J. Parallel Distrib. Comput. 96 (2016) 121–133

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

enDebug: A hardware–software framework for automated energy debugging

Jie Chen, Guru Venkataramani *

The George Washington University, Washington, DC, United States

HIGHLIGHTS

- We explore the design of a hardware–software cooperative energy profiler.
- We design automated recommendation system using the guided genetic algorithm to explore energy optimizations in the program code.
- Our guided genetic algorithm can substantially reduce program energy on top of the highest GNU C compiler settings.

ARTICLE INFO

Article history:
Received 13 June 2015
Received in revised form
1 February 2016
Accepted 2 May 2016
Available online 24 May 2016

Keywords:
Energy profiling
Energy optimization
Genetic programming

ABSTRACT

Energy consumption by software applications is one of the critical issues that determine the future of multicore software development. Inefficient software has been often cited as a major reason for wasteful energy consumption in computing systems. Without adequate tools, programmers and compilers are often left to guess the regions of code to optimize, that results in frustrating and unfruitful attempts at improving application energy. In this paper, we propose enDebug, an energy debugging framework that aims to automate the process of energy debugging. It first profiles the application code for high energy consumption using a hardware–software cooperative approach. Based on the observed application energy profile, an automated recommendation system that utilizes artificial selection genetic programming is used to generate the energy optimizing program mutants while preserving functional accuracy. We demonstrate the usefulness of our framework using several Splash-2, PARSEC-1.0 and SPEC CPU2006 benchmarks, where we were able to achieve up to 7% energy savings beyond the highest compiler optimization (including profile guided optimization) settings on real-world Intel Core i7 processors.

1. Introduction

Innovations in computer architecture and semiconductor technologies have increased the computational performance of systems exponentially. At the same time, energy cost incurred by software applications on servers and network devices has grown rapidly resulting in the need to adopt energy-aware software development methodologies beyond just relying on dynamic, hardware-level techniques. Addressing energy fundamentally at the application source code level yields much higher benefits by tuning the source code for energy, and saves the cost of dynamically deploying energy saving mechanisms at runtime.

In 2014, energy is projected to be a major expense for most major data centers, and is estimated to account for up to 3% of world's total energy consumption [34]. To highlight the importance of application energy and its impact on data service efficiency, companies like eBay have online dashboards for the users to track energy consumption per transaction in their servers [11]. Such efforts clearly illustrate the motivation of major software giants in removing the inefficiencies in their applications, that will eventually only lead to higher energy bills without driving up the revenue or performance. A plethora of solutions ranging from virtualization to application aware power management has been proposed to reduce system energy footprint. While such techniques are useful, a more effective solution is to incorporate energy smartness into the software itself such that the application performance can be aligned more closely with revenue. Being able to automate the process of energy debugging would be vital to the future of energy-aware software development.

Conventionally, execution time of applications is a commonly adopted proxy measure for software developers to identify the energy bottlenecks in their program code. Recent studies by Hao et al. [17] have shown that the execution time and energy consumption do not have a strong correlation because of several

* Corresponding author.
E-mail addresses: jiec@gwu.edu (J. Chen), guruv@gwu.edu (G. Venkataramani).

http://dx.doi.org/10.1016/j.jpdc.2016.05.005
0743-7315/

factors such as (1) multiple power states—at two different frequencies f_1 and f_2, even if the execution times are the same, the energy drawn will be different, (2) asynchronous design of system and API calls—when the application sends data over the network, the data is handled by the OS which results in the corresponding data-sending application not being charged for the data transmission time. These necessitate a dedicated framework for energy debugging.

Further, the application energy profile and optimizations are often specific to the processor architecture and hardware configurations. That is, a set of code optimizations that improve energy in one processor configuration does not necessarily improve the energy in other platforms. Without having a sound understanding of the underlying hardware details, software-only energy accounting often ignores the numerous and complex interactions between shared hardware resources and overlapped instruction execution that occur during application runtime. To overcome such problems, it is critical to debug the energy consumption of applications through profiles generated on the target architecture.

In this paper, we present *enDebug*, a hardware software cooperative framework that attributes energy consumption by applications to fine-grained regions of program code (say functions), and utilizes an automated recommendation system to explore energy improvements in the program code. In doing so, we enable the participation of software developers and toolchains (such as compilers and runtime) in energy-aware software development without necessarily having them to rely on expensive runtime energy saving strategies.

We utilize mostly existing hardware support with minimal modifications to accurately and efficiently gather energy-related metrics in applications, and then use software support to compute the energy. This two-step strategy provides better precision in targeting code regions for energy optimization without having to *guess* or *resort to proxy measures*.

Once the energy-expensive code regions are located, a genetic programming-based *automated recommendation system* shall explore opportunities to improve application energy by performing a set of mutations on the program code. In contrast to prior approaches [40], we adopt a *guided*[1] version of genetic programming that uses heuristics based on profiler feedback. This helps us to effectively apply processor- and workload-specific energy optimizations to the program code, and avoids having to deal unnecessarily with the program versions that may not optimize energy. When tested using the benchmarks from PARSEC-1.0 [3], Splash-2 [49] and SPEC CPU2006 [43] suites using Intel Core i7 processors, we were able to obtain up to 7% better energy savings (in Fluidanimate benchmark) over highest optimization settings (-O3 and with profile guided optimizations or PGO) in the GNU Compiler Collection [14].

The contributions of this paper are as follows:

- We motivate the need for application energy profiling, and propose *enDebug* that incorporates a hardware–software cooperative profiler and an automated recommendation system to explore energy optimizations in the program code.
- We explore a fine-grained energy estimation methodology that is largely based on existing hardware support with minimal additions to the hardware.
- We design an Artificial Selection Genetic Programming (ASGP) algorithm that performs genetic mutation operations guided through heuristics generated by the hardware profile. Our ASGP algorithm explores several code mutation operations to reduce energy on the program code regions identified by the energy profiler.
- We evaluate our framework using several code samples from Splash-2, PARSEC-1.0 and SPEC CPU2006 benchmark suites. Our results indicate that we are able to achieve up to 7% additional energy savings beyond the highest optimization settings enabled in GNU Compiler Collection [14].

[1] We call this process as Artificial selection instead of natural (random) selection used in most genetic programming algorithms.

Table 1
Energy and performance profile of functions in Splash-2 and PARSEC-1.0 benchmarks with 8 threads.

Benchmark	Function	% of Energy	% of Time
Ocean	relax	30.31%	15.02%
	slave2	18.98%	30.30%
	jacobcal2	14.47%	12.22%
	laplacalc	12.68%	9.85%
Radiosity	v_intersect	11.06%	6.60%
	compute_diff_disc_formfactor	7.39%	14.07%
	traverse_bsp	4.83%	5.53%
	four_center_points	3.03%	5.06%
Bodytrack	InsideError	25.38%	9.12%
	Exec	19.20%	4.34%
	EdgeError	18.53%	6.77%
	ImageProjection	10.66%	3.67%

2. The need for energy debugger

With increasingly complex interactions between instruction execution and the associated timing in the processor pipeline (due to parallelism), execution time can no longer be considered a good proxy for accurate energy measurement. To illustrate this effect, we conduct experiments on several real-world applications from Splash-2 [49] and PARSEC-1.0 [3] benchmark suites, where the energy consumption characteristics of individual functions drastically differ from their corresponding execution time profiles.

Table 1 shows the energy and execution time profiles on several applications with 8 threads running on 8 cores. All of our experiments were done using SESC [36], a cycle-accurate, multicore architecture simulator that is integrated with McPAT power model [29]. Table 2 shows the processor configuration details as input to the McPAT power model.

1. **Ocean**: relax() consumes 30.31% of the total energy but only accounts for 15.02% of the total execution time. On the other hand, slave2() accounts for 30.30% of execution time, but only consumes 18.98% energy. Upon further examination, we observed *highly overlapped execution* of double-word arithmetic instructions in relax() led to higher energy with lower execution time. However, slave2() had higher numbers of branch and load/store instructions leading to longer execution time despite consuming lower energy than relax().
2. **Radiosity**: v_intersect() consumes 11.06% of total energy with only 6.60% of the total execution time, while compute_diff_disc_formfactor() has 14.07% of the total execution time with only 7.39% of the total energy. On a closer review, we found that v_intersect() heavily used complex instructions like madd.d (that perform multiply–add of double word values) leading to higher energy, while compute_diff_disc_formfactor() had a lot of load operations leading to higher execution time despite consuming lower energy than v_intersect().
3. **Bodytrack**: The top four energy consuming functions account for 74% of the total energy, but only account for about 24% of the total execution time. About 64% execution time is actually spent on lock and barrier synchronizations implemented by pthread_cond_wait() that actually puts threads into sleep without consuming much energy.

The examples above clearly show that a debugging framework is necessary to better understand the energy profile of applications beyond just performance, and use this profile information for energy optimization.

Table 2
Processor configuration and power model.

Processor	3 GHz, 8-core CMP; 4-wide issue/retire, out-of-order execution; 8-entry instruction queue; 4096-entry BTB, hybrid branch predictor; 176-entry ROB; 64-entry LD/ST queues; 48-entry scheduler; 90 floating point registers; 96 integer registers;
Memory Sub-system	32 kB, 4-way, I-cache; 32 kB, 4-way, D-cache; 256 kB, 8-way, private L2 cache; 16 MB, 16-way, shared L3 cache; 64-entry ITLB/DTLB;
Interconnect	Shared bus below private L2 caches
Power model	McPAT, 32 nm, $V_{dd} = 1.25$ V

Are current hardware energy meters sufficient? Modern high performance processor architectures [24,37] have begun integrating hardware energy meters that can be read through software driver interfaces. For example, starting from Sandy Bridge, Intel provides a driver interface called RAPL (Running Average Power Limit) that can let programmers periodically sample processor energy usually at the granularity of a few milliseconds of program execution time. While this is going to be a useful first step toward helping programmers to understand the processor energy consumption, it is still far from providing them with a more practical feedback at a granularity that relates the processor energy consumption back to the program source code.

3. Fine-grained energy profiling

To help compilers, runtime optimizers or even programmers effectively apply the energy optimizations to the right code regions, energy profile information is needed at the level of fine-grained code, say functions or certain critical loop structures. We note that the current hardware energy profiling infrastructure such as RAPL interface [37] can provide energy information at a granularity of several microseconds to a few milliseconds. To bridge the gap between the current hardware support and the fine-grained energy profile (needed to attribute energy back to application source code), we undertake a two step strategy—*First*, we build an energy model (regression) using certain well-known hardware performance counters. *Second*, we show how fine-grained energy for functions can be obtained using simple hardware support and the energy regression model from the first step. Essentially, our solution is able to estimate energy and attribute them at the granularity of program functions without requiring extensive hardware support.

We note that the first step (building the energy model) can be done on many current processor platforms that have support for hardware energy and performance counters (which can be read using lightweight tools such as likwid [45]). However, the second step of fine-grained profiling and attribution does not exist in any modern processor platform to the best of our knowledge. Therefore, to provide an accurate view of our energy profiler, we implement the entire fine-grained energy profiling framework using SESC [36], a cycle-accurate, out-of-order issue, multi-core processor simulator in which we custom integrated the McPAT energy model [29].

3.1. Energy model using performance counters

Current hardware support to measure energy and power consumption [37] is at a very coarse granularity that cannot be directly used by code optimizers. Therefore, to attribute energy consumption to fine-grained regions of code (say functions), we first build an energy model that estimates energy consumption through regression on a set of hardware performance counters.

Prior works [21,33,22,10,9] have shown that linear regression is an efficient solution to estimating processor energy with high accuracy. The energy model builds a relationship between the processor energy, denoted as E, and a set of key performance metrics, denoted as P. Most modern processors have hardware performance counters that can dynamically capture many critical performance metrics made available through the performance monitoring infrastructure. With this existing hardware support, we build our energy model as follows: We choose a subset of benchmarks from Splash-2 [49] and PARSEC-1.0 [3] to train our energy model. In each benchmark run, for every 1 million cycle windows, we gather CPU clock cycles, instruction count, L1 Data cache access count, L2 cache access count, floating point operation count through hardware performance counters. We also tabulate the corresponding energy consumption within the observation period from the energy counters. An alternative, but less efficient approach is to use basic blocks as P as shown in our prior work [8], which, nevertheless, requires training regression model over a large number of parameters. Using the measured features, the energy consumption E (in nanojoules) is estimated as a function of the key performance metrics.

$$E = 1.347 * CPUCycles + 0.484 * InstCount + 0.867 * L1DCacheAccess + 1.097 * L2CacheAccess + 0.104 * FloatOps.$$

A separate set of Splash-2 and PARSEC-1.0 benchmarks (non-overlapping with the training set) is used for testing and validation. We calculate the relative error between the estimated core energy and the ground-truth energy numbers for the core using McPAT [29]. To assess the robustness of our trained energy model, we adopt Ten-fold Cross-validation [28] method where 90% of the samples are used as the training set and the remaining 10% of them are used for validation. This step is repeated ten times where a different validation set is selected during each time. In our experiments, we found that the average cross-validation errors are 2.13%, and the worst-case absolute error to be less than 12%. To calibrate the model on a different processor, the above modeling, testing and validation steps should be performed again. Note that our purpose of using the energy model is to identify fine-grained functions that are energy-hungry, while measuring the whole (end-to-end) program energy consumption on a real processor was done using RAPL.

3.2. Attributing energy to program functions

To attribute energy back to the program source code, an important question that needs to be answered is the granularity at which energy should be attributed. Note that the program code can be analyzed at different granularities, for example, instructions, basic blocks, functions, or the whole program. Attributing energy to individual instructions and basic blocks is practically very difficult because modern superscalar processors can execute multiple instructions (and basic blocks) in an overlapped fashion. This makes it very hard to accurately attribute energy back to each of such entities. On the other hand, with the energy profile information at larger granularities such as whole program level,

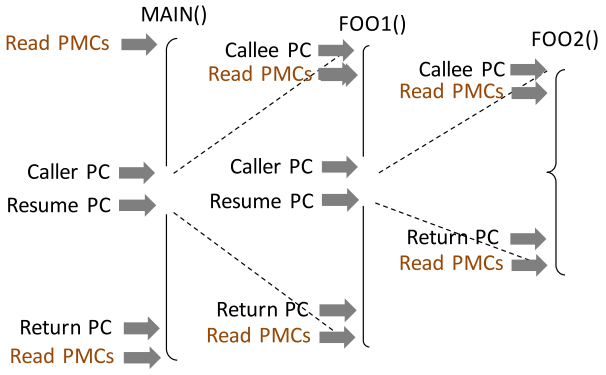


Fig. 1. Recording performance counters and PCs at function calls and returns.

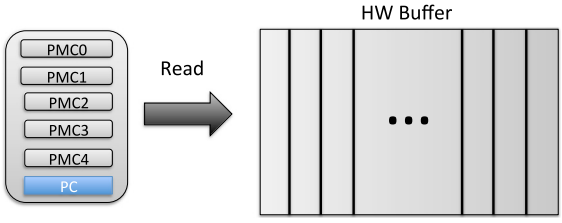


Fig. 2. Storing performance counters and PCs to the hardware buffer.

we may not have meaningful feedback to effect code changes. Therefore, we attribute energy to program functions, that already have programmer-defined boundaries and have a more bounded scope than the entire program (see Figs. 1 and 2).

To attribute the energy profiles to the corresponding functions, during every function call, we record the (1) performance counters from Section 3.1, (2) *Program Counter* (PC) at the caller site and the (3) callee function address (that can be later attributed back to program source code using well known tools such as GNU addr2line utility [14]). Similarly, during function return, we also record (1) the performance counters from Section 3.1, (2) the PC at the return site and (3) the PC at the caller function after return. At the end of the program execution, the corresponding energy profiles of function fragments (due to possibly multiple calls and returns between caller/callee pairs) are accumulated and attributed back to individual functions in the program code.

We use a 16-entry hardware buffer to record the performance counters and the PCs. The hardware buffer values are periodically logged into a file that can later be analyzed further by software modules. In particular, the logged values and the regression coefficients from Section 3.1 can be used to determine the energy consumption by individual functions during program execution. Since the hardware buffer is off the critical path, and has access latency of 0.18 ns using Cacti 5.1 [25] (less than one CPU clock cycle even on a 3 GHz processor), it is unlikely to adversely impact the processor performance. We note that not all function calls are executed through *call* instructions, for example, compilers may generate function calls using *jmp* instructions, e.g., virtual functions are implemented using jump table in C++. In such cases, we can still correctly identify functions by monitoring the execution of *push* instructions (stack pushing operations) before *jmp*.

4. Automated recommendation system for energy optimization

Once the profiler identifies the application code regions (functions) that are behind high energy consumption, the next logical step in energy debugging is to explore automated ways to optimize energy. Frequently, this step lends substantial benefits in large scale software development that involves a huge code base and numerous input test samples.

To automate the process of energy optimization, we design a genetic programming-based algorithm in our approach. This algorithm uses *artificial selection* or guided mutation strategy to create energy-improving program code mutants based on the observed runtime profile.

In this section, we will first briefly describe the basics of genetic programming, and then show the design of our algorithm explaining why artificial selection is necessary.

4.1. Basics of genetic programming

Genetic programming (GP) [5] is the process of evolving computer program code to inductively find preferable program output given an input program. The process involves a series of mutations (or code transformations) to the input program such that we eventually arrive at a preferred output. Similar to animal and plant species evolving through natural selection process, GP applies a set of powerful genetic operators to *randomly* alter the "gene" (or code structure) in the program. These genetic operators range from simple replication of the program code to random variations on the inner parts of a program. Once the randomly selected genetic operators are applied, GP breeds new mutants that may or may not resemble the original program from which they are mutated. GP usually tests if these new mutants are qualified to remain in the evolving population using a fitness test. The fitness test is defined by a function F which measures the error distance between the mutant program output and the preferred outcome. Examples of fitness tests include testing for functional correctness (i.e., the mutant has the exact same functional behavior as the original), performance improvement, power reduction, or combinations of one or more of such factors. If a mutant is *deemed fit*, it is usually retained and allowed to join the evolution population (i.e., allowed to participate in the next round/generation of mutations). The premise is that, by breeding new programs from this dynamic pool, GP will find program mutants that will ultimately have the preferred program output.

Note that fitness test is usually the most time consuming step in the entire process of GP evolution. For example, if the fitness test involves checking for functional correctness, a representative range of possible input values have to be supplied to the program to make sure that the original program and its mutant versions behave alike on all of such representative inputs. As a result, natural selection or unguided mutation could take several hours to produce useful mutants, and is popularly dubbed as *overnight optimization* [40].

4.2. Artificial selection genetic programming

To overcome the challenges associated with natural (or unguided, random) selection based approach, we propose artificial (or guided) selection for genetic operator selection. In biological terms, artificial selection (also known as Selective Breeding) refers to the process of human-controlled mutation and breeding that accelerates the evolution of preferable traits, instead of relying on slow natural selection. In our approach, we apply a set of code transformations guided by heuristics derived from program code structure or runtime profile. Our fitness function expects to see lesser energy consumption in the offspring than the parent, while making sure that functional correctness is preserved on a range of inputs. In this vein, we name our algorithm as Artificial Selection Genetic Programming (ASGP, for short).

4.3. ASGP algorithm

As a first step, a control data dependency graph (or CDDG) is used to represent the profiler-identified high energy program

functions. Algorithm 1 shows how our CDDGs are constructed from the program assembly code. Our graph construction algorithm goes through every constituent basic block and builds a data dependency subgraph for each of them starting from the very last instruction (branch) and proceeding backwards until the first instruction in the block by tracing the dependency of data values. A directed edge from node A to node B means that node A depends on node B's output to compute its own output. After constructing data dependency subgraphs, our graph construction algorithm will connect all of the subgraphs based on control dependencies, that is, each basic block's branch node will be connected to all its target nodes in other subgraphs.

Algorithm 1: CDDG construction algorithm

```
input  : Assembly code of the profiler-identified function
output: A Graph G that shows data and control dependencies

//Construct data dependency subgraphs for every basic block;
foreach basic block in the identified function do
    Create an empty set S;
    //Go through instructions in reverse order;
    foreach instruction in the basic block do
        //This if-else only applies to instructions that has
        destination operand;
        if Operand_dst is not in S then
            Create a new node with label Operand_dst;
            Add Operand_dst to S;
        else
            Delete Operand_dst from the S;
            Delete memory operands derived from Operand_dst
            from S;
        end
        Include Opcode in the node labeled with Operand_dst;
        foreach Operand_src in the instruction do
            if Operand_src is not in S then
                Create a new node with label Operand_src;
                Add Operand_src to S;
            end
            Create a directed data dependency edge from
            Operand_dst to Operand_src;
        end
    end
end

//Connecting sub-graphs through branch nodes;
foreach branch node (with branch Opcode) do
    Locate all target nodes in the graph;
    Create a control dependency edge from target nodes to this
    branch node;
end
```

To further assist the ASGP algorithm, certain nodes in the CDDG are annotated with runtime profile information. For example, the branch nodes are annotated with the taken frequency (from which branch taken vs. non-taken ratio can be derived). We note that these annotations based on runtime information are simply to guide our ASGP algorithm to make decisions in an informed manner rather than arbitrary, random choices.

We note that instruction operands can be immediate, register, or memory. Among these, memory-type operands need special attention due to unknown memory dependencies at compile time. In several popular ISAs such as x86, data is read or stored into a particular memory address by using different memory addressing modes, such as direct displacement, register indirect, base indexed. Note that registers are specifically used as base or index registers as part of the calculation of memory address where data is read or written. In order for CDDG to track the true dependencies between two memory-type operands, we monitor for changes to the registers that are used as base and/or index in memory addressing. For example, we mark that a memory-type Operand_dst node from instruction *A* depends on a memory-type Operand_src node from instruction *B* only if all the memory address-related registers used in both A and B do not get updated by any other instructions between A and B. In other words, if there is any update to a register used as an index/base in another instruction's operand, we do not establish dependency between the instructions A and B. In addressing modes, such as memory indirect, where it is more complex to resolve memory dependencies prior to runtime, we chose to represent these operands as two separate nodes in CDDG. To address situations where multiple operands might write to the same memory location using different base and index registers (memory aliasing problem that are usually unresolved prior to runtime), our CDDG algorithm flags all of the instructions that have memory operands as destinations. Our genetic programming algorithm, ASGP, can use this as a guide to prevent reordering of instructions past these instruction boundaries and potentially avoid memory conflicts. Note that our fitness test acts as a safety net, and will fail if ASGP unintentionally created any memory conflicts.

Algorithm 2: Artificial Selection Genetic Programming Algorithm

```
input  : Original program (assembly)
output: Lesser energy consuming program mutants

Initialize the population with the original program code and
mutants derived using neutral transforms;
Run the initial mutants and obtain their energy consumption;
repeat
    Randomly select one mutant from the evolution population
    as parent;
    Mutate using one or more genetic operators with triggering
    heuristics;
    Apply fitness test on the evolved mutant code;
    if the mutant passes the fitness test then
        Replace its parent in the evolution population;
    else
        Discard the mutant from the evolution population;
    end
    At every fifth generation, discard mutants with energy
    greater than original program;
until maximum number of generations is reached from all of the
initial mutants;
Output the mutant that consumes the least energy in the
evolution population;
```

Based on the information contained in the CDDG, the first generation mutants are evolved using *neutrally transformed* program mutants from CDDG. Neutrally transformed mutant refer to a program version that preserves the functional equivalence derived either by applying algebraic rules or via merging opcodes already available from the ISA (see Section 4.4). This step helps initialize the evolution population with multiple mutant samples (with functionally equivalent versions to the original program) such that the ASGP algorithm can do a more effective exploration by applying its genetic operators on this population. Successive generations of mutants are created by applying genetic operators on the evolution population guided by triggering heuristics (see Section 4.5). The fitness function tests whether the new mutant preserves the functional equivalency on all of the available input sets while potentially optimizing energy consumption over its parent mutant, and then is added to the evolution population. In other words, if the mutant is not functionally equivalent to the original program or does not show the potential to improve energy over its parent, the mutant is discarded from further

consideration. We determine functional equivalency by comparing all of the registers (data, flags and program counters) and the set of all memory locations modified by the original program and the mutant in the *region of interest* (i.e., where the mutation operation was applied). The *potential for energy optimization* is defined as the mutant that already lowers energy consumption of the parent or is within 5% above the parent's energy consumption. We allow this slight increase in energy consumption among the mutants to avoid being stuck in local minimas and being unable to find more energy optimizing mutants in the future without exploring slightly more energy-expensive mutants. However, to avoid the mutant population from being polluted with higher energy consuming variations, we periodically (after every five generations in our implementation) remove the mutants that consume energy higher than the original input program.

After reaching a maximum number of generations starting from all of the initial mutants, the algorithm stops. The maximum number of generations is input by the user based on the number of tries (or cost) that she is willing to pay for the ASGP algorithm to explore the program mutants.

4.4. Neutral transforms

If unguided or natural selection process is used to create initial mutants, most of them will likely not be functionally equivalent to the original program, thus failing the fitness test. As a result, a significant amount of computational time is wasted in evolving these random programs. To avoid wasting time on evolving such "pure random" programs, our current implementation of enDebug uses the following set of *neutral transforms* on the original CDDG to populate the set of initial program mutants, and hence increase the efficiency of our approach.

1. **Sign conversion**—This helps generate complements of numbers that can be primarily exploited to cluster certain types of operations and reduce energy if such operations are supported by the underlying ISA. For example, $a - b + c + d$ can also be represented $a + (-b) + c + d$ that can be utilized by vector operations to cluster operands with the same operator.
2. **Commutativity**—Commutativity applies on arithmetic operations such as add and multiply nodes, for example, $a+b = b+a$, $a \times b = b \times a$. This can be used to group clusters of operands in nearby memory locations and potentially improve spatial locality to save energy.
3. **Distributivity**—Distributivity also applies on arithmetic operations that helps to reduce the number of multiplication/division operations in the program. For example, $a \times b + a \times c = a \times (b+c), a/b+c/b = (a+c)/b, (a/b)/(c/d) = (a \times d)/(b \times c)$, $(a/b)/c = a/(b \times c)$.
4. **Merge**—This serves to optimize energy consumption by combining certain operations into ISA-supported more complex types. Energy savings can be had from lesser number of instruction fetches and execution. Examples of merge operation include: 1. when nearby nodes in CDDG have mul.d and add.d, the ASGP algorithm can consider replacing the node with madd.d in certain ISAs, 2. when nearby nodes in the CDDG have load and address increment/decrement, a vector load can be applied.

   In the benchmark suites that we studied in this work, we have observed that a substantial amount of program code involves algebraic computations. This provides us with many code regions that are amenable to transforms using algebraic rules (sign conversion, commutativity rule and distributivity rule) and ISA-specific transforms (merge certain operators into more complex types). We note that more neutral transformation rules could always be explored and used for other applications.

4.5. Genetic operators and triggering heuristics

We first describe the four standard genetic operators [5] that are used in our genetic algorithm, and then summarize the heuristics that guide the usage of such operators on program mutants.

1. **Delete**—Delete operator serves to eliminate the CDDG nodes that unnecessarily increase the program energy consumption. As examples, 1. when a branch is always not taken for all inputs, the branch instruction and all instructions that compute the condition value for the branch can be deleted (see Fig. 3(a)), 2. certain subexpressions or instructions can be removed that may be eventually moved to another place in the program code.
2. **Copy**—Copy operator works either on the block level or individual instruction level. At the block level, copy operator can potentially increase the ratio of useful code while reducing energy spent on meta code and the related control transfer instructions. When copy operator is applied, the subgraphs need to be properly replicated (similar to loop unrolling)—memory indices should be correctly adjusted, common registers should be renamed, and the dependency edges be connected to the right node. Potential example applications of this operator include the loop structures that were not able to be unrolled by the optimizing compiler at static time; at the instruction level, copy operator helps achieve physically moving an instruction to another place. This movement of instruction requires copy to be jointly used with delete operator.
3. **Swap**—Swap operator swaps the positions of two nodes (instructions) or subgraphs (blocks). This can be used to accomplish useful transforms such as code reordering. As examples: 1. If consecutive nodes exhibit long latencies due to stalls in integer execution unit (resource contention), swapping this node with an another node (that has a different set of operations not competing for the integer execution unit) would alleviate the energy wasted over stall time. 2. In a if..elsif..else code, let us say that the percentage of execution of if{}, elsif{} and else{} blocks are 10%, 10% and 80% respectively. A swap of the else{} and if{} blocks would save the unnecessary control transfer through two basic blocks a majority of the time.
4. **Crossover**—Crossover operator takes subgraphs from two parent mutants, and creates two new offsprings. For example, between two parent mutants A and B, exchanging subgraphs SG3 and SG6 results in an improved offspring, A (see Fig. 3(d)). Specifically, we look for mutants that show reduced energy consumption compared to their parents, and mark the mutated subgraphs (the portion of program code that was mutated in the parent). Then a crossover operation is performed to check if two mutants having marked subgraphs from different program locations will create a new mutant that combines the energy savings from both of its parental mutants.

Table 3 summarizes some of the possible heuristics that were used by our ASGP algorithm to find program mutants that can lead to better energy consumption. We note that this table shows a list of heuristics used in our current implementation rather than being exhaustive.

Tables 4 and 5 show the number of distinct mutants generated by the unguided (natural selection) and guided (artificial selection) algorithms assuming that the user has no limit on the maximum number of generations. We found that the energy-optimized mutants found by both the GP and the ASGP algorithms were identical for each of our benchmark. In each experiment, the mutants that are not deemed fit by the fitness function either due to functional incorrectness or not exhibiting the capability to reduce energy are considered as *discarded mutants* (refer Section 4.3). Functional correctness was determined by comparing the outputs of the original and the mutant codes on all of the input sets made available by the benchmark developers. All of the mutants that pass the fitness test

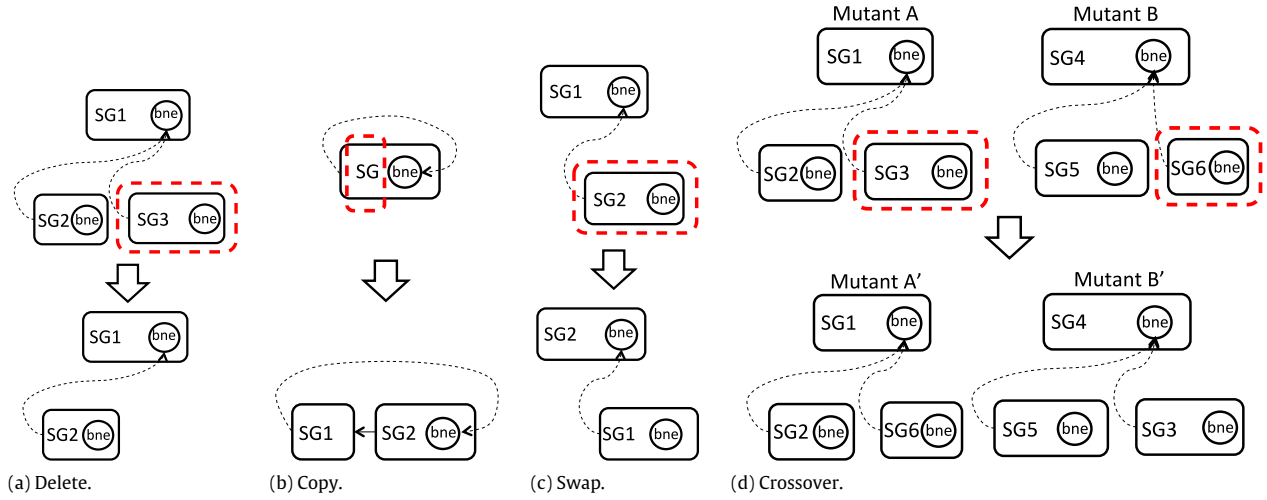


Fig. 3. Examples of genetic operator applications on CDDG.

Table 3
Triggering heuristics for genetic operators.

Operator/transform	Triggering heuristics
Delete	Always taken/not taken branches, redundant instruction
Copy	High branch to non-branch instruction ratios, known iterator count values
Swap	Skewed frequencies of execution in branch chains, cluster of instructions of similar type
Crossover	Energy optimized mutants in two separate subgraphs among two different mutants

Table 4
Number of mutants for the un-guided natural selection GP algorithm.

Application	# discarded mutants	# kept mutants
Fmm	558	15
Ocean	440	20
Cholesky	40	19
Water-sp	290	33
Water-n2	290	33
Fluidanimate	275	11
Streamcluster	486	14
Libquantum	344	19
Sphinx3	427	22

Table 5
Number of mutant for our ASGP algorithm with triggering heuristics.

Application	# discarded mutants	# kept mutants
Fmm	13	11
Ocean	14	17
Cholesky	0	14
Water-sp	31	8
Water-n2	31	9
Fluidanimate	11	19
Streamcluster	5	7
Libquantum	7	12
Sphinx3	16	9

are *kept mutants*, i.e., considered part of the evolution population. As seen in our experiments, the ASGP reduces the total number of mutants anywhere from 4× (in Cholesky) to over 40× (in streamcluster). This helps to speedup the process of choosing energy optimized mutants of the program code.

Table 6 shows the execution time (in seconds) for an un-guided natural selection GP algorithm and our guided, artificial selection GP algorithm. We note that the execution time includes the time to run the algorithm and fitness test for each of the benchmarks. In every benchmark, functional correctness was determined through multiple runs of all of the input sets made available by the benchmark developers. We observe a speedup of 4.1× (in Cholesky) and up to 41.7× (in streamcluster).

Table 6
Execution time overhead comparison between the natural selection GP algorithm and our heuristic-triggered ASGP algorithm.

Application	GP (s)	ASGP (s)	Speedup (times)
Fmm	701.6	28.1	25.1×
Ocean	705.1	47.6	14.8×
Cholesky	19.5	4.7	4.1×
Water-sp	145.7	17.6	8.3×
Water-n2	936.6	116.3	8.1×
Fluidanimate	1354.9	142.1	9.5×
Streamcluster	2881.5	69.2	41.7×
Libquantum	5227. 5	273.6	19.1×
Sphinx3	26 994.2	1563.1	17.2×

In this work, we rely on developer-supplied input sets to make sure that energy optimized mutants will work correctly for user-specified inputs. Note that it is important for developers to carefully select the most typical input sets such that ASGP is optimizing energy for the most common usage scenarios of the program. Using less-representative or completely random input sets to ASGP may not yield intended energy benefits, and may even adversely affect energy consumption leading to unproductive optimizations to program code. For a given input test, our ASGP algorithm uses hardware support to rigorously compare the values produced by the mutated code and the original code, which will be described in Section 4.6. Exhaustively testing a program with all possible inputs for correctness is not feasible even in unmodified programs. We assume that the programmer who wants to optimize her program for energy using our approach is aware of critical input parameters and values for which the program correctness needs to be preserved. Prior works such as [40] have made similar assumptions to verify the correctness of the mutant code. In an environment where function correctness has to be verified over a wide range of inputs, automatic input testing tools can be deployed to help ease the verification process before the mutant program went for deployment. For example, random test input generation tools, such as DART [15] and CUTE [41], can be used for the purpose. Note that such extra tools may incur significant performance

Table 7
Energy reduction in benchmarks as measured by SESC simulator (Baseline: GCC -O3).

Application	ASGP-evolved code region	# code changes	Energy *reduction*[a]	Energy *reduction*[b]
Fmm	interaction.C: line 398	16	3.7%	4.1%
Ocean	jacobcalc.C: line 310	15	2.2%	4.3%
Cholesky	numLL.C: line 436 & 473	44	1.2%	1.4%
Water-sp	cshift.C: line 58	34	1.9%	2.7%
Water-n2	cshift.C: line 54	34	2.3%	1.3%
Fluidanimate	parallel.cpp: line 689	12	4.3%	4.2%
Streamcluster	streamcluster.cpp: line 159	31	18.0%	15.5%
Libquantum	gates.c: line 74	29	3.6%	3.8%
Sphinx3	cont_mgau.c: line 575	12	1.1%	1.4%

[a] Running with input ref1.
[b] Running with input ref2.

Table 8
Changes in application-related performance events (positive number indicates reduction in the number of events; negative number indicates increase in the number of events).

Application	CPU cycles	Inst count	L1DCache access	L2Cache access	L2 misses	Float Ops
Fmm	1.5%	9.1%	−0.4%	−0.1%	0.1%	0.0%
Ocean	2.1%	1.9%	2.5%	1.9%	0.0%	6.4%
Cholesky	0.7%	1.4%	1.1%	−0.2%	0.0%	0.0%
Water-sp	2.3%	1.9%	0.4%	0.0%	0.0%	0.0%
Water-n2	3.8%	1.9%	0.8%	0.0%	0.2%	0.0%
Fluidanimate	5.1%	3.6%	5.7%	13.1%	3.0%	0.4%
Streamcluster	13.6%	11.2%	0.0%	0.0%	0.1%	0.0%
Libquantum	5.6%	0.1%	0.1%	0.1%	0.0%	0.0%
Sphinx3	1.4%	0.1%	0.1%	−0.1%	0.0%	0.0%

overheads during program optimization phase without necessarily adding substantial benefits to the outcome.

4.6. System support for automated energy optimization

The energy regression function can derive its inputs from the already available hardware performance counters. To attribute the energy back to functions, we need to log information about the performance counters and the corresponding program counters in a separate hardware buffer.

Our ASGP algorithm can be implemented as a compile-time optimizer with most mutations that can be applied during compilation time. A few operations such as merging ISA supported operations can be easily done during post-compilation phase. On the hardware side, the algorithm needs support for three modules, energy meter, functional correctness (fitness test) and heuristics. For energy metering, an off-the-shelf energy measurement interface such as Intel RAPL would suffice since the objective is to obtain overall program energy. However, for verifying functional equivalence, we need system support that will compare the values produced by mutated regions of program code and the original code. This requires hardware support to track all of the registers (including data, flags and Program Counter) and the set of memory locations altered by the program in the mutated region of program code. To this end, we design a small hardware structure, *hardware shadow buffer*, that maintains a shadow copy of the registers and the modified memory locations. Empirically, we determined that the maximum number of memory locations that were modified in energy critical code sequences were typically less than 4 kB. Therefore, we design our shadow buffer to store up to 4 kB memory locations. Using Cacti 5.3 [25], we found that the area overhead was 0.31 mm^2 and the shadow buffer's per-access latency was 0.26 ns. *Start_log* and *Stop_log* instructions are added to the ISA for recording values. In this work, we annotate the code regions that are transformed and check the register and memory values at the entry and exit points of these annotated regions. With further hardware support, we can do it at finer granularities. Finally, most of our heuristics can be derived in a relatively straightforward manner using existing performance counters in most modern architectures such as branch taken/not taken profile, and resource stalls [20].

5. enDebug evaluation

We evaluate over 30 Splash-2, PARSEC-1.0 and SPEC CPU2006 applications, and summarize the energy-related code optimizations that we found using our enDebug framework on eight such representative cases below. We perform our first round of experiments on SESC simulator with an integrated McPAT model. This is due to ease of verifying the functional correctness of the mutants through comparing all of the register and memory values produced by the mutant with the original code. We were unable to turn on profile guided optimizations (PGO) on our GCC cross compiler infrastructure for MIPS. Table 7 summarizes the observed energy savings on two different reference input sets with -O3 settings without PGO. Our simulator experiments here model an Intel i7-like processor (see configurations in Table 2) with a DRAM-based main memory. We model an 8 GB DRAM with 8 banks (at 32 nm technology node) and derive its dynamic energy numbers using the CACTI tool [25].

To understand the implication of ASGP optimization on hardware performance events, we measure changes in various factors such as CPU cycles, instruction counts, L1 data cache accesses, L2 cache accesses, L2 cache misses, and the number of floating point instructions as a result of code optimization. The results are summarized in Table 8 when applications are run using one of the reference input sets (ref1 from Table 7). In terms of instruction count, the highest reduction was seen in Streamcluster (11.2%). In terms of floating point operations, Ocean had the highest reduction (6.4%). For cache accesses, we observed highest reduction in Fluidanimate (5.7% in L1 data cache, 13.1% in L2 cache, and 3.0% in L2 cache misses). We observed reduced number of CPU cycles in all applications, this was mostly due to reduced number of executed instructions and improved memory accesses.

1. **Fmm**—Fig. 4 shows the high energy consuming code region found by our energy profiler. Using our profiler-guided heuristics, the ASGP algorithm applied the *copy* operator based

on high branch to non-branch instruction ratio inside the tight loop. After a few rounds of delete and some swap operations among instructions, the algorithm outputs a mutant with least amount of energy consumption. We found that the final mutant produced by ASGP algorithm did not have the if..else.. control part, and the energy consumption had dropped at least 3.7% during our fitness test. This code is functionally equivalent since the COMPLEX_SUB works on odd j, and COMPLEX_ADD on even j values that is replaced by j and $j+1$ during *copy* operation.

2. **Ocean**—Fig. 5 shows the high energy consuming code region found by our energy profiler. Within this dense loop, eight local variables ($f1$ to $f8$) are calculated, and then all of them are added together to find t1c[iindex]. ASGP found that there was a single node (load of $t1a$[*iindex*]) that was connected to another 8 nodes. After applying a few neutral transforms, deletions and a crossover operation, the mutated program reduced the instruction count by 13, and consumed up to 4.3% less energy than the original program.
3. **Cholesky**—Fig. 6 shows the high energy consuming code region found by our energy profiler. In this case, our ASGP algorithm was able to make effective use of merge (a neutral transform) and delete operators to find energy-optimized mutant. The original code was written in a way that prevented the compiler to generate more compact instructions like msub.d (in MIPS). After applying algebraic laws on the original CDDG, the resulting neutral transforms were more conducive to merging the adjacent nodes and forming madd.d. ASGP was able to generate a program mutant that primarily used madd.d instructions, and reduced the overall program energy consumption by slightly over 1%.
4. **Water-sp and Water-n2**—Fig. 7 shows the high energy consuming code regions found by our energy profiler. Our runtime profile data indicated that two branches in this code region (see *sign* macro in the source code line#2) were never taken for all of the available program inputs. This was because the values of a were always positive in all of the input sets. Based on the branch frequency heuristic, the ASGP algorithm generated two separate mutants by deleting the blocks corresponding to the first two branches, and then a crossover from these two mutant versions resulted in an energy optimized code. The overall energy savings in water benchmarks were around 2%.
5. **Fluidanimate**—Fig. 8 shows the high energy consuming code region found by our energy profiler. This loop contains very intensive arithmetic operations. Note that Vec3 is a structure that has three double elements, for which $+=$ and $/=$ have been overridden. The energy reduction came from the initial neutral transform and delete operators. Our ASGP moved the $/=$ around through commutative and distributive rules, which deleted certain expensive (long pipeline cycles) division instructions. Our experiments show up to 4.3% energy savings.
6. **Streamcluster**—Fig. 9 shows the high energy consuming code region found by our energy profiler. Based on the runtime profile data, the ASGP algorithm chose to apply *copy* genetic operator. Even at O3 compiler optimization level, this loop could not be unrolled by the compiler since the range or actual values of iteration count (function parameter *dim* is unknown at compile time). We found that *dim* is a fixed number during program execution. This mutant version improved the energy savings by about 18% since this loop was a prominent kernel in streamcluster benchmark.
7. **Libquantum** Fig. 10 shows the high energy consuming code region found by our energy profiler. In this case, the ASGP chooses to apply the *copy* and *delete* genetic operators based on high branch to non-branch instruction ratio inside the tight loop. We found that the energy-optimized program mutant essentially reduces the ratio of branch code overheads in the loop. This mutant version improves the energy by about 3.8%. Note that at O3 compiler optimization level, this loop in the original program could not be unrolled because the loop iteration count *size* is unknown.
8. **Sphinx3** We found a case similar to Libquantum in Sphinx3. Fig. 11 shows the high energy consuming code region found by our energy profiler. Based on the runtime profile data, the ASGP algorithm applies *copy* to the for loop. This mutant version improves the energy savings by about 1.4%. Note that this loop in the original program was not able to be unrolled by the compiler because the loop iteration count *veclen* is unknown.

We have also tried our best to manually optimize the above code regions independently. For all the cases, we find that the energy saving achieved by the manually optimized program is on par with that of ASGP generated energy-optimized mutant. We note that applications from PARSEC-1.0, Splash-2 and SPEC CPU2006 have been heavily optimized over the years by their developers. Energy savings obtained by ASGP algorithm are already on top of the application developers' optimization.

```
1  //Fmm (Splash−2): interaction.C
2  for (j = 1; j < Expansion_Terms; j++) {
3      temp.r = C[i + j − 1][j − 1];
4      temp.i = (real) 0.0;
5      COMPLEX_MUL(temp, temp, temp_exp[j]);
6      if ((j & 0x1) == 0x0) {
7          COMPLEX_ADD(result_exp, result_exp, temp);
8      } else {
9          COMPLEX_SUB(result_exp, result_exp, temp);
10     }
11 }
```

Fig. 4. Fmm code snippet.

```
1  //Ocean (Splash−2): jacobcalc.C
2  for (iindex=firstcol;iindex<=lastcol;iindex++) {
3      indexp1 = iindex+1;
4      indexm1 = iindex−1;
5      f1 = (t1b[indexm1]+t1d[indexm1]−t1b[indexp1]−t1d[indexp1])
6          ∗(t1f[iindex]−t1a[iindex]);
7      f2 = (t1e[indexm1]+t1b[indexm1]−t1e[indexp1]−t1b[indexp1])
8          ∗(t1a[iindex]−t1g[iindex]);
9      f3 = (t1d[iindex]+t1d[indexp1]−t1e[iindex]−t1e[indexp1])
10         ∗(t1a[indexp1]−t1a[iindex]);
11     f4 = (t1d[indexm1]+t1d[iindex]−t1e[indexm1]−t1e[iindex])
12         ∗(t1a[iindex]−t1a[indexm1]);
13     f5 = (t1d[iindex]−t1b[indexp1])∗(t1f[indexp1]−t1a[iindex]);
14     f6 = (t1b[indexm1]−t1e[iindex])∗(t1a[iindex]−t1g[indexm1]);
15     f7 = (t1b[indexp1]−t1e[iindex])∗(t1g[indexp1]−t1a[iindex]);
16     f8 = (t1d[iindex]−t1b[indexm1])∗(t1a[iindex]−t1f[indexm1]);
17
18     t1c[iindex] = factjacob∗(f1+f2+f3+f4+f5+f6+f7+f8);
19 }
```

Fig. 5. Ocean code snippet.

```
//Cholesky (Splash−2): numLL.C
while (srcNZ0 != last) {
    d0 = *dest0; d1 = *dest1;
    tmp0 = *srcNZ0++; d0 −= ljk0_0*tmp0; d1 −= ljk0_1*tmp0;
    tmp1 = *srcNZ1++; d0 −= ljk1_0*tmp1; d1 −= ljk1_1*tmp1;
    tmp0 = *srcNZ2++; d0 −= ljk2_0*tmp0; d1 −= ljk2_1*tmp0;
    tmp1 = *srcNZ3++; d0 −= ljk3_0*tmp1; d1 −= ljk3_1*tmp1;
    *dest0++ = d0; *dest1++ = d1;
}
```

Fig. 6. Cholesky code snippet.

```
//Water−sp/Water−n2 (Splash−2): cshift.C
#define sign(a,b) (b < 0 ) ? ( (a < 0) ? a : −a) : ( (a < 0) ? −a : a)
for (I = 0; I < 14; I++) {
    /* if the value is greater than the cutoff radius */
    if (fabs(XL[I]) > BOXH) {
        XL[I] = XL[I] −(sign(BOXL,XL[I]));
    }
}
```

Fig. 7. Water-sp/Water-n2 code snippet.

```
//Fluidanimate (Parsec−1): parallel.cpp
for(int iparNeigh = 0; iparNeigh < numNeighPars; ++iparNeigh) {
...
    Vec3 acc = disp * pressureCoeff * (hmr*hmr/dist)
                        * (cell.density[j]+neigh.density[iparNeigh] −
                            doubleRestDensity);
    acc += (neigh.v[iparNeigh] − cell.v[j]) * viscosityCoeff * hmr;
    acc /= cell.density[j] * neigh.density[iparNeigh];
...
}
```

Fig. 8. Fluidanimate code snippet.

```
//Streamcluster (Parsec−1): streamcluster.cpp
float dist(Point p1, Point p2, int dim)
{
    int i;
    float result=0.0;
    for (i=0; i<dim; i++)
        result += (p1.coord[i]−p2.coord[i])*(p1.coord[i]−p2.coord[i]);
    return(result);
}
```

Fig. 9. Streamcluster code snippet.

```
//Libquantum (SPEC CPU2006): gates.c
for(i=0; i<reg−>size; i++)
{
    /* Flip the target bit of a basis state if both control bits are set */
    if(reg−>node[i].state & ((MAX_UNSIGNED) 1 << control1))
    {
        if(reg−>node[i].state & ((MAX_UNSIGNED) 1 << control2))
        {
            reg−>node[i].state ^= ((MAX_UNSIGNED) 1 << target);
        }
    }
}
```

Fig. 10. Libquantum code snippet.

```
//Sphinx3 (SPEC CPU2006): cont_mgau.c
for (i = 0; i < veclen; i++) {
    diff1 = x[i] − m1[i];
    dval1 −= diff1 * diff1 * v1[i];
    diff2 = x[i] − m2[i];
    dval2 −= diff2 * diff2 * v2[i];
}
```

Fig. 11. Sphinx3 code snippet.

5.1. Validation on a real system

By conducting experiments on a real system, we validate the energy-optimized mutant programs generated by our prototype enDebug framework implemented on SESC simulator with McPAT power model. We note that these energy-optimized mutant versions have already passed the fitness test for both functional accuracy and energy optimization. Our goal is to see if similar energy savings can be observed on a real-world system. Our test environment is a Dell Latitude E6520 workstation laptop with 8 GB memory, and 4-core Intel(R) Core(TM) i7-2720QM processor (Sandy Bridge) run at 2.2 GHz with turbo mode at 3.3 GHz using RAPL interface. Each benchmark is run three times to obtain the average energy consumption using the largest input sets (native inputs in PARSEC-1.0, reference inputs in Splash-2 and SPEC CPU2006).

Table 9 shows the energy savings in our real system experiments on two baseline settings: 1. with -O3 optimization level, 2. with both -O3 and GCC's PGO. To enable PGO, we use -fprofile-generate and -fprofile-use flags. Note that -fprofile-generate enables -fprofile-arcs, -fprofile-values and -fvpt flags that include value profile transformations and program flow arcs. The flag -fprofile-use enables -fbranch-probabilities, -funroll-loops, -fpeel-loops, -ftracer that include tracking probabilities of branches and removal of small loops with constant number of iterations. The results show that, at -O3, the energy savings trend is fairly similar to our experiments on SESC (Table 7). With PGO enabled, we find that ASGP is still able to achieve fairly high energy savings (above 5%) in benchmarks such as Fluidanimate, Libquantum, Water-sp and Ocean due to its ability to apply mutations such as copy, deletion, swap and crossover that the compiler normally does not handle. We note that PGO optimizes some additional parts in the program source code based on profile, which makes the total energy consumption of the program a bit smaller than that compiled with just "GCC -O3". So when computing energy reduction percentages, we have smaller denominators which make reduction percentages higher in Ocean, Water-sp, and Water-n2 under the column of "Baseline: GCC -O3 and PGO". In other benchmarks such as streamcluster, the savings are diminished due to the advanced optimization settings in the compiler that can achieve a similar

Table 9
Energy reduction on a real system (Intel Core i7).

Application	Energy reduction (Baseline: GCC -O3)	Energy reduction (Baseline: GCC -O3 and PGO)
Fmm	7.1%	4.6%
Ocean	4.0%	5.7%
Cholesky	N/A	N/A
Water-sp	5.1%	5.7%
Water-n2	2.1%	2.2%
Fluidanimate	10.4%	7.0%
Streamcluster	17.6%	1.6%
Libquantum	6.2%	6.1%
Sphinx3	3.4%	3.2%

outcome as our ASGP algorithm. Overall, we find that our enDebug framework can be extremely useful in finding energy-optimized mutants of program code even after extensive optimizations by the compiler.

6. Related work

We discuss prior works in two broad categories—energy estimation and optimization.

6.1. Energy estimation

Isci et al. [21] propose runtime power monitoring techniques for processor core and functional units. Some recent works demonstrate the feasibility of using a limited set of metrics to estimate processor component power [33,22,4]. These works do not explicitly address how to attribute energy back to the program code. In contrast to these prior schemes, we investigate ways to attribute energy to fine-grained code.

Tiwari et al. [44] developed an energy model using instruction counts, and assume a pre-determined cost for various instruction types. This ignores dynamic hardware effects such as parallelism and interference that occur in most modern architectures. Alternative strategies that use a specific set of hardware events (such as cache misses) [1,42] for energy estimation often fail to include a comprehensive view of application execution and ignore system-level effects and interactions with other instructions (e.g., pipeline stalls, pipeline flushes due to mispredicted instructions). In contrast to these prior approaches, our methodology uses largely existing hardware (with simple modifications) to accurately gather the total energy consumed by fine-grained sections of code (functions).

6.2. Energy optimization

Microarchitecture—Hardware modifications have been proposed to minimize energy consumption by the microarchitectural units. For instance, to reduce cache energy, prior works have investigated reconfigurable cache with variable associativity or number of banks [52,32], bypassing expensive tag check operations [48], trading cache capacity for low supply voltage [47], and reducing redundant writes to the memory [7]. For issue logic and load/store queues that rely on energy-expensive structures such as Content Address Memory (CAM), researchers have studied dynamically adjusting issue or load/store queue sizes and avoiding wasteful wake-up checks [13,6,23]. Recent works have started investigating custom hardware accelerators for specific types of applications [46,16,51]. Prior works also consider optimizing processor pipelines for low power and energy [12,39,18].

Dynamic voltage and frequency scaling (DVFS)—Prior works have shown how to exploit slack time in the running thread to put the processor core in a sleep state [30], or run at a low voltage/frequency level [35,2]. DVFS has also been demonstrated to do system-level power and energy management [27]. Since our approach is complementary to this line of work, we may gain even further energy savings by applying DVFS to our energy-optimized program mutants.

Power/energy aware compilation—Compiler techniques have been studied for energy-aware instruction scheduling and code generation. [44,38] have studied the energy impact of instruction scheduling by using instruction-level energy cost based on electric current measurements. In modern day processors with heavily overlapped instructions and shared functional units, this type of per-instruction energy accounting is difficult, if not impossible. Tuning voltage and frequency settings using generated program code to reduce program energy have also been studied by [19,31,50]. [53,26] have studied the energy impact of loop centric optimization techniques such as loop permutation, loop tiling, and loop fusion. Our enDebug framework optimizes energy on compiler generated code and provides further energy reduction.

Automated Code Optimization—As software inefficiency increasing becomes the dominant source of wasteful energy consumption, there is a strong urge to develop energy-aware program code. Chen et al. [10] explored hardware–software support for pinpointing the code and root causes for high power peaks while program is running. Recently, Schulte et al. [40] proposed a post-compiler software optimization technique. In this work, authors designed a genetic optimization algorithm that exhaustively searches for program variant with functional correctness and lesser energy consumption. Although both of our works broadly use genetic programming for mutating program code, our enDebug differs from [40] in the following ways: 1. enDebug incorporates a fine-grained energy profiler in hardware that estimates energy for program functions to apply targeted optimizations, while [40] uses an energy model that estimates energy at the process level. 2. enDebug applies mutant operations only on the most energy consuming code region, while [40] approach variates the entire program, which may make the search space become prohibitively expensive for large scale programs 3. enDebug adopts a guided (artificial selection) approach by taking advantage of heuristics derived from program code structure or runtime profile, while [40] selects mutation operators randomly which is not performance friendly as shown in Table 6.

7. Conclusions

In this paper, we showed the need to understand application energy profile. We show the design of our solution approach, enDebug, that has two major components—1. energy profiler, and 2. automated recommendation system for program energy optimization.

We designed an energy profiler using largely existing hardware support that measures the energy of program functions. We explored an automated recommendation system that incorporates artificial selection genetic programming algorithm to generate program mutants that minimize energy consumption. In contrast to prior approaches, we adopt a guided approach to mutant generation that reduces the search space and quickens the time

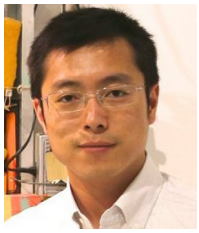
(4×–41×) to arrive at energy-optimized program mutants. We show case studies from several Splash-2, PARSEC-1.0 and SPEC CPU2006 benchmarks, and demonstrate energy savings of up to 7% beyond the highest compiler optimization settings when tested on real-world Intel Core i7 processors.

Acknowledgment

This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557.

References

[1] F. Bellosa, The benefits of event: Driven energy accounting in power-sensitive systems, in: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, EW 9, 2000.
[2] A. Bhattacharjee, M. Martonosi, Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors, in: Proceedings of ISCA, 2009.
[3] C. Bienia, S. Kumar, J. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, Princeton University Technical Report TR-811-08, 2008.
[4] W. Bircher, L. John, Complete system power estimation using processor performance events, IEEE Comput. (2012).
[5] M. Brameier, W. Banzhaf, Linear Genetic Programming, Springer, 2007.
[6] A. Buyuktosunoglu, T. Karkhanis, D.H. Albonesi, P. Bose, Energy efficient co-adaptive instruction fetch and issue, in: Proceedings of ISCA, 2003.
[7] J. Chen, R.C. Chiang, H.H. Huang, G. Venkataramani, Energy-aware writes to non-volatile main memory, SIGOPS Oper. Syst. Rev. 45 (3) (2012) 48–52. http://dx.doi.org/10.1145/2094091.2094104. URL: http://doi.acm.org/10.1145/2094091.2094104.
[8] J. Chen, G. Venkataramani, A hardware–software cooperative approach for application energy profiling, Comput. Archit. Lett. 14 (1) (2015) 5–8. http://dx.doi.org/10.1109/LCA.2014.2323711.
[9] J. Chen, G. Venkataramani, G. Parmer, The need for power debugging in the multi-core environment, IEEE Comput. Archit. Lett. (2012).
[10] J. Chen, F. Yao, G. Venkataramani, Watts-inside: A hardware–software cooperative approach for multicore power debugging, in: 2013 IEEE 31st International Conference on Computer Design, ICCD, 2013, pp. 335–342. http://dx.doi.org/10.1109/ICCD.2013.6657062.
[11] eBay Inc., Digital service efficiency. http://dse.ebay.com/.
[12] D. Ernst, N.S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, T. Mudge, Razor: A low-power pipeline based on circuit-level timing speculation, in: Proceedings of MICRO, 2003.
[13] D. Folegnani, A. González, Energy-effective issue logic, in: Proceedings of ISCA, 2001.
[14] Free Software Foundation, Inc, GCC, the GNU compiler collection. http://gcc.gnu.org.
[15] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, SIGPLAN Not. 40 (6) (2005) 213–223. http://dx.doi.org/10.1145/1064978.1065036. URL: http://doi.acm.org/10.1145/1064978.1065036.
[16] V. Govindaraju, C.H. Ho, K. Sankaralingam, Dynamically specialized datapaths for energy efficient computing, in: Proceedings of HPCA, HPCA'11, 2011.
[17] S. Hao, D. Li, W.G.J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, in: Proceedings of ICSE, 2013.
[18] M. Hayenga, V. Reddy, M.H. Lipasti, Revolver: Processor architecture for power efficient loop execution, in: Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, 2014.
[19] C.-H. Hsu, U. Kremer, The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction, in: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI'03, ACM, New York, NY, USA, 2003, pp. 38–48. http://dx.doi.org/10.1145/781131.781137. URL: http://doi.acm.org/10.1145/781131.781137.
[20] Intel Corporation, Intel® 64 and IA-32 Architectures Optimization Reference Manual, no. 248966-018, 2009.
[21] C. Isci, G. Contreras, M. Martonosi, Live, runtime phase monitoring and prediction on real systems with application to dynamic power management, in: Proceedings of MICRO, 2006.
[22] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, R. Eickemeyer, Abstraction and microarchitecture scaling in early-stage power modeling, in: Proceedings of HPCA, 2011.
[23] T.M. Jones, M.F.P. O'Boyle, J. Abella, A. Gonzalez, Software directed issue queue power reduction, in: Proceedings of HPCA, 2005.
[24] R. Jotwani, S. Sundaram, S. Kosonocky, A. Schaefer, V. Andrade, G. Constant, A. Novak, S. Naffziger, An x86-64 core implemented in 32nm soi cmos, in: Proceedings of ISSCC, 2010.
[25] N.P. Jouppi, et al. Cacti 5.1. http://quid.hpl.hp.com:9081/cacti/.
[26] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, W. Ye, Influence of compiler optimizations on system power, IEEE Trans. Very Large Scale Integr. Syst. 9 (6) (2001) 801–804. http://dx.doi.org/10.1109/92.974893.
[27] W. Kim, M. Gupta, G.-Y. Wei, D. Brooks, System level analysis of fast, per-core dvfs using on-chip switching regulators, in: IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008, 2008.
[28] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence—Vol. 2, IJCAI'95, 1995.
[29] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, N.P. Jouppi, Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures, in: MICRO, 2009.
[30] J. Li, J.F. Martinez, M.C. Huang, The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors, in: Proceedings of HPCA, 2004.
[31] G. Magklis, M.L. Scott, G. Semeraro, D.H. Albonesi, S. Dropsho, Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor, in: Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA'03, ACM, New York, NY, USA, 2003, pp. 14–27. http://dx.doi.org/10.1145/859618.859621. URL: http://doi.acm.org/10.1145/859618.859621.
[32] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, K. Roy, Reducing set-associative cache energy via way-prediction and selective direct-mapping, in: Proceedings of MICRO, 2001.
[33] M.D. Powell, A. Biswas, J. Emer, S. Mukherjee, B. Sheikh, S. Yardi, Camp: A technique to estimate per-structure power at run-time using a few simple parameters, in: Proceedings of HPCA, 2009.
[34] A. Rallo, Data center efficiency trends for 2014. http://www.energymanagertoday.com/data-center-efficiency-trends-for-2014-097779/.
[35] K.K. Rangan, G. Wei, D. Brooks, Thread motion: fine-grained power management for multi-core systems, in: Proceedings of ISCA, 2009.
[36] J. Renau, et al. SESC. http://sesc.sourceforge.net.
[37] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, E. Weissmann, Power-management architecture of the intel microarchitecture code-named sandy bridge, IEEE Micro (2012).
[38] J. Russell, M. Jacome, Software power estimation and optimization for high performance, 32-bit embedded processors, in: International Conference on Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings, 1998, pp. 328–333.
[39] J. Sartori, B. Ahrens, R. Kumar, Power balanced pipelines, in: Proceedings of HPCA, 2012.
[40] E. Schulte, J. Dorn, S. Harding, S. Forrest, W. Weimer, Post-compiler software optimization for reducing energy, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.
[41] K. Sen, D. Marinov, G. Agha, Cute: A concolic unit testing engine for c, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, ACM, New York, NY, USA, 2005, pp. 263–272. http://dx.doi.org/10.1145/1081706.1081750. URL: http://doi.acm.org/10.1145/1081706.1081750.
[42] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, Z. Chen, Power containers: An os facility for fine-grained power and energy management on multicore servers, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'13, 2013.
[43] Standard Performance Evaluation Corporation, SPEC Benchmarks. http://www.spec.org.
[44] V. Tiwari, S. Malik, A. Wolfe, M.T. Lee, Instruction level power analysis and optimization of software, J. VLSI Signal Process. Syst. 13 (2–3) (1996).
[45] J. Treibig, G. Hager, G. Wellein, Likwid: A lightweight performance-oriented tool suite for x86 multicore environments, in: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 207–216. http://dx.doi.org/10.1109/ICPPW.2010.38.
[46] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, M.B. Taylor, Conservation cores: reducing the energy of mature computations, in: Proceedings of ASPLOS, 2010.
[47] C. Wilkerson, H. Gao, A.R. Alameldeen, Z. Chishti, M. Khellah, S. Lu, Trading off cache capacity for low-voltage operation, IEEE Micro (2009).
[48] E. Witchel, C.S. Larsen, S. Ananian, K. Asanović, Direct addressed caches for reduced power consumption, in: Proceedings of MICRO, 2001.
[49] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The splash-2 programs: characterization and methodological considerations, in: Proceedings of ISCA, 1995.
[50] Q. Wu, M. Martonosi, D.W. Clark, V.J. Reddi, D. Connors, Y. Wu, J. Lee, D. Brooks, A dynamic compilation framework for controlling microprocessor energy and performance, in: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 38, IEEE Computer Society, Washington, DC, USA, 2005, pp. 271–282. http://dx.doi.org/10.1109/MICRO.2005.7.
[51] V.J.R. Yuhazo Zhu, Webcore: Architectural support for mobile web browsing, in: Proc. of International Symposium on Computer Architecture., 2014.
[52] C. Zhang, F. Vahid, W. Najjar, A highly configurable cache architecture for embedded systems, in: Proceedings of ISCA, 2003.
[53] Y. Zhu, G. Magklis, M.L. Scott, C. Ding, D.H. Albonesi, The energy impact of aggressive loop fusion, in: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT'04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 153–164. http://dx.doi.org/10.1109/PACT.2004.28.



Jie Chen is a Senior Performance Engineer at MathWorks, Inc. He received his Ph.D. in Computer Engineering from George Washington University in 2015. His general research interests are in the areas of computer architecture, hardware systems, and software engineering. He is especially interested in hardware and software support for improving application performance, power efficiency, memory system reliability, and solutions for mitigating hardware security vulnerabilities. He received Best Poster award in IEEE/ACM PACT 2011, and has an invited article in ACM OS Review as an author of Best of HotPower 2011 workshop.

Guru Venkataramani is an Associate Professor of Electrical and Computer Engineering at George Washington University since 2009. He received his Ph.D. from Georgia Institute of Technology, Atlanta in 2009. His research area is computer architecture, and his current interests are hardware support for energy/power optimization, debugging and security. He is a recipient of NSF Faculty Early Career award in 2012, Best Poster award in IEEE/ACM PACT 2011, has an invited article in ACM OS Review as an author of Best of Hotpower 2011 workshop, and ORAU Ralph E. Powe Junior Faculty Enhancement Award in 2010. He is a senior member of both IEEE and ACM.