used in large scale server environments, and `blender` and `libav` are mature, production-scale programs that we believe represent realistic targets for optimization using our tool.

Table 1 lists the benchmarks we used for evaluation. Besides two workloads for `blender` and `libav`, we included the PARSEC benchmarks [11] evaluated in earlier energy optimization experiments [10], [19]. We note that `libav` and `blender` together are 28× the size of the PARSEC benchmarks combined, allowing a more indicative assessment of PowerGAUGE's scalability to larger datacenter-scale applications [12].

PowerGAUGE considers both the energy used to run each test input and the error in the output produced. For all benchmark programs, we used the unmodified program to generate reference outputs, which we used to measure the output error of every individual during the search. Six of our benchmarks (two workloads apiece for `blender` and `libav`, plus `vips` and `x264`) produce image or movie files as their primary output. We treated each movie as a sequence of one image per frame. For all of these benchmarks, we convert the image from RGB to the Lab color space, a perceptually uniform color space [56]. We then compute the total Euclidean distance between all pairs of pixels in the output and reference images. The primary outputs for `blackscholes`, `bodytrack`, `freqmine`, and `swaptions` benchmarks are vectors of floating point numbers (integer in the case of `freqmine`). For these benchmarks, we simply compute the root mean square (RMS) error between the

11. http://blender.org/news/hardcore-henry-using-blender-for-vfx/
12. http://multimedia.cx/eggs/googles-youtube-uses-ffmpeg/

program output and the reference generated by the original program. The `ferret` benchmark computes a number of image similarity queries; the output consists of one list of similar images, ranked by similarity, for each query. Our error metric for `ferret` computes Kendall's τ, which quantifies the similarity of order between two sequences. Finally, `fluidanimate` writes out a C struct; since this admits less human intuition about the meaning of "acceptable" levels of error, we simply compute the Hamming distance between the two files.

Each benchmark has an associated indicative workload used as a target for our optimizations. In the case of the PARSEC benchmarks the indicative workload is induced by the test suite. For the larger applications, the indicative workload was adapted from provided tests to represent datacenter-scale use. For example, the `blender` indicative workload consists of rendering a single frame of a movie, because rendering each frame is logically independent and is typically carried out in parallel on separate machines [57].

Each benchmark is compiled to assembly using `gcc`; POWERGAUGE operates on the resulting assembly. For experiments that use profile information, the profile is gathered from an execution of the indicative workload (see Section 5.5).

6.2 Human Acceptability

In practice it is difficult to create an error metric that captures all of the properties that a human considers when making subjective judgments. For example, our simple error metric for images sums the Euclidean distance of each pixel in the Lab color space from its value in the original image. This error metric would assign a large error to an image that is slightly shifted from its original location, even though it would likely be considered acceptable to a human making a subjective judgment (see Section 7.1 for a concrete example of this scenario).

While in the particular domain of computer graphics there exist error metrics that can account for such motion (e.g., the Earth Mover's Distance [58] or a Structural Similarity Index Metric (SSIM) [59]), there are two main problems with a general approach to more precise error metrics. First, many of our benchmarks require domain-specific knowledge (e.g., domain-specific models of readability have required hundreds of annotators [60]), to judge acceptability and domain experts would be needed to train or create models. Second, since every individual created during a search must be measured with the error metric, we would like to minimize the time cost of measuring error by using efficient error models.

A strength of POWERGAUGE is that it can find optimized programs with human-acceptable levels of error despite imperfect error metrics, because GAs can tolerate a noisy fitness function [15], [16], [17]. A programmer can create a simple error metric for use with POWERGAUGE, then a domain expert can be consulted after the search to subjectively judge the outputs of 10-20 optimized programs created during the search, selecting the most efficient program with acceptable output.

The outputs of the programs on the Pareto frontiers discovered by the POWERGAUGE searches were manually inspected to judge human-acceptable levels of error in our benchmarks. Subjective judgment was relatively straightforward for `blender`, `bodytrack`, `libav`, `vips`, and `x264`, since all of these benchmarks output an image or video that can be directly inspected and compared to the original. Examples of output from these types of benchmarks judged to be human-acceptable can be seen in Figure 1 and Figure 6.

Other benchmarks required different strategies to judge acceptability. The `blackscholes` and `swaptions` benchmarks output a list of calculated prices and standard errors for financial instruments in a simulated market. The authors considered output to be acceptable if the calculated prices were within 5% of the values computed by the unmodified program. This 5% error bound is more strict than the 10% used by a similar study [33]; the POWERGAUGE user may choose any appropriate error level. The `ferret` benchmark takes a list of images as input and uses content-based search to find similar images in its database, outputting a ranked list of the top ten matches. Output of a modified program was considered human-acceptable if, for each of the 64 test queries, there were at least five images that appeared in both its output and the output of the original program. Finally, `freqmine` takes as input a list of tokens and outputs their frequency. We considered frequency counts within 5% of the correct value to be human-acceptable. The last benchmark, `fluidanimate`, did not allow for a subjective judgment of human acceptability. `fluidanimate` outputs a serialized data structure, and we were unable to manually inspect its output in a meaningful way.

Although the authors are not experts in image processing or finance, the output of POWERGAUGE allows developers to select the program that has the optimal energy consumption to error ratio for specific users or use cases. For example, an image encoded for viewing on small mobile screens and an image encoded for viewing on a large high-resolution display can likely contain different levels of error, and developers for each of these applications might use different optimized programs. Even if any error is completely unacceptable, in many cases POWERGAUGE is still able to find energy reductions.

6.3 Parameters and Hardware Specifications

We use a value of 512 for *PopSize* and apply exactly one mutation operator to each child after the crossover stage. The number of individuals created varied per benchmark — we targeted a runtime of approximately two weeks for each. With this target in mind, we chose 16,384 individuals for `libav (mpeg4)`, 32,768 individuals for all `blender` benchmarks and `libav (prores)`, and 65,536 individuals for the remaining benchmarks. Note that the total runtime of POWERGAUGE is not only a function of the runtime values in Table 1, but also of factors including the time to build and to estimate output error. It is also possible that a generated program can enter an infinite loop upon execution. Because of this, we set a maximum runtime of 60 seconds for each individual during fitness evaluation. Including test case timeouts to guard against infinite loops is a standard practice when using GAs to generate and validate new programs [61].

Energy measurements were made using the prototype apparatus described in Section 4.3 on a Dell PowerEdge

Program	% Energy Red. Baseline		% Energy Red. Best	
	0% Error	Accept.	0% Error	Accept.
blackscholes	91	91	**92**	**92**
blender (car)	0	0	1	**10**
blender (planet)	0	0	0	0
bodytrack	0	0	0	**59**
ferret	0	**30**	0	**30**
fluidanimate*	0	0	0	0
freqmine	0	0	8	**8**
libav (mpeg4)	0	0	0	**36**
libav (prores)	3	3	3	**92**
swaptions	39	**68**	39	**68**
vips	21	**29**	21	**29**
x264	0	**65**	0	**65**
average	13	24	14	**41**

TABLE 2: High-level summary of the energy reductions found by our technique. The "Baseline" columns show the results of the experiments when run without search space reductions or profiling, while the "Best" columns contain the best results of all experiments, including those run with search space reductions, profiling, or both. The results are subdivided into "0% Error", where the output of the optimized program is identical to the output of the original program, and "Accept.", where a human-acceptable level of error is allowed in the output. The `fluidanimate` benchmark has no acceptable error level because its output is a serialized binary and this error metric is not suitable for subjective assessment. We find optimizations in 10 out of 12 benchmarks and an average energy reduction of 41 when allowing for human-acceptable error and using search-space reductions and profiling.

R430 server with a 3 GHz Intel E5-2623 processor and 16 GB of ram.

All of our modified benchmarks and the results of our experimental runs are available online.[13]

7 Results

In this section, we address each of the research questions posed in Section 6.

7.1 Human Acceptability

To answer RQ1, we ran PowerGauge on each benchmark and measured the energy reductions found at 0% error. We then manually inspected the output of programs that had larger energy reductions, but contained some error and used the criteria described in Section 6.2 to determine if they were human-acceptable. The results of these first searches are shown in the "Baseline" columns of Table 2.

We first note that PowerGauge is able to find optimizations for the `blackscholes`, `libav (prores)`, `swaptions`, and `vips` benchmarks when the output is required to be identical to the original program. The "Baseline: 0% Error" column of Table 2 shows that PowerGauge reduced the energy consumption of our benchmarks by an average of 13% for this restrictive case. This result shows that PowerGauge is effective even when no reduction in output quality can be tolerated.

When allowing for an acceptable amount of error, PowerGauge finds additional energy reductions. For the `swaptions` and `vips` benchmarks, energy consumption was reduced by an additional 29% and 8% respectively. Energy reductions were also found for the `ferret` and `x264` benchmarks. For `blackscholes`, `bodytrack`, and `fluidanimate` PowerGauge found energy reductions when allowing for error, but we judged none of the outputs to be human-acceptable. On the remaining benchmarks, PowerGauge was unable to find any energy reduction without incorporating the algorithmic improvements discussed in Section 5.4 and Section 5.5. We discuss the results of augmenting PowerGauge with these features in Section 7.2.

An example of how allowing for a small amount of error can lead to additional energy reduction can be seen in Figure 6. We observe that the images are subjectively very similar, especially in a use case where the output is consumed by human eyes. The `vips` image has a slight vertical deformation or stretching. As a result, the pixels do not line up exactly and there is slight error in each individual pixel (as shown in the inset heat map). In this article we intentionally consider simple, indicative error metrics that represent a worst-case scenario for the search. Allowing error in the `vips` benchmark allowed PowerGauge to find an additional 5.3% energy reduction at the error rate corresponding to the image in Figure 6 when compared to the maximum energy reduction with no allowed error.

This case study provides a concrete example of PowerGauge enabling additional energy reductions with human-acceptable levels of error. This, combined with the positive results for `ferret`, `swaptions`, and `x264` leads us to conclude that PowerGauge can discover energy reductions with human-acceptable levels of error in many cases.

7.2 Search Space Reductions and Profiling

To address RQ2, we ran PowerGauge on each benchmark in four different configurations: without search space reductions or profiling, with search space reductions only, with profiling only, and with both. The best results from each of these four runs is shown under the "Best" columns of Table 2.

Two of the larger benchmarks, `blender` and `libav`, are representative of software that is normally deployed in server farms. Since these benchmarks have much larger codebases than the PARSEC suite, we were prompted to modify the PowerGauge algorithm to target scalability. The codebase for `libav` is particularly large, at 22.8 MLOC, and thus PowerGauge has an extremely large search space. We are able to greatly reduce the number of allowed edit locations by applying search space reductions and profiling as discussed in Section 5.4 and Section 5.5. For example, without these reductions PowerGauge can apply the *Delete* operator to any of the 22.8 MLOC in `libav`, but after profiling this is reduced to only 34,634 lines in the case of the `prores` input, a search space reduction of 99.8%. The results of searches with profiling and reductions are

13. http://dijkstra.eecs.umich.edu/projects/powergauge/results.tar.bz2

Fig. 6: `vips` comparison with allowed error. The image on the left is the reference and the image on the right is the output of an individual with 3% allowed error. Inset on the right is a comparison with red pixels corresponding to differences between the images (there is a slight difference in the vertical stretching between the two images). By allowing this level of error, POWERGAUGE is able to find a program that reduces power consumption by an additional 5.3%.

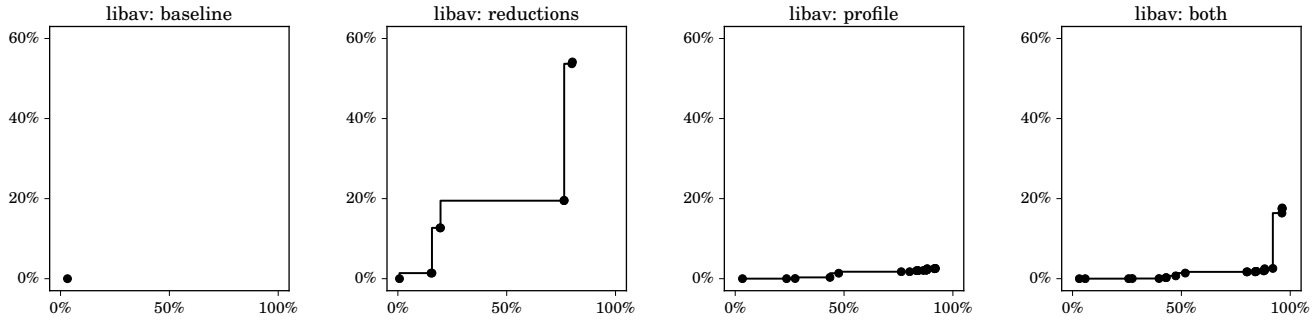

Fig. 7: Results of the multi-objective search on the `libav (prores)` benchmark when using POWERGAUGE with search space reductions and profiling. As in all Pareto frontiers in this article, the X axes indicate the percent energy reduction achieved and the Y axes indicate error. The leftmost figure is the Pareto frontier after running POWERGAUGE without search space reductions or profiling. From left-to-right, the next three figures are with search space reductions, with profiling, and with both.

shown under the "Best" columns of Table 2 and an example of improved Pareto frontiers from these searches is shown in Figure 7.

At 0% error, we observe that POWERGAUGE now discovers optimizations for the `blender (car)`, `freqmine`, and `libav (prores)` benchmarks. While using the combination of search space reductions and profiling and allowing for human-acceptable levels of error, POWERGAUGE finds greater improvements for the `blender (car)`, `bodytrack`, `freqmine`, and both `libav` benchmarks than without these algorithmic modifications. On average POWERGAUGE discovers energy reductions of 41 as compared to the 24% found when these optimizations were not used.

An example of the search improvements that result from these algorithmic changes can be seen in Figure 7. This figure shows the Pareto frontier results of searches performed on the `libav (prores)` benchmark. The leftmost image is the Pareto frontier after running POWERGAUGE without search space reductions or profiling, and from left-to-right the next three figures are with search space reductions, with profiling, and with both. Note that in these graphs, the X axes indicate the percent energy reduction achieved and the Y axes indicate percentage error using the per-benchmark error metric. An ideal optimization would introduce no error while using minimal energy and would therefore fall

Program	POWERGAUGE (%)	Loop Perforation (%)
blackscholes	96	92
bodytrack	82	62
ferret	86	70
swaptions	82	52
x264	85	50
average	86	65

TABLE 3: Percentage of the energy-error space dominated by the Pareto frontier induced by each technique. Larger values are better and indicate greater energy savings and/or lower error.

in the lower-right corner of our Pareto frontiers. The left-most image in the figure shows the results of a search that discovered only one program containing an optimization. The search corresponding to the second image from the left provides a developer with more options and allows one to choose a binary that outputs a lower-quality video but saves larger amounts of energy. The results of this search are not as desirable as the results of the two rightmost searches, corresponding to running POWERGAUGE with only profiling and with both profiling and search space reductions respectively. These two searches also allow for error vs. energy tradeoffs, but much less output quality must be conceded to achieve energy reductions equal to those provided by the previous search performed with search-space reductions alone (i.e., the points on the Pareto frontiers fall closer to the lower-right corner of the graph).

We note that combining both search space reductions and profiling does not automatically lead to the best results. For example, in our experiments with the `freqmine` benchmark, we observed an 8% improvement when using search space reductions alone; no improvements were found when profiling was used. We suspect that in this case profiling may have led the search *away* from locations in the code that lead to optimizations.

The improved search results shown in Table 2 and Figure 7 lead us to conclude that our search space reduction techniques and profiling allow POWERGAUGE to find useful optimizations more effectively.

7.3 Comparison to Other Approximate Techniques

In this section we address RQ3, and compare POWERGAUGE to other, more specialized approximate computing techniques.

7.3.1 Loop Perforation

We directly compare our results to the loop perforation techniques proposed by Sidiroglou et al. [10], which cites energy savings as a potential use of the technique. Because the paper only reports their results with respect to runtime, we reimplemented their technique and reevaluated while measuring energy consumption. For each benchmark for which their paper describes particular loops to perforate we replicated their technique as closely as possible. Loop perforation is performed by skipping iterations of a loop at a specified rate. For example, a perforation rate of 0.25 means that one out of every four loop iterations is skipped. For this comparison and following their paper, we enumerated all possible combinations of perforations of the specified loops at the previously published perforation rates (0.25, 0.50, 0.75, and 1 iteration). Since our input to the `x264` benchmark did not cover some of the specified loops, perforation had zero effect in several cases. To evaluate loop perforation's effectiveness on this benchmark more accurately, we compare our results on the `x264` benchmark to the results of applying loop perforation as specified in an earlier paper [33], which perforates functions that are covered by our input.

These results are shown via orange squares in Figure 8. Recall that in these graphs we would like optimizations to fall in the lower-right corner of the graph, simultaneously minimizing both error and energy consumption. Consider the `x264` subgraph and note that programs generated by POWERGAUGE are consistently closer to the lower-right corner than the programs generated by loop perforation, suggesting that we are able to find better optimizations. However, we acknowledge that it may be difficult to compare all of the graphs visually (e.g., the `bodytrack` and `ferret` subgraphs). Thus, we adapted the hypervolume indicator [62], a metric often used to evaluate the quality of a set of non-dominated points, to our experiments. To compare two Pareto frontiers, we calculate the fraction of the space between 0 and 100% energy and error that is dominated by the points in each Pareto frontier. A hypothetical optimization that achieved 100% energy improvement with no error would dominate all possible points and have a score of 1 by this metric. The original program dominates no points and has a score of 0. The values for loop perforation and our technique are shown in Table 3. We find that POWERGAUGE performs better than loop perforation on all the benchmarks for which we have both.

POWERGAUGE often finds optimizations in the form of loop perforations. In `blackscholes`, POWERGAUGE finds exactly the same power savings at zero error as loop perforation. This is because the `blackscholes` benchmark deliberately performs redundant calculations in a loop. Both techniques short-circuit this loop, but note that POWERGAUGE is able to find even more power savings than loop perforation when some error is allowed. In `bodytrack`, `swaptions`, and `x264`, POWERGAUGE finds more effective optimizations at low error rates. On the `ferret` benchmark, loop perforation slightly outperforms POWERGAUGE at low error rates, but POWERGAUGE finds greater overall power savings.

In addition to more common loop perforation techniques, such as skipping every nth iteration of a loop or terminating a loop early, POWERGAUGE's general mutations can lead to other types of loop modifications. For example, when a compiler performs loop unrolling, several copies of the loop body are created to speed up execution time. Each of these unrolled loop bodies can be independently modified, allowing POWERGAUGE to insert code that only executes in a specific loop iteration .

7.3.2 Precision Scaling

Precision scaling techniques trade off accuracy for better performance [34], [35]. We compared to one particular form of precision scaling by identifying benchmarks that perform floating point calculation and changing the precision of the floating point numbers. `bodytrack`, `ferret`, `swaptions`,

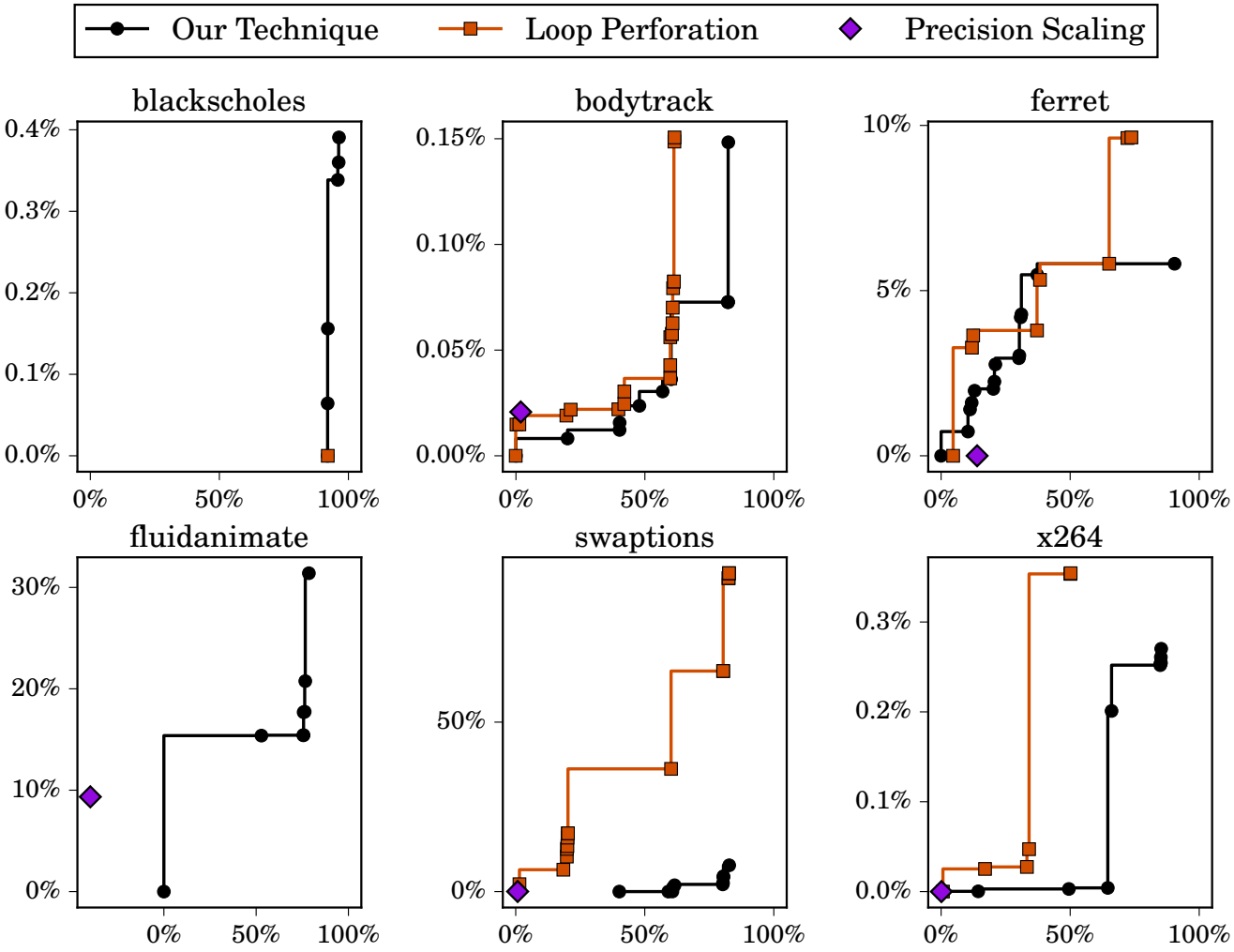

Fig. 8: Comparison of our technique with with loop perforation and precision scaling. The X axes indicate the percent energy reduction achieved; the Y axes indicate the per-benchmark error metric. The baseline original program has 0 error and 0% energy savings (lower left corner). Points in the lower-right indicate optimizations and points in the upper-right corner indicate tradeoffs between energy and output fidelity. Loop perforation and precision scaling results were evaluated using our energy meter and error metrics.

`vips`, and `x264` all use double-precision floating point by default, while `fluidanimate` uses single-precision floating point by default. For `bodytrack`, `ferret`, `swaptions`, and `x264`, all double-precision floating point calculations and variables were changed to single-precision, while for `fluidanimate` all single-precision floating point calculations were changed to double-precision. This simple approach to precision scaling represents a common technique often considered by developers; for example, the `blackscholes`, `fluidanimate`, and `swaptions` benchmarks already use explicit compile-time typedefs to control precision scaling in this manner.

The results are shown in Figure 8 and indicated by a purple diamond. Uniformly changing the precision produced interesting tradeoffs for all benchmarks except `vips`, for which it resulted in error far in excess of any other discovered optimization. In the cases of `swaptions` and `x264`, precision scaling had little noticeable effect, while for `bodytrack` it yielded an energy savings of approximately 2%, but at an error greater than optimizations found by PowerGauge that lead to an power savings of over 40%. In `fluidanimate`, increasing the precision lead to a higher power usage. Precision scaling was most effective with the `ferret` benchmark, where it lead to a 14% power savings at zero error (suggesting that the original developers should perhaps have used single-precision floating point). We note, however, that PowerGauge was able to find much greater power savings when some error is allowed. For example, PowerGauge finds a 26% power savings at 3% error in `ferret` as compared to the 14% achieved by precision scaling alone.

The results shown in Table 3 and Figure 8 lead us to conclude that PowerGauge compares favorably with loop perforation and precision scaling.

8 Discussion

In this section we first discuss the benchmarks where we were unable to report optimizations from PowerGAUGE, even after allowing for error and applying search space reductions and profiling. Next, we discuss the nature of the optimizations found by PowerGAUGE. We then analyze the similarities and differences between optimizing for energy consumption and optimizing for runtime. Finally, we address threats to validity.

8.1 Applicability to Benchmarks

Although PowerGAUGE successfully discovers optimizations for 10 out of 12 of our benchmark programs, it did not find optimizations for the "planet" input to `blender` or `fluidanimate`. For many of the other benchmarks, PowerGAUGE was unable to find optimizations without allowing for any subjective change in the output quality, but it was able to find energy reductions for most of our benchmarks when some error is allowed. We look deeper into the reasons for PowerGAUGE's performance on the two remaining benchmarks in this subsection.

`fluidanimate`: Even though PowerGAUGE did not find any energy reductions at human-acceptable levels of error for the `fluidanimate` benchmark, it was still able to find Pareto-optimal tradeoffs between error and energy reductions with respect to our error metric (see the `fluidanimate` plot in Figure 8). Since we were unable to judge the human-acceptability of error in `fluidanimate` due to its output format, we considered any error at all to be unacceptable. However, the existence of a Pareto frontier suggests that with more domain knowledge we might be able to make subjective judgments to find acceptable tradeoffs and save energy. Note that the user's ability to make judgments about the output quality of a program is an assumption of the technique presented in this article, and `fluidanimate` was included in our evaluation to allow for comparison to GOA, which was also unable to find optimizations for this benchmark [19].

`blender`: In contrast, the searches for `blender` optimizations using the "planet" input did not expose any Pareto-optimal tradeoffs at all. This is surprising: it is easy to contrive Pareto-optimal but unacceptable programs (e.g., a program that outputs an empty file and immediately terminates would use a very small amount of energy at a very high error), but none were observed after the searches. When examining the logs of the search, we discovered that PowerGAUGE found a program modification that produced the correct output and a large energy savings (approximately 14%), however the behavior of this program is non-deterministic (i.e., it occasionally crashes). Because of the large fitness in both error and energy consumption, this modification prevented the creation of other, less optimal programs. During the minimization stage of the search, the edits were rejected due to the nondeterministic behavior that was discovered when the program was executed several times. One possible solution for this problem would be the reevaluation of programs generated during the search to eliminate programs with nondeterministic behavior, however this would greatly increase the search time. Since we only observe this behavior on one input for one of our benchmark programs we leave the solution of this problem for future work.

8.2 Nature of Optimizations

We have explored optimizations created by PowerGAUGE for many of our benchmarks in an effort to identify important properties of input programs or common modifications that might be applied by compilers or programmers in lieu of a stochastic search, but there was no single modification that seemed to be most common. The observed optimizations ranged from modifying arguments in function calls to changing elements of a convolution matrix to more traditional techniques such as loop perforation. We see this wide variety of found optimizations as an advantage of the technique presented in this article.

8.3 Energy vs. Runtime

A common observation of work in energy optimization is that many reductions in energy consumption are the result of reducing execution time rather than performing energy-specific optimizations. We observe that this is the case: in our experiments we found a 0.993 correlation between runtime and energy reductions of our optimized programs. Despite this strong correlation, we still believe that optimizing with respect to an energy meter is well motivated. Although discovering novel energy-specific optimizations would be interesting, we are primarily focused on minimizing overall energy costs by reducing as much energy consumption as possible.

The high correlation between energy reduction and runtime reduction seems to indicate that we could have optimized with respect to runtime alone and elided the energy meters entirely, but this can introduce problems. First, previous work has found program modifications that increase runtime but reduce the overall energy consumption (see the GOA results for the `ferret` benchmark [19]), and it is also known that instruction reordering can have a positive effect on energy consumption without changing runtime [39]. Runtime cannot account for these optimizations and can even lead the search away from them, thus we must use measured energy if we wish to allow their discovery. Second, even when using runtime as a naïve model of energy consumption during the search, we must verify the energy consumption of the final program. Although it is difficult to imagine energy-specific optimizations of programs written by humans and optimized by compilers, it is much easier to conceive of ways that a stochastic search can create two programs that have the same runtime but consume different amounts of energy. For example, in preliminary experiments, we observed PowerGAUGE program modifications that enabled or disabled multithreading. A further modification could create a program that runs a computation on a single core while launching superfluous threads on other cores, but terminate before the primary computation completes.

These problems do not necessarily completely preclude the use of runtime as a fast, inexpensive energy model. It could be possible to use both an energy meter and runtime in a hybrid approach, where the search is performed with

respect to runtime, but the population of programs is occasionally sampled and verified against a physical meter. This would allow for a search to be parallelized on many unmonitored systems that submit sampled programs to a test server connected to an energy meter for verification.

8.4 Threats to Validity

Although our experiments show that our technique is able to find energy optimizations in indicative programs at varying levels of output fidelity, our results may not generalize. We identify the following threats to validity:

8.4.1 Relaxed Semantics and Error

Since our technique often performs transformations that do not preserve semantics, it is possible that an optimization can change program behavior in an undesirable way. We mitigate this problem by incorporating an error metric into our search and direct POWERGAUGE to minimize the induced error. If desired, a developer can specify particular program properties to preserve by incorporating them into the error metric (e.g., assigning an infinite error to any violation of a key invariant or assertion).

8.4.2 Noise in Energy Measurement

Although we attempt to improve upon our previous technique by measuring instead of modeling energy, we find that energy measurements tend to be noisy. Because of this, it is possible that our algorithm interprets fluctuations in energy readings as optimizations. We mitigate noise in energy measurement by reevaluating the energy consumption of both the original and optimized programs and using the Mann-Whitney U-test [52] to verify the power savings reported by our tool. All of the final optimizations reported in this article (e.g., points on Pareto frontiers, etc.) are the average of at least 25 measurements at the wall socket.

8.4.3 Representative Benchmarks

The PARSEC suite [11] and the `libav` and `blender` benchmarks were selected to be representative of types of applications typically deployed in data centers. However, success on these benchmarks does not imply that POWERGAUGE would be successful at finding energy optimizations on all applications deployed in a data center. Note that POWERGAUGE is input agnostic, and its only requirements for an input program are compiled assembly and a test suite. Unlike techniques such as loop perforation [10], [33] or precision scaling [34], [35], POWERGAUGE does not require specific code structure such as loops or floating point numbers, nor does it require guidance from developers to identify possible optimization locations. This, combined with observations that the optimizations found by POWERGAUGE do not follow an identifiable pattern, increases our confidence that it can be applied to a wide variety of software.

There are some applications where POWERGAUGE is unlikely to be an appropriate technique. For example, we make the assumption that the state of the system as a whole is identical at the start of every test run, but this is not always practical. In a database system where hard disks consume a large amount of energy, the state of on-disk cache is likely to change after an access during a test, which could affect the energy consumption of a subsequent access. Addressing this problem could require modifying the testing framework or the POWERGAUGE algorithm.

8.4.4 Architecture

Another possible threat to validity is that our experimental setup only includes Intel Xeon CPUs. It is possible that optimizations targeted for one architecture will not generalize. We note, however, that this is a concern for optimizations in general and is not specific to our technique. In addition, Schulte et al. evaluated a similar technique to POWERGAUGE on both AMD and Intel CPUs, and were able to find optimizations for benchmarks run on both [19]. We also consider a data center-style use case in which the hardware and an indicative workload are known in advance [12], [63].

9 Conclusion

Data center scale computation accounts for a significant fraction of energy consumption and has a growing economic impact on business. Although there are advances in hardware and compilers that partially address this problem, software perspectives on energy reduction are relatively unexplored. Advances in search-based software engineering have shown that automated program optimization techniques can successfully be applied to the domain of energy reduction, but current techniques do not scale and can only improve modeled (as opposed to measured) energy.

By leveraging insights from search-based software engineering and profile-guided optimization, we present POWERGAUGE, a software-level energy-reduction that scales to much larger applications than were previously possible. The search is guided by a measurement device that combines of off-the-shelf components and specialized firmware to create an inexpensive device capable of monitoring the energy consumption of server systems. We also present large search space reductions and use precise instruction-level profiling to direct the search in order to find optimizations in programs with over 20 million lines of assembly. Our technique is able to find optimizations that reduce the energy consumption in 10 of 12 benchmarks by 14% on average overall while still passing a provided test suite, and by 41% with human-acceptable error.

Acknowledgments

We would like to thank Kevin Angstadt for his help setting up the microcontrollers, debugging their software, and for many fruitful discussions. We are also grateful to Eric Schulte for his contributions to the early stages of this work and Shane Clark at Raytheon BBN Technologies for his helpful suggestions on measuring real-world energy.

References

[1] M. Glinz, "On non-functional requirements," in *International Requirements Engineering Conference*, ser. RE '07, 2007, pp. 21–26.
[2] J. Koomey, *Growth in data center electricity use 2005 to 2010*. Oakland, CA: Analytics Press, 2011.
[3] J. Whitney and P. Delforge, "Data center efficiency assessment," *Issue Paper, Aug*, 2014.

[4] U. Hoelzle and L. A. Barroso, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[5] K. J. Nowka, G. D. Carpenter, E. W. MacDonald, H. C. Ngo, B. C. Brock, K. I. Ishii, T. Y. Nguyen, and J. L. Burns, "A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1441–1447, 2002.

[6] J. Mars, L. Tang, K. Skadron, M. Soffa, and R. Hundt, "Increasing utilization in modern warehouse-scale computers using bubble-up," *IEEE Micro*, vol. 32, no. 3, pp. 88–99, May 2012.

[7] M.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and minimization techniques for embedded DSP software," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 5, no. 1, pp. 123–135, 1997.

[8] S. Reda and A. N. Nowroz, "Power modeling and characterization of computing devices: A survey," *Foundations and Trends in Electronic Design Automation*, vol. 6, no. 2, pp. 121–216, 2012.

[9] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 503–514.

[10] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 124–134.

[11] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[12] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *International Symposium on Computer Architecture*, ser. ISCA '15, 2015, pp. 158–169.

[13] M. Orlov and M. Sipper, "Flight of the FINCH through the Java wilderness," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 166–192, 2011.

[14] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *International Conference on Automated Software Engineering*, ser. ASE '13, 2013, pp. 356–366.

[15] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[16] B. L. Miller, B. L. Miller, D. E. Goldberg, and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex Systems*, vol. 9, pp. 193–212, 1995.

[17] T. Jones and S. Forrest, "Fitness distance correlation as a measure of problem difficulty for genetic algorithms," in *International Conference on Genetic Algorithms*, ser. ICGA '95, 1995, pp. 184–192.

[18] S. O. Haraldsson and J. R. Woodward, "Genetic improvement of energy usage is only as reliable as the measurements are accurate," in *Genetic and Evolutionary Computation Conference*, ser. GECCO '15, 2015, pp. 821–822.

[19] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014, pp. 639–652.

[20] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Genetic and Evolutionary Computation Conference*, ser. GECCO '15, 2015, pp. 1327–1334.

[21] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of GUIs in Android apps: A multi-objective approach," in *Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 143–154.

[22] H. Massalin, "Superoptimizer: A look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, pp. 122–126, 1987.

[23] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, pp. 305–316.

[24] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *Symposium on Compiler Construction*, ser. SCC '82, 1982, pp. 120–126.

[25] R. Gupta, E. Mehofer, and Y. Zhang, *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002, ch. Profile Guided Compiler Optimizations, pp. 143–174.

[26] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *ACM SIGPLAN Notices*, vol. 25. ACM, 1990, pp. 16–27.

[27] H. Jacobson, P. Bose, Z. Hu, A. Buyuktosunoglu, V. Zyuban, R. Eickemeyer, L. Eisen, J. Griswell, D. Logan, B. Sinharoy, and J. Tendler, "Stretching the limits of clock-gating efficiency in server-class processors," in *Symposium on High-Performance Computer Architecture*, ser. HPCA '05, 2005, pp. 238–242.

[28] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *European Test Symposium*, ser. ETS '13, 2013, pp. 1–6.

[29] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Design Automation Conference*, ser. DAC '15, 2015, pp. 120:1–120:6.

[30] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *International Conference on Supercomputing*, ser. ICS '06, 2006, pp. 324–334.

[31] ——, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '07, 2007, pp. 369–386.

[32] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application partitioning," in *European Conference on Object-Oriented Programming*, ser. ECOOP '02, 2002, pp. 178–204.

[33] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," MIT, Technical Report, 2009.

[34] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, "ApproxMA: Approximate memory access for dynamic precision scaling," in *Great Lakes Symposium on VLSI*, ser. GLSVLSI '15, 2015, pp. 337–342.

[35] O. Sarbishei and K. Radecka, "Analysis of precision for scaling the intermediate variables in fixed-point arithmetic circuits," in *International Conference on Computer-Aided Design*, ser. ICCAD '10, 2010, pp. 739–745.

[36] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, Jan. 2013.

[37] Z. Yang, J. Han, and F. Lombardi, "Transmission gate-based approximate adders for inexact computing," in *International Symposium on Nanoscale Architectures*, ser. NANOARCH '15, 2015, pp. 145–150.

[38] S. Lu, "Speeding up processing with approximation circuits," *IEEE Computer*, vol. 37, no. 3, pp. 67–73, 2004.

[39] C. Lee, J. K. Lee, T. Hwang, and S. Tsai, "Compiler optimization on instruction scheduling for low power," in *International Symposium on System Synthesis*, ser. ISSS '00, 2000, pp. 55–60.

[40] D. McIntire, T. Stathopoulos, S. Reddy, T. Schmidt, and W. J. Kaiser, "Energy-efficient sensing with the low power, energy aware processing (LEAP) architecture," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 2, pp. 27:1–27:36, Jul. 2012.

[41] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for Android applications," in *International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 78–89.

[42] T. Bessa, P. Quintão, M. Frank, and F. Magno Quintão Pereira, "JetsonLeap: A framework to measure energy-aware code optimizations in embedded and heterogeneous systems," in *Brazilian Symposium on Programming Languages*, ser. SBLP '16, 2016, pp. 16–30.

[43] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '14, 2014, pp. 345–360.

[44] M. A. Ferreira, E. Hoekstra, B. Merkus, B. Visser, and J. Visser, "SEFLab: A lab for measuring software energy footprints," in *International Workshop on Green and Sustainable Software*, ser. GREENS '13, 2013, pp. 30–37.

[45] F. Ulaby and M. Maharbiz, *Circuits*, 2nd ed. National Technology & Science Press, 2013.

[46] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.
[47] W. B. Langdon, J. Petke, and B. R. Bruce, "Optimising quantisation noise in energy measurement," in *Parallel Problem Solving from Nature*, ser. PPSN XIV, 2016, pp. 249–259.
[48] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Genetic and Evolutionary Computation Conference*, ser. GECCO '11, 2011, pp. 1427–1434.
[49] C. Le Goues, S. Forrest, and W. Weimer, "Representations and operators for improving evolutionary software repair," in *Genetic and Evoluationary Computation Conference*, ser. GECCO '12, 2012, pp. 959–966.
[50] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr 2002.
[51] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '99, 1999, pp. 253–267.
[52] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.
[53] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1301–1321, 1991.
[54] E. Schulte, J. DiLorenzo, S. Forrest, and W. Weimer, "Automated repair of binary and assembly programs for cooperating embedded devices," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013, pp. 317–328.
[55] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005, pp. 190–200.
[56] J. M. Kasson and W. Plouffe, "An analysis of selected computer interchange color spaces," *ACM Transactions on Graphics*, vol. 11, no. 4, pp. 373–405, Oct. 1992.
[57] R. Villemin, C. Hery, S. Konishi, T. Tejima, R. Villemin, and D. G. Yu, "Art and technology at Pixar, from Toy Story to today," in *SIGGRAPH Asia 2015 Courses*, ser. SA '15, 2015, pp. 5:1–5:89.
[58] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International Journal of Computer Vision*, vol. 40, no. 2, pp. 99–121, 2000.
[59] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.
[60] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 107–118.
[61] C. Le Goues, N. Holtschulte, E. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *ACM Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.
[62] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. Da Fonseca, "Performance assessment of multiobjective optimizers: An analysis and review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117–132, 2003.
[63] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing Google's warehouse scale computers: The NUMA experience," in *International Symposium on High Performance Computer Architecture*, ser. HPCA '13, 2013, pp. 188–197.

Jonathan Dorn received the BS degree in computer science from the University of Texas at Austin and the MS and PhD degrees from the University of Virginia. He is currently a senior scientist at GrammaTech, Inc. His research interests include the optimization of non-functional properties in programs, naturalness of software, and human factors in software engineering.

Jeremy Lacomis received the BA degree in computer science from the University of Virginia. He is currently pursuing a PhD in computer science at Carnegie Mellon University. His main research interests include optimization of non-functional program properties and automated program repair.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently a professor at the University of Michigan. His main research interests include static and dynamic analyses to improve software quality and fix defects.

Stephanie Forrest received the BA degree from St. Johns College and the MS and PhD degrees from the University of Michigan. She is currently at Arizona State University, where she directs the Center for Biocomputation, Security and Society and is Professor of Computing, Informatics, and Decision Systems Engineering. Her research interests include biological modeling, evolutionary computation, software engineering and computer security. She is a fellow of the IEEE.