Automatically Reducing Energy Consumption of Software

Jeremy Lacomis,[*] Jonathan Dorn,[†]
Westley Weimer,[‡] and Stephanie Forrest[§¶]

*To whom correspondence should be addressed.

Introduction

As computation continues to migrate from personal computers to large-scale data centers the energy required to run computers has become a significant economic and environmental concern. For example, between 2005 and 2010 data center electricity consumption grew by 24%, and by 2014 data centers accounted for 1.8% of U.S. energy consumption (Shehabi et al. 2016). Although this large energy footprint has led to some mitigation efforts, energy consumption in data centers continues to rise. Current estimates project U.S. energy use to increase a further 4% from

[*]Carnegie Mellon University, Institute for Software Research, Wean Hall 5216, 5000 Forbes Avenue, Pittsburgh, PA 15213; jlacomis@cmu.edu

[†]GrammaTech, Inc., 531 Esty Street, Ithaca, NY 14850; jdorn@grammatech.com

[‡]University of Michigan, 4636 Beyster Building, 2260 Haward Street, Ann Arbor, MI 48109; weimerw@umich.edu

[§]Biodesign Institute, Arizona State University, P.O Box 877801, Tempe AZ, 85287; stephanie.forrest@asu.edu

[¶]Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM 87501

2014 to 2020. In 2016, in anticipation of the possible environmental impact of their growing energy demand, Google announced a $2.5 billion commitment to the purchase of energy from renewable sources.[1]

Computer hardware efficiency directly effects data center energy consumption, and this effect is multiplied by the support systems required for deployment. Mechanical and electrical systems, such as lighting, cooling, air circulation and uninterruptible power supplies, can quadruple the power required by the computational hardware itself (Hoelzle and Barroso 2009).

The software running in the center can further multiply energy consumption. For example, data centers must be provisioned with sufficient hardware to run the desired algorithms in a timely fashion. Algorithmic inefficiencies in software implementations can increase run times, leading to greater emphasis on parallelism to compensate. Contention for resources such as networks, disks, memory or caches leads to overprovisioning hardware (Mars et al. 2012). At sufficient scales, hardware reliability concerns require implementation of redundant resources and computations—for example, Microsoft implements redundancy for all customer data in Azure storage accounts to meet the uptime guaranteed by their Service-Level Agreements.[2] Because the load on the support systems scales with computational load, improving computational efficiency could significantly reduce overall energy costs of data centers.

The problem of computational energy consumption has typically been addressed by optimizing hardware (Douglis, Krishnan, and Bershad 1995; Delaluz et al. 2001; Nowka et al. 2002), compilers (Lee et al. 1997; Hsu and Kremer 2003; Reda and Nowroz 2012), or cluster scheduling (Mars et al. 2012), leaving open the question of how to write energy-efficient applications. These optimizations are largely independent, and energy reductions from different perspectives are composable to achieve greater

1. https://environment.google
2. https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy

savings. Reducing software energy use is challenging because it is often unclear how implementation decisions impact energy consumption, making it difficult for developers to write programs that minimize energy use (Manotas, Pollock, and Clause 2014).

In this chapter, we focus on emerging techniques for *automatically* reducing energy consumption in existing computer programs. First, we give an overview of specialized approaches that leverage knowledge about specific properties of some software. Next, we consider new, more general approaches that use insights from evolutionary computation (Schulte et al. 2014; Dorn et al. 2017). By modifying software and measuring the difference in energy consumption, these approaches are able to automate the "change, observe, iterate" loop that a developer might use when she finds it difficult or impossible to reason about how changes to the software will impact performance. These generic techniques are applicable to many types of software and sometimes reveal new generalizable methods for optimizing specific types of software.

Reducing Software Energy Consumption

Reducing the energy consumption of a program requires that its execution be changed in some way. There are three main approaches used today: semantics-preserving techniques, approximate computing techniques, and techniques based on stochastic search.

Semantics-Preserving Techniques

These techniques require that all transformations to the program preserve input/output behavior that is identical to the original program. For example, standard compiler optimizations for reducing run-time guarantee that program semantics are not changed. These techniques generate programs that are correct by construction.

Instruction Scheduling Techniques

Modern CPU internals are primarily composed of transistors used as switches. CPUs combine these transistors into small units that perform simple operations (e.g., addition, division, logical operations), select memory to operate on, or do basic control and input/output. When a computer is running, the CPU is given *instructions* that direct the CPU to combine these units and perform the desired computation.

The transistors inside the CPU account for a majority of its energy use. There are two types of energy consumption in transistors: *static* and *dynamic* (Niu and Quan 2004). Static energy consumption is the energy required to hold a transistor at a steady-state, while dynamic energy consumption is the energy dissipated as heat when a transistor is switched from one state to another. Although static energy consumption in modern processors is important, the majority of energy consumption is dynamic (Sarwar 1997). To minimize dynamic energy consumption, researchers use *instruction scheduling* (Lee et al. 1997).

Instruction scheduling techniques reduce the number of times that transistors change state, thus reducing dynamic energy consumption. This is accomplished by optimizing the ordering of independent instructions—instructions that produce the same computation if executed in a different order—to minimize the number of times that transistors change state. For example, if a transistor T is used in the independent instructions I_1, I_2, and I_3 where the values of T are T_{off}, T_{on}, and T_{off} respectively, the instruction ordering $I_1 \rightarrow I_2 \rightarrow I_3$ requires T to switch twice ($T_{off} \rightarrow T_{on} \rightarrow T_{off}$), while the instruction ordering $I_1 \rightarrow I_3 \rightarrow I_2$ only requires T to switch once ($T_{off} \rightarrow T_{off} \rightarrow T_{on}$). This ordering consumes less dynamic energy while executing the same instructions and performing the same computation.

A limitation of instruction scheduling is that any reordered instructions must be independent of one another, which is often not the case. It also requires an available energy model for each instruction (as different transistors may have different dynamic power consumption). As a re-

sult, its use has typically been limited to applications where minimizing energy usage is paramount, such as ultra-low power mobile devices.

Superoptimization

Superoptimization (Massalin 1987; Schkufza, Sharma, and Aiken 2013) is similar to instruction scheduling, since both reorder low-level instructions. Superoptimization, however, is more general: instead of considering specific properties (e.g., reduced switching), it reorders instructions in any way that preserves semantics, and then performance is measured empirically. Assuming an effective mechanism to verify functionality, superoptimization can identify the best sequence of instructions by exhaustive enumeration. However, his strategy works only for very short sequences of instructions: modern instruction sets are large and exhaustive search is infeasible for more than a few instructions. Stochastic search enables these techniques to scale to longer sequences that compute more complex functions, but cannot guarantee that the best sequence has been found. Stochastic superoptimization approaches remain constrained by the verification mechanism, as complex functions are generally difficult or impossible to verify formally (Rice 1953).

Approximate Computing

Approximate computing relaxes the requirement to preserve exact semantics, allowing a tradeoff between computational accuracy and runtime or energy consumption (Palem 2014). This tradeoff is analogous to how lossy compression formats such as MP3 or MPEG4 achieve smaller file sizes than lossless formats in exchange for reduced output fidelity. Approximate computing techniques exist for both hardware (Lu 2004; Gupta et al. 2013; Palem and Lingamneni 2013; Yang, Han, and Lombardi 2015) and software (Han and Orshansky 2013; Venkataramani et al. 2015). We discuss three approximate computing approaches that are suitable for software implementation—*precision scaling*, *task skipping*, and *loop perforation*.

Precision Scaling

Precision scaling reduces computational cost by modifying the *precision* of floating-point variables used to represent real numbers in arithmetic (Sarbishei and Radecka 2010; Tian et al. 2015). Precision refers to the number of bits that are used to represent a number in memory: higher-precision representations use more bits and are more accurate. However, more bits implies more space in memory, which can affect energy consumption. In certain cases it is possible to adjust the precision of variables to reduce energy. For example, reducing precision in hardware can reduce the size of the circuit required for computation, thereby reducing its power consumption; in software, changing the precision of a variable can change memory layouts, reducing the time to look up values[3] and decreasing energy usage. However, these effects are difficult to predict and scaling may have no effect at all.

Task Skipping

Task skipping (Rinard 2006, 2007) reduces energy consumption or runtime by halting or skipping the execution of tasks in a program when the results are unneeded. The strategy uses a model to characterize the trade-off between accuracy and performance to decide if an execution should be skipped. Before task skipping can be applied, the program must be manually decomposed into tasks by a developer. This requires programmers with domain-specific knowledge to perform a nontrivial amount of manual work, and sometimes this type of decomposition is not possible (Tilevich and Smaragdakis 2002). This greatly limits the applicability of the technique.

3. Frequently-used values are often stored in *cache*, a special type of extremely fast memory located inside the CPU itself. Although cache memory is fast, it is expensive and its size is limited. Larger data might not fit into cache, and can also introduce problems with *alignment*, a topic beyond the scope of this chapter.

Loop Perforation

Similar to task skipping, loop perforation reduces runtime and energy consumption by skipping unnecessary computation (Hoffmann et al. 2009; Sidiroglou-Douskos et al. 2011). In loop perforation, individual iterations of loops are skipped during a calculation. This approach is effective for algorithms that iteratively improve the accuracy of a computation, such as spigot algorithms that iteratively compute the digits of π (Rabinowitz and Wagon 1995) or iterative approximations of integrals (Kammerer and Nashed 1972). Skipping iterations of these loops produces an approximate answer with less energy than a fully-precise answer.

Loop perforation has two main advantages over task skipping: (1) loops do not have to be manually specified by domain experts because they can be identified easily and automatically; (2) loops are ubiquitous in software, so there are many more applicable programs for loop perforation. However, not all loops are improved with perforation, and in certain cases loop perforation actually *decreases* performance. For example, a loop might be used to filter a list before it is passed to an expensive processing step (Hoffmann et al. 2009), so skipping iterations of the filter loop can actually increase energy consumption.

Energy Reduction using Genetic Improvement

Although these earlier approaches to software energy reduction can be effective, they either use very limited transformations (e.g., instruction scheduling and superoptimization) or require programs to have specific properties (e.g., task skipping). Generally, it is difficult to predict the impact of a given transformation on the behavior and energy consumption of a program. This difficulty motivates the use of stochastic optimization methods, such as simulated annealing (Kirkpatrick, Gelatt, and Vecchi 1983), ant colony optimization (Dorigo and Birattari 2011), and genetic algorithms (GAs) (Holland 1992).

In the following we focus on GAs and related methods, which are known collectively as *evolutionary computation* (EC), because they have

been applied successfully to software modification. A GA takes as input a representation and a fitness funtion. The representation specifies a set of properties that can be assembled into a *chromosome* to form a candidate solution (called an *individual*). The fitness function computes the goodness, or *fitness*, of each individual, returning a numerical rating. Execution of a GA begins with an initial *population* of individuals, either generated randomly or provided by some other means. Each individual's fitness is evaluated and used to stochastically *select* individuals, which are then mutated and recombined with other individuals to form the next generation. Although the details of these transformations vary according to the specific implementation, there are two main techniques for creating new individuals from the previous generation: *mutation* and *crossover*. Mutation randomly modifies an individual (e.g., through bit flips), while crossover recombines the chromosomes from two or more parents, analogous to crossing over in biology. These processes of fitness evaluation, selection and variation are iterated for many generations, *evolving* an improved solution to a problem.

An important subfield of EC is genetic programming (GP) (Koza 1992), where programs are evolved to approximate the input/output behavior of a hidden function (a form of function approximation). GP methods have been applied to several problems in software engineering, such as repairing bugs (Le Goues et al. 2012), obfuscating code (Petke 2016), and implementing new functionality (Harman, Jia, and Langdon 2014). In these examples, GP is used to improve extant software, and the term *genetic improvement* refers to this class of applications (Langdon 2015).

Energy optimization is, of course, another form of software improvement, and there are several recent efforts along these lines (Schulte et al. 2014; Bruce, Petke, and Harman 2015; Linares-Vásquez et al. 2015; Bruce et al. 2018). In these applications, fitness corresponds to energy reduction, which can be directly measured or modeled. An extension to this work uses multi-objective optimization to trade off energy and accuracy, a form of approximate computing. For example, our work

on PowerGAUGE (Dorn et al. 2017) relaxes the requirement of strict preservation of semantics to achieve greater energy reductions. Note that the optimization of software along two dimensions does not return one "best" implementation, but a set of *Pareto optimal* (Zitzler et al. 2003) programs. A program that is Pareto optimal is one for which no other observed program has lower energy consumption without a corresponding increase in error, nor does another program have a lower error without an increase in energy consumption. The fitnesses of these Pareto optimal programs lie on a *Pareto frontier*, which can be visualized on an error vs. energy reduction graph, as shown in Figure 1.

The PowerGAUGE *Energy Reduction Algorithm*

The PowerGAUGE energy reduction algorithm uses a multi-objective GA to optimize multi-dimensional fitness functions. We use the NSGA-II implementation (Deb et al. 2002) to optimize software with respect to both energy consumption and error. PowerGAUGE takes as input a program to be optimized and an n-dimensional fitness function. In our case $n = 2$, and the fitness function computes the energy consumption of the program and error in the program output. Note that PowerGAUGE is not restricted to optimizing energy consumption and output error; it could optimize with respect to any properties that can be represented by a fitness function, such as program file size or memory usage.

Recall that one of the fundamental concepts in EC is the choice of how to represent individuals in the population. Although we start with programs written in the C programming language, we represent them in *assembly language*, a low-level representation of the instructions to be executed by the CPU, which is generated from high-level code by a *compiler*.[4]

4. Assembly code can also be hand-written, but it is typically verbose and difficult to understand, so this practice is unusual.

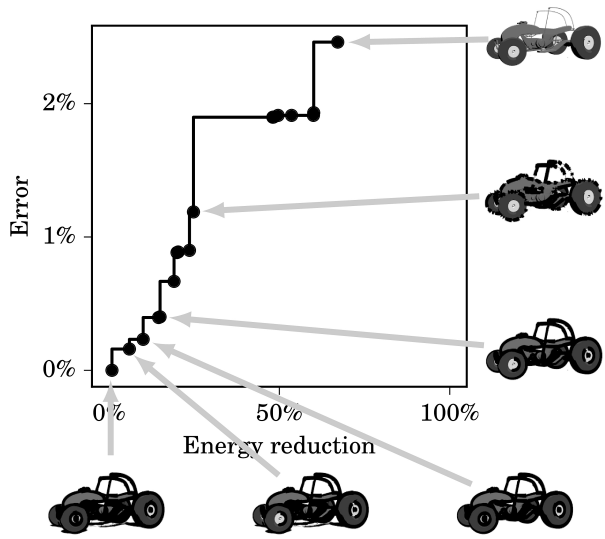

Figure 1. *Pareto frontier for the* B1ender *benchmark. The X-axis indicates the percentage energy reduction (greater is better), and the Y-axis indicates error (lower is better). Each point corresponds to a program generated by* PowerGAUGE. *Example output images are provided for some programs. The image in the lower-left has no error and corresponds to a 1% energy savings. The image in the lower-right has a small amount of error, but corresponds to a 10% energy savings.*
Images generated using a B1ender *demo file by Rylan Wrigh / CC BY, https://download.blender.org/demo/test/AtvBuggy.zip*

There are several advantages to representing programs at this level:

- Compilers optimize the performance of software by applying semantics-preserving transformations to the code. By manipulating the output of the compiler, we can incorporate these optimizations.
- Optimization passes consume the majority of time spent during compilation. Thus, the transformation from high-level code to low-level assembly takes much more time than the transformation from low-level assembly into a binary format suitable for execution on a CPU. By modifying the assembly directly we avoid repeating these costs for each fitness evaluation.
- High-level code has many syntactic restrictions that must be respected when making modifications. Assembly is much less strict with many more sequence of instructions considered to be legal. This flexibility enhances expressiveness and the creation of programs that could not be generated from source code.
- When targeting a particular machine, compilers for different high-level languages (e.g., C, Pascal, Fortran) all use identical assembly language. This feature implies that PowerGAUGE can operate on programs generated from multiple source-level languages.

One challenge of applying EC to programs, especially assembly programs, is the large number of instructions and high redundancy of code across individuals of a population. To cope with this, we adopt the *patch representation* developed in earlier work on automated program repair (Le Goues et al. 2012). Instead of representing an individual as a complete program, each individual is represented as a list of modifications (edits) to the original program. In our case, we treat the original assembly code as a sequence of instructions, and then represent each individual as a sequence of modifications in which instructions are *deleted*, *swapped* (i.e., exchanging the positions of two instructions), or *copied* (i.e., duplicating an instruction and inserting it at a random location). Although other

types of code transformations are possible (e.g., synthesizing new code (Mechtaev, Yi, and Roychoudhury 2016) or template-based edits (Kim et al. 2013)), we found that these particular mutations are both effective and computationally inexpensive.

The second input to a genetic algorithm is its fitness function. In PowerGAUGE, fitness is two-dimensional: one dimension represents the level of error in the program's output, and the other corresponds to the amount of energy consumed while running the program. Creating a fitness function for energy consumption is conceptually straightforward—simply run the program and measure or model the energy consumed (in practice this is surprisingly difficult for some applications). The fitness function for error is more domain-specific, however. For the purposes of this chapter we consider a simple example: the image produced by `Blender`, an open-source 3D rendering application. The fitness function measures error as the "distance" from the image output by the program to a reference image. There are natural distance metrics for images, but for other applications it may not be feasible to quantify the output error, e.g., for an email client. This may seem like a limitation, but in these cases PowerGAUGE can still optimize for energy consumption alone.

Figure 1 shows the Pareto frontier of different versions of `Blender` generated during an actual PowerGAUGE run. The X-axis indicates the percentage energy reduction (greater is better), while the Y-axis indicates the percentage error (lower is better). The "ideal" program would fall in the lower-right of this graph, generating identical output to the original program while consuming zero energy. To give some intuition about what these errors look like, several of the Pareto programs are annotated with their output. The program in the lower-left generates an image with no error and uses 1% less energy than the original program. As we tolerate more error and move along the Pareto frontier energy savings increase, until we reach the upper-right of the figure. At this point, error is high, but there is a 67% energy savings over the original program. A user of PowerGAUGE could generate these Pareto optimal programs and

select the one that has an acceptable level of error for her application to maximize energy savings.

Open Questions and Future Directions

Although PowerGAUGE and related approches show promise for reducing the energy budgets of software, there are still many open questions and future directions to explore. Here, we discuss two: the open problem of effective energy measurement, and the practical application of PowerGAUGE to security.

Energy Measurement

Any empirically-based search method for energy reduction requires a method for measuring or estimating energy consumption of candidate programs. Modeling is one approach that avoids the expense and communication overhead of additional hardware. However, in our preliminary studies, we found that energy models were inaccurate to the point of interfering with the search, and physical measurements using specialized hardware became an attractive option. This observation may not apply to all energy models or all search methods, but others have made similar observations (Haraldsson and Woodward 2015), which focused our attention on energy measurement methods.

Modern processors often include internal features such as Running Average Power Limit (RAPL) (David et al. 2010), which report consumption measurements, but in our experiments we found these to be unreliable, and the documentation of these features was inadequate. In addition, measuring only internal CPU energy ignores the energy consumption of other system components, which might be affected by a change to software (e.g., memory modules, hard disks, graphics cards). An ideal technique would capture the energy consumption of the entire system. Commercial power monitoring devices exist, but they tend either to be

prohibitively expensive, only applicable to mobile devices, or to have insufficient resolution with respect to time and/or energy. In our case, the simplest solution was to construct our own device to measure energy accurately and cost-effectively to use in our search process.

Although we used open-source hardware and provide the source code for the measurement device, better built-in support for energy monitoring would enable anyone to more easily run a search algorithm without having to construct their own hardware. For example, monitoring circuits with a software interface could be included in server power supplies, or CPU manufacturers could provide more detailed energy consumption measurements.

Side-Channel Attack Mitigation

A common security concern is the leaking of sensitive information through *side-channels*. A side-channel is a source of information related to the effects of a computation on its environment, rather than through weaknesses in the algorithm itself. An example of this is a timing attack in the Unix `login` command. Early versions of `login` only ran the expensive hash function required to check password validity if the provided login name was a known system user. This allowed an attacker to easily check if a username was legal: if the login failed quickly, the username did not exist, while if the login took a long time to fail, the username did exist and the system was checking the validity of the password. Once a legitimate username was discovered, the attacker could move on to trying common passwords and often succeeded in gaining entry. This type of side-channel attack was also a key component of the recent Spectre and Meltdown vulnerabilities found in x86 processors (Kocher et al. 2018; Lipp et al. 2018).

Similarly, differences in energy consumption can leak information about computations. Search-based techniques could be used to mitigate these effects by normalizing the energy consumption of commands. In the case of the `login` program, the side-channel attack was mitigated by

adding a random delay when a login failed, purposefully increasing the runtime to improve security. Although in this chapter our focus is primarily on the reduction of energy consumption, search-based techniques could also potentially be used to *increase* the energy consumption of software or even target a specific level of energy used by a computation.

Conclusion

As computation migrates to large-scale datacenters, concern about the environmental and economic impact of software is growing. Despite interest in limiting the energy footprint of software, few general-purpose techniques have specifically targeted the creation of low-energy programs. This chapter describes a search-based approach to this problem. Using methods from evolutionary computation, combined with sufficiently accurate energy measurements or models, PowerGAUGE automatically modifies programs and identifies those that consume less energy while preserving required functionality. Genetic improvement methods such as PowerGAUGE do not require human guidance or prior knowledge of how a particular software modification will impact program behavior. Although the research described here is still in its early stages, we are optimistic that effective and developer-friendly techniques for improving the energy efficiency of software will ultimately play a role in reducing energy computation.

References

Bruce, Bobby R., Justyna Petke, and Mark Harman. 2015. "Reducing Energy Consumption Using Genetic Improvement." In *Genetic and Evolutionary Computation Conference*, 1327–1334. GECCO '15.

Bruce, Bobby R., Justyna Petke, Mark Harman, and Earl Barr. 2018. "Approximate Oracles and Synergy in Software Energy Search Spaces." *Transactions on Software Engineering* PP (99): 1–1.

David, Howard, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. "RAPL: Memory Power Estimation and Capping." In *International Symposium on Low-Power Electronics and Design*, 189–194. ISLPED '10.

Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II." *Transactions on Evolutionary Computation* 6 (2): 182–197.

Delaluz, V., M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. 2001. "DRAM Energy Management Using Software and Hardware Directed Power Mode Control." In *International Symposium on High-Performance Computer Architecture*, 159–169. HPCA '01.

Dorigo, Marco, and Bauro Birattari. 2011. "Ant Colony Optimization." In *Encyclopedia of Machine Learning*, 36–39. Springer.

Dorn, Jonathan, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. 2017. "Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs." *Transactions on Software Engineering* PP (99): 1–1.

Douglis, Fred, P. Krishnan, and Brian Bershad. 1995. "Adaptive Disk Spin-down Policies for Mobile Computers." In *Symposium on Mobile and Location-Independent Computing*, 121–137. MLICS '95.

Gupta, Vaibhav, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. 2013. "Low-Power Digital Signal Processing Using Approximate Adders." *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, no. 1 (January): 124–137.

Han, Jie, and Michael Orshansky. 2013. "Approximate Computing: An Emerging Paradigm for Energy-Efficient Design." In *European Test Symposium*, 1–6. ETS '13.

Haraldsson, Saemundur O., and John R. Woodward. 2015. "Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate." In *Genetic and Evolutionary Computation Conference*, 821–822. GECCO '15.

Harman, Mark, Yue Jia, and William B. Langdon. 2014. "Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System." In *International Symposium on Search-Based Software Engineering*, 247–252. SSBSE '14.

Hoelzle, Urs, and Luiz Andre Barroso. 2009. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines.* 1st. Morgan / Claypool Publishers.

Hoffmann, Henry, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. 2009. *Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures.* Technical Report MIT-CSAIL-TR-2009-042. MIT.

Holland, John Henry. 1992. *Adaptation in Natural and Artificial Systems.* Second edition (First edition, 1975). Cambridge, MA: MIT Press.

Hsu, Chung-Hsing, and Ulrich Kremer. 2003. "The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction." In *Programming Language Design and Implementation*, 38–48. PLDI '03.

Kammerer, W. J., and M. Z. Nashed. 1972. "Iterative Methods for Best Approximate Solutions of Linear Integral Equations of the First and Second Kinds." *Journal of Mathematical Analysis and Applications* 40 (3): 547–573.

Kim, Dongsun, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. "Automatic Patch Generation Learned from Human-written Patches." In *International Conference on Software Engineering*, 802–811. ICSE '13.

Kirkpatrick, Scott, C. Daniel Gelatt, and Mario P. Vecchi. 1983. "Optimization by Simulated Annealing." *Science* 220 (4598): 671–680.

Kocher, Paul, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. "Spectre Attacks: Exploiting Speculative Execution." *ArXiv e-prints* (January). eprint: 1801.01203.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Langdon, William B. 2015. "Handbook of Genetic Programming Applications." Chap. Genetically Improved Software, edited by Amir H. Gandomi, Amir H. Alavi, and Conor Ryan. Springer.

Le Goues, Claire, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each." In *International Conference on Software Engineering*, 3–13. ICSE '12.

Lee, Mike Tien-Chien, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. 1997. "Power Analysis and Minimization Techniques for Embedded DSP Software." *Transactions on Very Large Scale Integration Systems* 5 (1): 123–135.

Linares-Vásquez, Mario, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2015. "Optimizing Energy Consumption of GUIs in Android Apps: A Multi-Objective Approach." In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 143–154. ESEC/FSE '15.

Lipp, Moritz, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. "Meltdown." *ArXiv e-prints* (January). arXiv: 1801.01207.

Lu, Shih-Lien. 2004. "Speeding Up Processing with Approximation Circuits." *IEEE Computer* 37 (3): 67–73.

Manotas, Irene, Lori Pollock, and James Clause. 2014. "SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework." In *International Conference on Software Engineering*, 503–514. ICSE '14.

Mars, Jason, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2012. "Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up." *IEEE Micro* 32 (3): 88–99.

Massalin, Henry. 1987. "Superoptimizer: A Look at the Smallest Program." *ACM SIGARCH Computer Architecture News* 15 (5): 122–126.

Mechtaev, Sergey, Jooyong Yi, and Abhik Roychoudhury. 2016. "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis." In *International Conference on Software Engineering*, 691–701. ICSE '16.

Niu, Linwei, and Gang Quan. 2004. "Reducing Both Dynamic and Leakage Energy Consumption for Hard Real-Time Systems." In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 140–148. CASES '04.

Nowka, Kevin J., Gary D. Carpenter, Eric W. MacDonald, Hung C. Ngo, Bishop C. Brock, Koji I. Ishii, Tuyet Y. Nguyen, and Jeffery L. Burns. 2002. "A 32-bit PowerPC System-on-a-Chip with Support for Dynamic Voltage Scaling and Dynamic Frequency Scaling." *IEEE Journal of Solid-State Circuits* 37 (11): 1441–1447.

Palem, Krishna V. 2014. "Inexactness and a future of computing." *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences* 372 (2018).

Palem, Krishna V., and Avinash Lingamneni. 2013. "Ten Years of Building Broken Chips: The Physics and Engineering of Inexact Computing." *Transactions on Embedded Computing Systems* 12 (2s): 87:1–87:23.

Petke, Justyna. 2016. "Genetic Improvement for Code Obfuscation." In *Genetic and Evolutionary Computation Conference Companion*, 1135–1136. GECCO '16 Companion.

Rabinowitz, Stanley, and Stan Wagon. 1995. "A Spigot Algorithm for the Digits of π." *The American Mathematical Monthly* 102 (3): 195–203.

Reda, Sherief, and Abdullah N. Nowroz. 2012. "Power Modeling and Characterization of Computing Devices: A Survey." *Foundations and Trends in Electronic Design Automation* 6 (2): 121–216.

Rice, Henry Gordon. 1953. "Classes of Recursively Enumerable Sets and Their Decision Problems." *Transactions of the American Mathematical Society* 74:358–366.

Rinard, Martin. 2006. "Probabilistic Accuracy Bounds for Fault-Tolerant Computations That Discard Tasks." In *International Conference on Supercomputing*, 324–334. ICS '06.

———. 2007. "Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points." In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 369–386. OOPSLA '07.

Sarbishei, O., and K. Radecka. 2010. "Analysis of Precision for Scaling the Intermediate Variables in Fixed-Point Arithmetic Circuits." In *International Conference on Computer-Aided Design*, 739–745. ICCAD '10.

Sarwar, Abul. 1997. *CMOS Power Consumption and C_{pd} Calculation.* Technical report. Texas Instruments.

Schkufza, Eric, Rahul Sharma, and Alex Aiken. 2013. "Stochastic Superoptimization." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 305–316. ASPLOS '13.

Schulte, Eric, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. "Post-Compiler Software Optimization for Reducing Energy." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 639–652. ASPLOS '14.

Shehabi, Arman, Sara Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Linter. 2016. *United States Data Center Energy Usage Report.* Technical report. Ernest Orlando Lawrence Berkeley National Laboratory, June.

Sidiroglou-Douskos, Stelios, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. "Managing Performance vs. Accuracy Trade-Offs with Loop Perforation." In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, 124–134. ESEC/FSE '11.

Tian, Ye, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu. 2015. "ApproxMA: Approximate Memory Access for Dynamic Precision Scaling." In *Great Lakes Symposium on VLSI*, 337–342. GLSVLSI '15.

Tilevich, Eli, and Yannis Smaragdakis. 2002. "J-Orchestra: Automatic Java Application Partitioning." In *European Conference on Object-Oriented Programming*, 178–204. ECOOP '02.

Venkataramani, Swagath, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2015. "Approximate Computing and the Quest for Computing Efficiency." In *Design Automation Conference*, 120:1–120:6. DAC '15.

Yang, Zhixi, Jie Han, and Fabrizio Lombardi. 2015. "Transmission Gate-Based Approximate Adders for Inexact Computing." In *International Symposium on Nanoscale Architectures*, 145–150. NANOARCH '15.

Zitzler, Eckhart, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. 2003. "Performance Assessment of Multiobjective Optimizers: An Analysis and Review." *Transactions on Evolutionary Computation* 7 (2): 117–132.