Data-Oriented Characterization of Application-Level Energy Optimization

Kenan Liu[1], Gustavo Pinto[2], and Yu David Liu[1]

[1] State University of New York, Binghamton, NY, US
{kliu20, davidL}@cs.binghamton.edu
[2] Federal University of Pernambuco, Recife, PE, Brazil
ghlp@cin.ufpe.br

Abstract. Empowering application programmers to make energy-aware decisions is a critical dimension in improving energy efficiency of computer systems. Despite the growing interest in designing software development processes, frameworks, and programming models to facilitate application-level energy management, little is empirically known on how *application-level features* impact energy management. In this paper, we illuminate the optimization space of application-level energy management, from a novel *data-oriented* perspective. First, we study the varying energy impacts of alternative data management choices by programmers, such as data access patterns, data precision choices, and data organization. Second, we attempt to build a bridge between application-level energy management and hardware-level energy management, by elucidating how various application-level features respond to Dynamic Voltage and Frequency Scaling (DVFS), arguably the most classic hardware-based energy management approach. Finally, we apply our findings to real-world applications, demonstrating their potential for greater energy savings. The empirical study is particularly relevant in the Big Data era, where data-intensive applications are large energy consumers, and their energy efficiency is strongly correlated to how data are maintained and handled in programs.

Keywords: Energy consumption, Application-level data management

1 Introduction

Modern computing platforms are experiencing an unprecedented diversification. Beneath the popularity of the Internet of Things, Android phones, Apple iWatch and Unmanned Aerial Vehicles, a critical looming concern is energy consumption. Traditionally addressed by hardware-level (*e.g.*, [13,6]) and system-level approaches (*e.g.*, [8,21]), energy optimization gains momentum in recent years by focusing on application development [4,5,16]. These *application-level energy management* strategies complement lower-level strategies with an expanded *optimization space*, yielding distinctive advantages: first, applications are viewed as a white box, whose structural features may be considered for energy optimization; second, the knowledge of programmers and their design choices can influence energy efficiency. Recent studies [19] show application-level energy management is in high demand among application developers.

The grand challenge ahead is the lack of systematic guidelines for application-level energy management. Unlike lower-level energy management strategies that often happen "under the hood," application-level energy management requires the participation of application software developers. For example, programmers need to understand the energy behaviors at different levels of software granularities in order to make judicious design decisions, and thus improve the energy efficiency. As indicated in recent studies, the devil

often lies with the details [3,20], and the guidelines are often anecdotal or incorrect [19]. Should we pessimistically accept that the optimization space of application-level energy management as unchartable waters, or is there wisdom we can generalize and share with application developers in their energy-aware software development?

This paper is aimed at exploring this important yet largely uncharted optimization space. Even though the energy impact of arbitrary developer decisions — *e.g.*, using encryptions when the battery level is high and no security otherwise — is impossible to generalize and quantify, we believe a sub-category of such design decisions — those related to *data* — have interesting and generalizable correlations with energy consumption. With Big Data applications on the rise, we believe the data-oriented perspective on studying application-level energy management may in addition have the forward-looking appeal on future energy-aware software development. In particular, we attempt to answer the following research questions:

RQ1 How does the choice of application-level features impact energy consumption?
RQ2 How does application-level energy management interact with hardware-level energy management?

For **RQ1**, we consciously look into features "middle-of-the-road" in granularity: they are coarser-grained than instructions [26] or bytecode [12,17] to help retain the high-level intentions of application developers, yet at the same time finer-grained than software architectures or frameworks to facilitate reliable quantification. Specifically, we study the impact of energy consumption over different choices of:

- *data access pattern*: For a large amount of data, does the pattern of access (sequential vs. random, read vs. write) impact energy consumption?
- *data organization and representation*: For different representations of the same data (unboxed vs. boxed data, primitive arrays vs. array lists) have impact on energy consumption?
- *data precision*: Do precision levels (short, integer, floating points, double, long) of data have significant impact on energy consumption?
- *data I/O strategies*: For I/O-intensive applications, do different choices of buffering and different levels of data intensity have impact on energy consumption?

To answer **RQ2**, we are aimed at connecting application-level energy management and its lower-level counterparts. It is our belief that energy consumption is the combined effect of interactions through application software, system software, and hardware; the best energy management strategy should be the harmonious coordination of all layers of the compute stack. Concretely, we reinvestigate the aforementioned data-oriented application features in the context of Dynamic Voltage and Frequency Scaling [13] (DVFS), arguably the most classic hardware-based energy management strategy. For instance, when the CPU operational frequency reduces from 2.6Ghz to 1.2Ghz, does it have *proportional* and *identical* impact on energy/performance of two programs, one with sequential access and the other with random access (or one with double precision and the other with integer precision)? The answer to this question explores the expanded optimization space where "software meets hardware, " over a frontier where software engineering research joins forces with hardware architecture research.

The paper makes the following contributions:

- It performs the first empirical study that systematically characterizes the optimization space of application-level energy management, from the fresh perspective of focusing on data. The energy optimization space is explored through multiple dimensions, ranging

from data access pattern, data organization and representation, data precision, and data I/Os intensity.
- It conducts a set of holistic experiments aimed at bridging application-level and hardware-level energy management, and constructing a unified optimization space connecting hardware and application software.
- It reports the release of `jRAPL`, an open-source library to precisely and non-invasively gather energy/performance information of Java programs running on Intel CPUs.

2 Methodology

In this section, we introduce our research methodology and the details of our experimental environment.

2.1 The Open-Source `jRAPL` Library

We have developed a set of APIs for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) [6] support. Originally designed by Intel for enabling chip-level power management, RAPL is widely supported in today's Intel architectures, including Xeon server-level CPUs and the popular `i5` and `i7`. RAPL-enabled architectures monitors the energy consumption information and stores it in Machine-Specific Registers (MSRs). Such MSRs can be accessed by OS, such as the `msr` kernel module in Linux. RAPL is an appealing design, particularly because it allows energy/power consumption to be reported at a fine-grained manner, *e.g.*, monitoring CPU core, CPU uncore (caches, on-chip GPUs, and interconnects), and DRAM separately.

Our library can be viewed as a software wrapper to access the MSRs. The RAPL interface itself has broader support for energy management, whereas our library only uses its capability for information gathering, a mode in RAPL named "energy metering." Since the `msr` module under Linux runs in privileged kernel mode, `jRAPL` works in a similar manner as system calls.

The user interface for `jRAPL` is simple. For any block of code in the application whose energy/performance information is to the interest of the user, she simply needs to enclose the code block with a pair of `statCheck` invocations. For example, the following code snippet attempts to measure the energy consumption of the `doWork` method, whose value is the difference between `beginning` and `end`:

```
double beginning = EnergyCheck.statCheck();
doWork();
double end = EnergyCheck.statCheck();
```

Additional APIs also allow time and other lower-level hardware performance counter information (for diagnostics) to be collected. The API can flexibly collect either CPU time, User Mode time, Kernel Mode time, and Wall Clock time. If not explicitly specified, all time reported in the paper is wall clock time. When the CPU consists of multiple cores, `jRAPL` can report data either individually or combined. Throughout the paper, all energy/power data for multi-core CPUs are reported as combined.

Compared with traditional approaches based on physical energy meters, the `jRAPL`-based approach comes with several unique advantages:

- *Refined Energy Analysis*: thanks to RAPL, our library can not only report the overall energy consumption of the program, but also the breakdown (1) among hardware components and (2) among program components (such as methods and code blocks).

As we shall see, refined hardware-based analysis allows us to understand the relative activeness of different hardware components, ultimately playing an important role in analyzing the energy behaviors of programs. In meter-based approaches, hardware design constraints often make it impossible to measure a particular hardware component (such as CPU cores only, or even DRAMs because they often share the power supply cables with the motherboard).

– *Synchronization-Free Measurement*: in meter-based measurements, a somewhat thorny issue is to synchronize the beginning/end of measurement with the beginning/end of the execution of interest. This problem would be magnified if one considers fine-grained code-block based measurement, where the problem *de facto* becomes the synchronization of measurement and the program counter. With `jRAPL`, the demarcation of measurement coincides with that of execution; no synchronization is needed.

One drawback of the `jRAPL`-based approach is the energy data collection itself may incur overhead. Fortunately, the time overhead for MSR access is in the microseconds, magnitudes lower than the execution time of our experiments.

2.2 Experimental Environment

We run each experiment in the following machine: a 2×8-core (32-cores when hyper-threading is enabled) Intel(R) Xeon(R) E5-2670, 2.60GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2 and L3) with 64KB per core (128KB total), 256KB per core (512KB total) and 3MB (smart cache), respectively. It is running Debian 6 (kernel 3.0.0-1-amd64) and Oracle HotSpot 64-Bit Server VM (build 20.1-b02, mixed mode), JDK version 1.6.0_26. The processor has the capability of running at several frequency levels, varying from 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4 and 2.6 GHz. Due to architecture design, the RAPL support can access CPU core, CPU uncore in addition access DRAM energy consumption data.

For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled to be realistic with real-world Java applications. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 6 times within the same JVM; this is implemented by a top-level 6-iteration loop over each benchmark. The reported data is the average of the last 4 runs. If the standard deviation of such 4 runs is greater than 5%, we executed the benchmark again until results stabilize. All experiments were performed with no other load on the OS. Unless explicitly specified in the paper, the default `ondemand` governor of Linux is used for OS power management. Turbo Boost feature is disabled.

3 Application-Level Energy Management

This section explores the optimization space of application-level energy management through four data-oriented characterizations.

3.1 Data Access Patterns

We first examine how energy consumption differs under sequential and random access. By access, we consider both read and write operations. The read micro-benchmark traverses a large array (of size N=50,000,000), retrieving the value at each position, and its write counterpart assigns integer 1 to each position. To construct a fair comparison between sequential and random access, we resort to an "index array" preloaded with index numbers:

numbers from 1 to N in that order for sequential access, and a random permutation of numbers between 1 and N for random access. Thanks to the index array, the program logic is identical for sequential and random access. The reported energy/performance results do not consider preloading.

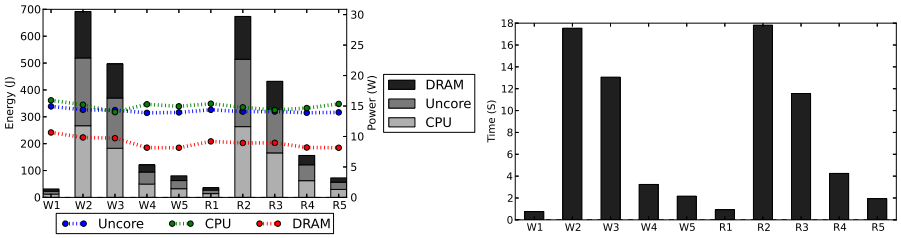


Fig. 1. Energy/Performance behaviors under different data access patterns

Figure 1 shows the benchmarking results, with energy (bars) and power (lines) data in the left figure, and time in the right figure. There are 10 bars for each figure, indicating the combination of sequential vs. random access and read vs. write access. The read/write accesses are differentiated by label prefix `R` and `W`, respectively. The suffix 1 represents sequential access, 2 for 100% random access, 3 for 1% random access, and 4 for 0.1% random access. The level of randomness is controlled by index range: For instance, we imitate 1% random access by allowing random permutation within each N $\times$ 1% interval of the array. We now discuss several findings of interest.

First, the most obvious observation is that random access consumes much more energy than its sequential counterpart. For example, 100% random write consumes 11.31x more energy than sequential write (`W2` and `W1`), and the same ratio for read is 10.34x (`R2` and `R1`). The root cause of this phenomenon is cache locality. In the random scheme, neither the hardware nor the runtime (*e.g.*, through pre-fetching or bulking optimization) is likely to be effective to reduce cache misses. Indeed, the execution time of random vs. sequential access (the figure on the right) show a similar trend, an unsurprising fact.

Second, energy and performance are not proportional for read vs. write — otherwise, the bars in the right figure would become a (boring) predictor of the bars in the left. By physics, energy is the multiplication of power and time. Thus, if the proportionality between energy and performance were to hold, power consumption would be a constant. Interestingly, read and write operations have consume roughly the same amount of energy. We believe this can be explained in terms of LOAD and STORE hardware instructions, which have similar overhead. Primitive arrays do not impose aditional significant overhead.

Third, DRAM power consumption appears to remain rather stable. This trend appears to hold for nearly all experiments we conducted for this research. This may be good news for systems where DRAM data are not available for its RAPL interface. We can estimate it based on execution time (and the roughly constant power consumption).

3.2 Data Representation Strategies

Let us now investigate the impact of different data representation strategies on energy consumption. First, we look into the difference between representing a sequence of integers as primitive arrays and `ArrayList`. We construct a similar experiment as one described in Section 3.1, by traversing the two data structures of a large size (N = 50,000,000). In

the `ArrayList` implementation, we mimic "read" through the `List.get(int i)` method, and "write" through the `List.set(int i, Object o)` method.

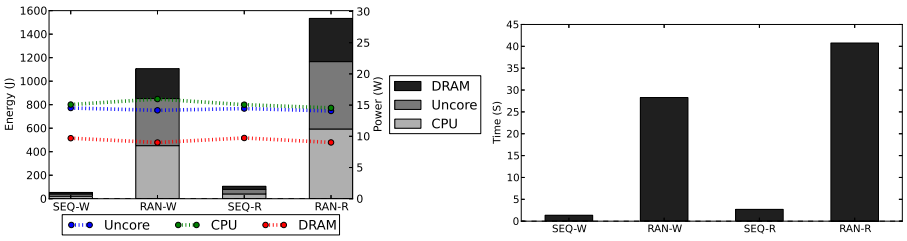


Fig. 2. Energy/Performance behavior of `ArrayList` representation

The energy/performance results of the `ArrayList` implementation are shown in Figure 2, where `SEQ/RAN/R/W` labels denote sequential, random, read, and write access, respectively. Compared with Figure 1, energy consumption is significantly higher: the `RAN-R` configuration with primitive array representation consumes around 670J, whereas its counterpart result here is around 1550J. This does not come as a surprise. After all, the getter/setter required by `ArrayList` are both method invocations, significantly more expensive than primitive array read/write. Other than this, most conclusions we drew in the previous subsection applies to the `ArrayList` implementation as well, in terms of both the relative standings between sequential vs. random access and the relative standings between read vs. write access.

Observant readers may find `ArrayList` uses boxed data (of `Integer` type) whereas our primitive array implementation uses unboxed data (of `int` type). Does data boxing/unboxing have significant impact on energy consumption? The more general question here is what representation of an object is being accessed: a reference to it, or a value it holds. We construct the next set of experiments to answer this question. Here, our experiments are divided in three groups:

- *Reference Query:* accesses the references of `Integer` objects;
- *Value Query:* accesses the value that `Integer` objects hold;
- *Type Query:* accesses the type held by `Integer` objects;

Figure 3 shows the experimental results when we perform three queries over a large array of `Integers`. The Reference Query, Value Query and Type Query are labeled, respectively, as RQ, VQ and TQ. Postfix 1 denotes sequential access, 2 denotes 100% random access, and 3 denotes 25% random access. Reference Query is the most efficient in both energy and time, consuming similar amount of energy when compared to its relative – the Type Query strategy. However, both approaches consume less energy than Value Query. Reference Query consumes 6.61x less energy, while Type Query consumes 10.19x less energy. Similar pattern is also observed in the time figure (References Query performs 5.03x faster than Value Query). Also, since Value Query

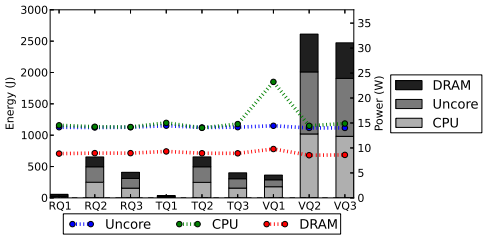


Fig. 3. Energy behavior: reference query, value query, and type query.

is mostly a CPU-intensive operation, we observed an increase of 8.19x in the CPU energy consumption when compared to Reference Query.

3.3 Data Organization

In the next experiment, we consider two programs in Figure 4 and Figure 5. Functional equivalent, the first *object-centric* program accesses a large array of objects with 5 fields, and the second *attribute-centric* program accesses 5 primitive arrays.

```
class Grouped {
  int a, b, c, d, e = ...;
}
class Main {
  Grouped[] group = ...;
  void calc() {
    for (int i = 0; i < N; i++) {
      group[i].e = group[i].a * group[i].b * group[i].c * group[i].d;
}}}
```

Fig. 4. Object-Centric Data Grouping

```
class Main {
  int[] a = ..; int[] b = ..; int[] c = ..; int[] d = ..; int[] e = ..;
  void calc() {
    for (int i = 0; i < N; i++) {
      e[i] = a[i] * b[i] * c[i] * d[i];
}}}
```

Fig. 5. Attribute-Centric Data Grouping

As shown in Figure 6, the object-centric data grouping consumes about 2.62x energy. The results here may reveal a trade-off between programming productivity and energy efficiency. Object-oriented encapsulation is known to have many benefits, such as modularity, information hiding, and maintainability. That being said, it does pay a toll on energy consumption, likely due to heap objects are allocated in non-contiguous space.

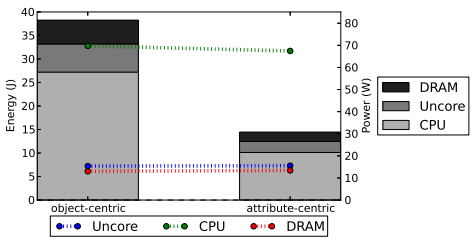


Fig. 6. Energy behavior under different data organizations.

3.4 Data Precision Choices

In this section, we analyze the energy consumption of different primitive data types. We start with a matrix multiplication using `int`, and then we compare this implementation with four other variations by replacing the array element type from `int` to `short`, `float`, `double`, and `long` types respectively. For our environment, `long`/`double`/`float`/`int`/`short` data types are 64/64/32/32/16 bits respectively.

Figure 7 shows the `int`-based matrix consumes 3.39x more energy than its `short` counterpart. The matrix of `double` data type consumes 1.45x energy than that of the `int` type, and 4.95x energy than `short`. We believe the difference results from the behaviors of the cache and the FPU. Values of a larger size require more space to store in the cache and also require more memory bandwidth. Both `float` and `double` entail increased activities in the FPU unit, which in turn lead to increased energy consumption. Our results show that, if programmers have a precise knowledge of the data range and the precision of output result, significant energy savings and performance gains are possible, *e.g.*, replacing `int` with `short`, or `double` with `float` for data types.

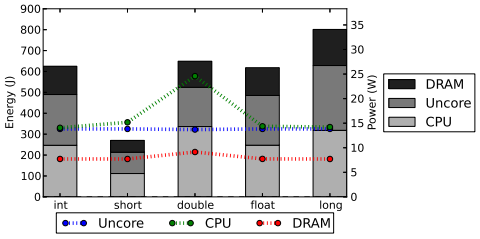


Fig. 7. Energy behavior under different data precision choices

3.5 Data I/O Configurations

In this section, we analyze the energy behaviors of I/O operations. We use two benchmarks that read and write to a file using `FileInputStream` and `FileOutputStream` objects, respectively. These operations are buffered using the `BufferedOutputStream` and `BufferedInputStream` objects. We also use two other variations of these benchmarks, in which read and write are not buffered. This modification prevents data buffering and hence causes additional system calls (one call for each written byte).

Figure 8 shows how energy consumption behaves under different I/O operations. First, buffering has significant impact on improving energy efficiency. In all cases, the energy consumption for buffered I/O is small/negligible compared with their unbuffered counterpart (for instance, write with buffer consumes 5.92x less energy than without buffer). Indeed, buffer removal in essence disables bulking of I/O operations, so its effect on energy consumption is dramatic. We then believe that buffering write and read operations should be mandatory, when possible. Second, data output is significantly more energy-consuming than data input. For unbuffered I/O, energy consumption between data output and data input is near 5x. Third, contrary to the popular belief that CPU/memory would be set to lower power state when I/O is being performed, power consumption for both CPU core, CPU Uncore, and DRAM remains on par with that from experiments in earlier sections. This implies Linux's default power management strategy — the `ondemand` governor — may not be aggressive enough. It may indeed be said that for the buffered experiments, the execution time may be short enough so that the `ondemand` governor does not have sufficient time to kick in, but it is baffling that for unbuffered I/O where the execution time can reach as long as 100 seconds, the CPU/memory is not placed to much lower power state.

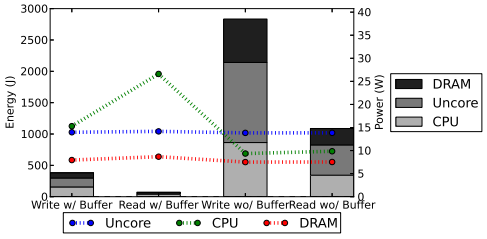


Fig. 8. Energy/Performance behaviors of data I/O operations

We also explored two additional I/O mechanisms: the first one – a lightweight I/O operation, which we call `stdout` – constitutes a simple invocation to the `System.out.println` method. When the `println()` method is invoked, the `flush()` method is automatically

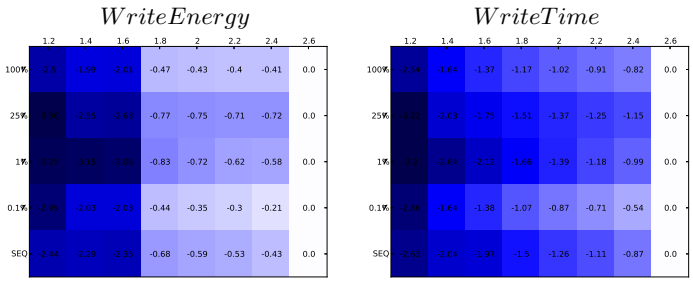


Fig. 10. DVFS and (Write) Access Patterns. (Labels on top are CPU frequencies, and labels to the left are random/sequential access patterns, with the same convention as in Fig. 1. All data are normalized against the 2.6Ghz data of the same row. Red indicates savings, whereas Blue indicates loss. The darker the Red shade, the more "favorable" the configuration is, *i.e.*, greater energy/time savings. The darker the Blue shade, the more "unfavorable" the configuration is *i.e.*, greater energy/time loss.)

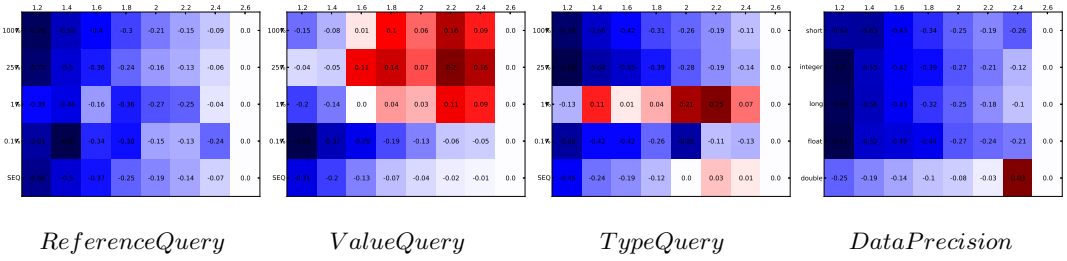


Fig. 11. DVFS and Data Representation/Precision. Only energy results are shown.

invoked and byte array is written in the standard output channel. The value of N for this benchmark is 500,000. The other benchmark – which we call `socket` – implements a basic socket operation. In the `socket` benchmark, we are only measuring the energy consumption in the receiver side. The input data of this benchmark comprises 3 binary files with sizes of 1.1GB, 536.9MB and 209.7MB, which are sent in a row. No concurrency is involved.

There are a number of interesting findings from the results, shown in Figure 9. First, different workloads have different impacts. The `socket` implementation consumes 17.15x energy than the `stdout` configuration. That being said, energy consumption in the `stdout` benchmark is not negligible either, at about 379J. Programmers should take this result in consideration, for instance, when deciding whether to log every event in their applications. We also observed the lowest CPU power consumption in the `socket` benchmark. There, CPU is almost entirely neglected, which enables the processor to go to the idle state and save power.

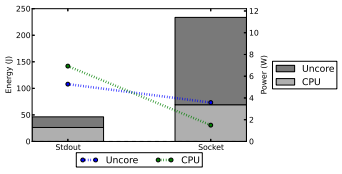
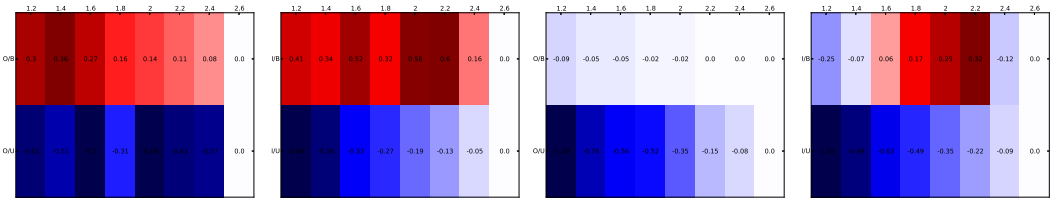


Fig. 9. Energy/Performance behaviors of different I/O workloads



output energy *input energy* *output time* *input time*

Fig. 12. DVFS and Data I/O (`O`: Output, `I`: Input, `B`: Buffered, `U`: Unbuffered). Only energy results are shown.

4 Unifying Application-Level Optimization with DVFS

This section places application-level energy management in a broader context, investigating its combined impact with hardware-based energy management.

Background DVFS [13] is a common CPU feature where the operational frequency and the supply voltage of the CPU can be dynamically adjusted. DVFS is a classic and effective power management strategy. The dynamic power consumption of a CPU, denoted as P, can be computed as $P = C * V^2 * F$, where V is the voltage, F is the frequency, and C is a (near constant) factor. The energy consumption E is an accumulation of power consumption over time t, *i.e.*, through formula $E = P * t$.

Scaling down the CPU frequency is effective in saving power. Saving energy, however, is more complex because a reduction of frequency may increase the execution time. Thus, DVFS energy management often deals with the trade-off between energy consumption and performance.

Result Summary Figure 10, Figure 11, and Figure 12 report selected results from the same experiments constructed in the previous section, except that the executions are conducted at different CPU frequencies. Due to page limit, we defer the complete data set in the online repository (See Sec. 8 for information). All figures are represented as heat map matrices.

A common trend among these experiments is that downscaling CPU often leads to less "favorable" results: in the majority of experiments, not only there will be a performance loss, but also increased energy consumption. The root cause is that DVFS only directly influences the CPU power consumption. The power consumption for the Uncore and the DRAM sub-systems remain roughly constant. Thus, since time increases as frequency decreases, energy consumption for these sub-systems increases as well when a lower CPU frequency is selected. For the CPU core sub-system, the effect on energy consumption in the presence of downscaling depends on whether the decrease in power (P) may offset the increase in time (t). This is a sober reminder of the applicability of DVFS as an energy management strategy: whileas downscaling can be effective in some scenarios, it blind DVFS is likely to fall short in goals. As a symptom, observe that the use of the lowest frequency in most cases consumes the *most* energy.

Indeed, our discussion above adopted a very narrow notion to define what is "favorable." For instance, running CPUs at the lowest frequency may reduce heat dissipation, and improve the reliability of program execution.

Overall, our results can serve as a "look up" chart to guide energy-aware programmers to desirable combinations of application-level energy management and hardware-level energy management. For example, if a programmer wishes to execute the matrix manipulation program in Figure 7 with a budget of 105J, she can look up the results from Figure 11, either choosing to run with `double` precision at 1.2GHz, or with `int` precision at 2.2GHz. The two configurations have distinct advantages: the 1.2GHz/`double` execution may reduce heat dissipation, whereas the 2.2GHz/`int` may produce results faster.

Specific Findings In Figure 10, the most interesting results is perhaps 100% random access or 25% random access. Here, executing the program at frequencies of 2.4GHz, 2.2GHz, 2GHz, and 1.8GHz can all yield energy savings. There is a performance loss in these configurations, but the loss is also smaller than their more sequential counterparts. These configurations may be useful for energy management since they represent a possible trade-off between energy savings and performance loss. We speculate the underlying reason why the more random access patterns react to DVFS better is that random access leads to significant cache misses and instruction pipeline stalls, so the CPU more frequently "wait for" data fetch. When the CPU frequencies are lowered, the relative impact on performance is hence smaller.

The most encouraging results come from Figure 12. Here, especially in the buffered I/Os, lowering CPU frequency can often yield energy savings. This is dramatic for cases such as buffered input (`I/B`), where the energy consumption for 2.2GHz is less than half of that of 2.6GHz, whereas the execution time at 2.2GHz turns out being shorter than that of 2.6GHz. This is a "sweet spot" in energy management, the program is not only more energy-efficient, but also runs faster. The cause behind this behavior is that CPUs running such I/O-intensive benchmarks are mostly idle, so lowering the CPU frequency has little impact on performance, but can significantly save energy. The improved performance may come as a mild surprise to some; we believe this demonstrates the execution time is not bound by CPU, but the storage system. The operations of the latter are often less deterministic, causing delays at unpredictable times.

5 Case Study

In this section we apply our findings to two real-world benchmarks, SUNFLOW and XALAN, from the well-known DaCapo suite benchmark [2]. SUNFLOW renders a set of images using ray tracing, a CPU-intensive benchmark. We performed the experiments varying the data types from `int` to `short`, `float`, `double`, and `long` only in the method that renders the figure. The rest of the source code remained untouched. Figure 13 and Figure 14 show the results.

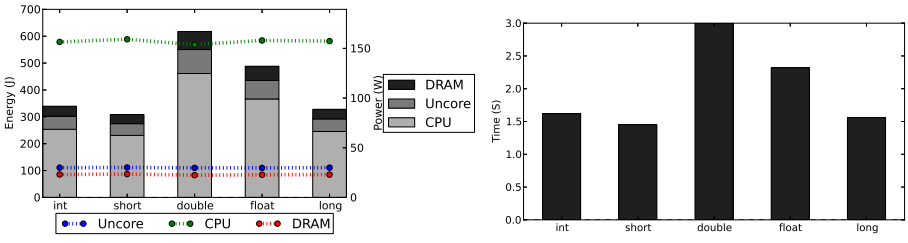


Fig. 13. SUNFLOW: energy/performance behaviors under different data precision choices

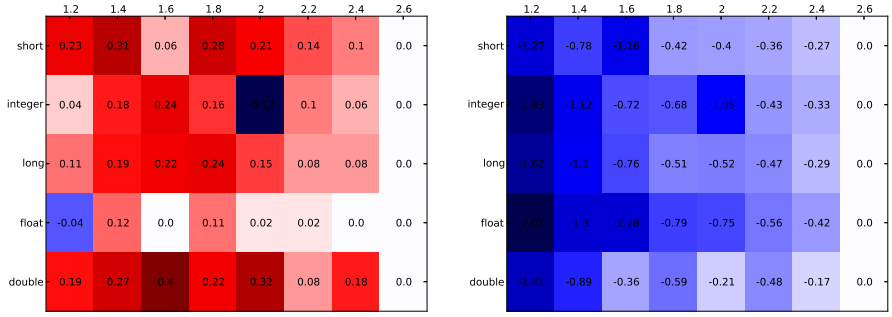


Fig. 14. SUNFLOW: energy/performance behaviors under different data precision choices with DVFS

Some patterns we learned from micro-benchmarking remain. For instance, `short` still consumes less energy than `int`, `float` still consumes less energy than `double`, and `short` and `int` still consume less energy than `float` and `double`. Differently, however, we found that SUNFLOW `float` and `double` data types proportionally consume much more energy than other data types. We believe that this can be explained in terms of the rounding error. In most programs the result of integer computations can be stored in 32 bits. However, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Thus, the result of a floating-point calculation must often be rounded to fit back into its finite representation. Although the IEEE standard requires that the result of addition, subtraction, multiplication and division be exactly rounded, the rounding process is proven to be expensive [10], in particular when operands differ greatly in size. Since there is no need to round the result of our matrix multiplication benchmark, this overhead is not observed.

Also, even though SUNFLOW is a complex application — it has more than 22,000 lines of Java code — we observed that a simple modification on the data types of a single method can have a considerable influence in the overall energy consumption of the application. For this experiment, however, the programmer needs to trade energy consumption for accuracy, since most of the computation are based on floating points. This also explains why `double` and `float` are more energy-consuming.

XALAN transforms XML documents into HTML. This benchmark performs reads and writes from input/output channels, and it has 170,572 lines of Java code. In its default version, the benchmark does not usage a buffer. We apply this modification in two single places in the `XSLTBench` class. When applying this modification, we observed an energy saving of 4.29%. Execution time kept roughly the same. Figure 15 and Figure 16 show the results.

Ultimatelly, it is also important to mention that, according to previous studies [14], micro-optimizations for power and energy are challenging to present significant improvements on real world benchmarks, once the benchmark is already optimized for performance. This is particular true for for all DaCapo benchmarks. We consider these results as an opportunity to further energy optimization.

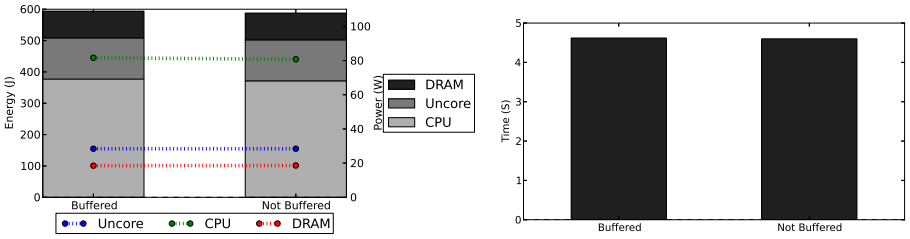
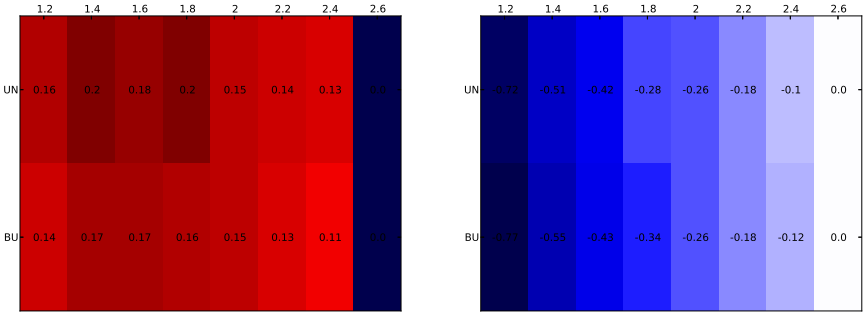


Fig. 15. XALAN: energy/performance behaviors under different data I/O strategies



Energy *Time*

Fig. 16. XALAN: energy/performance behaviors under different data I/O strategies with DVFS

6 Threats to Validity

We divide our discussion on threats to validity into internal factors and external factors.

Internal factors: First, accessing MSRs also consumes energy (see discussions in Section 2.1). This overhead cannot be ignored if MSR accesses are too frequent, *e.g.*, at microsecond intervals. We mitigate this problem by using the RAPL interface only at the beginning and at the end of the benchmark execution. Second, the readings from the RAPL interface are hardware (CPU core or socket)-based. It cannot isolate energy consumption due to OS execution, VM execution, or application execution. As our experiments are mostly set up to be *comparative* — such as demonstrating the difference in energy consumption between sequential vs. random access, and the difference under different frequency settings — but our OS/VM settings remain unchanged throughout experiments, the root cause of relative difference in energy consumption for different experiments is likely to be the (direct and indirect) effect of applications, not OS or VM. Third, analyzing code with a short execution time may disproportionally amplify the noise from hardware and OS. We mitigate this problem by increasing the execution length of our benchmarks (such as via designing the benchmark to operate on a large amount of data) and averaging the results of multiple executions.

External factors: First, our results are limited by our selection of benchmarks. Nonetheless, our corpus spans a wide spectrum, ranging from data access, data representation, data precision, and data intensity distribution over hardware components. Second, there are other possible data manipulations beyond the scope of this paper. With our tool, we expect similar analysis can be conducted by others when other aspects of data-related application features become relevant. Third, our results are reported with the assumption

that JIT is enabled. This stems from our observation that later runs of JIT-enabled executions do stabilize in terms of energy consumption and performance. We experienced differences in standard deviation of over 30% when comparing the warmup run (first 2 executions) and later runs, but less than 5% when comparing the last 4 runs.

7 Related Work

Studying energy efficiency at the application level is an emerging direction. In this section we describe the studies overlapping with the scope of our work.

Energy management. The most established energy management approaches are focused on the hardware level (*e.g.*, [13,6]) and the OS level (*e.g.*, [9,18]). In recent years, a number of studies have explored energy management strategies at the application level as an attempt to empower the application programmer to take energy-aware decisions, since design choices might influence energy efficiency. Some of these studies focus on the design of new programming models, with examples such as Green [1], EnerJ [23], Energy Types [5], and LAB [15]. In these systems, recurring patterns of energy management tasks are incarnated as first-class citizens. Approximated programming [4] trades and reasons about occasional "soft errors", *i.e.*, errors that may reduce the accuracy of the results, for a reduction in energy consumption. The relationship between this line of work and our work is complementary: existing work provides language support to facilitate energy optimization, whereas our work experimentally and empirically establishes the room of the energy optimization space.

Energy measurement. Energy measurement is a broad area of research. Prior work has attempted to model energy consumption at the individual instruction level [26], system call level [7], and bytecode level [24]. Recent progress also includes fine-grained measurement for Android programs [12,17], with detailed energy measurement of different hardware components such as camera, Wi-Fi and GPS. RAPL-based energy measurement has appeared in recent literature (*e.g.*, [14,25]); its precision and reliability has been extensively studied [11].

Empirical studies. Existing research that dealt with the trade-off of comparing individual components of an application and energy consumption has covered a wide spectrum of applications. These studies vary from concurrent programming [20], VM services [3,14], cloud offloading [16], and refactorings [22]. To the best of our knowledge, our study is the first in exploring how different choices of fine-grained data manipulation impact on the energy consumption of different hardware sub-systems, and how application-level energy management and lower-level energy management interact.

8 Conclusion

In this paper, we take a data-centric view to empirically study the optimization space of application-level energy management. Our investigation is unique for several reasons: (1) it focuses on application-level features, instead of hardware performance counters, CPU instructions, or VM bytecode; (2) it is carried out from the data-oriented perspective, charting an optimization space often known to be too "application-specific" to quantify and generalize; (3) it offers the first clues on the impact of unifying application-level energy management and hardware-level energy management; (4) it provides an in-depth analysis from a whole-system perspective, considering energy consumption not only resulting from CPU cores, but also from cache and DRAM.

A detailed description of our results, the source code of `jRAPL`, the benchmarks, and all raw data, can be found online at `http://bit.ly/fase2014`.

References

1. W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.
2. S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, 2006.
3. T. Cao, S. Blackburn, T. Gao, and K. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA*, 2012.
4. M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.
5. M. Cohen, H. Zhu, S. Emgin, and Y. Liu. Energy types. In *OOPSLA*, 2012.
6. H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *ISLPED*, 2010.
7. M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *MobiSys*, 2011.
8. K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *SIGMETRICS*, 2000.
9. R. Ge, X. Feng, W. Feng, and K. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *ICPP*, 2007.
10. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.
11. M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, January 2012.
12. S. Hao, D. Li, W. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, 2013.
13. M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. IEEE Symposium*, 1994.
14. Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *OOPSLA*, pages 329–344, 2014.
15. A. Kansal, T. Saponas, A. Brush, K. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, 2013.
16. Y. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, 2013.
17. D. Li, S. Hao, W. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *ISSTA*, 2013.
18. A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *EuroSys*, 2006.
19. G. Pinto, F. Castor, and Y. Liu. Mining questions about software energy consumption. In *MSR*, 2014.
20. G. Pinto, F. Castor, and Y. Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, 2014.
21. H. Ribic and Y. Liu. Energy-efficient work-stealing language runtimes. In *ASPLOS*, 2014.
22. C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *ESEM*, 2014.
23. A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.
24. C. Seo, S. Malek, and N. Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *PerCom*, 2008.
25. Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. In *ICPE*, 2013.
26. V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13:1–18, 1996.