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@binghamton.edu
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 of energy optimization for computer systems. In this paper, we study the energy impact 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 data management features respond to Dynamic Voltage and Frequency Scaling (DVFS). Finally, we apply our findings to real-world applications, demonstrating their potential for guiding application-level energy optimization. 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.*, [11,5]) and system-level approaches (*e.g.*, [7,18]), energy optimization is gaining momentum at the level of application development [1,3,4,13,15,23]. 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 [16] 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 [2,17], and the guidelines are often anecdotal or incorrect [16]. 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 data management 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 [22] or bytecode [10,14] 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*: Do different representations of the same data (unboxed vs. boxed data, a primitive array vs. an `ArrayList`, an array of objects vs. multiple arrays of primitive data) have impact on energy consumption?
- *data precision*: Do precision levels (*e.g.*, `short`, `int`, and `long`) of data have impact on energy consumption?
- *data I/O strategies*: For I/O-intensive applications, do different choices of data processing — such as buffering — 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 (DVFS) [11], arguably the most classic

hardware-based energy management strategy. This exploration expands the energy optimization space where "software meets hardware, " over a frontier where software engineering research joins forces with hardware architecture research.

Overall, this 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 *data*. The multi-dimensional study ranges from data access pattern, data organization and representation, data precision, and data I/O intensity.
- It conducts experiments to bridge application-level and hardware-level energy management, and constructs 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) [5] 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 monitor the energy consumption information and store 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 (L3 cache, 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 a 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 cable 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 program execution of interest. This problem is magnified for 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, orders of magnitude 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.

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. We chose to report the last 4 runs because JIT execution

tends to stabilize in the later runs [17]. 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.

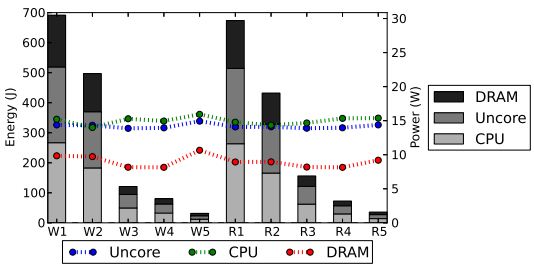
3 Application-Level Energy Management

This section explores the optimization space of application-level energy management through five 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 `int` array (of size N=50,000,000) and retrieves the value at each position, while the write counterpart micro-benchmark 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 results do not consider index array preloading.

The figure on the right shows[3] the benchmarking results, with bars for energy data and lines for power data. We do not explicitly show the execution time, which by physics, can be derived as the division of energy and power. There are 10 bars for each figure, the first five of which (with prefix `W`) indicate write access, and the remaining five (with prefix `R`) indicate read access. In each group, suffix 1 represents 100% randomness in access, 2 for 25% randomness, 3 for 1% randomness, and 4 for 0.1% randomness, and 5 for 0% randomness, *i.e.*, sequential access. The level of randomness is controlled by index range: *e.g.*, we imitate 1% random access by allowing random permutation within each N × 1% interval of the array.

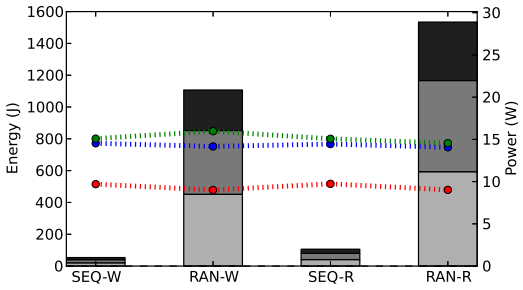


The data reveals the significant impact of access randomness on energy consumption. The more random data access is, the more energy is consumed. This is consistent with hardware behaviors due to cache locality. Further observe that read vs. write accesses make little difference on energy consumption. The conventional folklore is that writes are often more expensive than reads, but this effect, if any, appears to be small on energy consumption. In fact, in one combination `R3` vs. `W3`, the opposite is true.

[3] Throughout the paper, all bar charts follow the same legends as those in this figure.

3.2 Data Representation Strategies

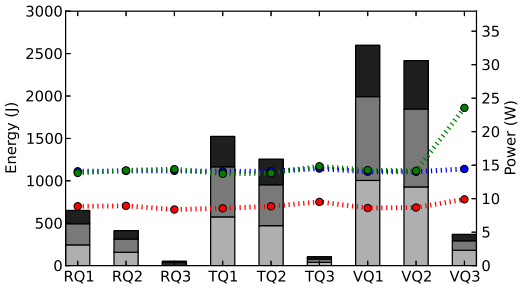
Let us now investigate the impact of different data representation strategies on energy consumption. First, we look into the difference between representing a sequence of integers as a primitive array and as an `ArrayList`. We construct a similar experiment as one described in Section 3.1, by traversing an `ArrayList` of `Integer`'s of a large size (N = 50,000,000). We mimic "read" through the `List.get(int i)` method, and "write" through the `List.set(int i, Object o)` method.



The results of the `ArrayList` implementation are shown the figure on the right, where `SEQ/RAN/R/W` labels denote sequential, 100% random, read, and write access, respectively. Compared with Section 3.1, energy consumption is much 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. `ArrayList` uses boxed data (of `Integer` type) whereas our primitive array implementation uses unboxed data (of `int` type). Furthermore, the getter/setter required by `ArrayList` are method invocations, more expensive than primitive array read/write.

This experiment motivated us to answer a more general question related to object-oriented languages: when we say an object is accessed, which representation of the object is being accessed: a reference to it, a value it holds, or the type it has? Do they have the same effect on energy consumption? We construct the next experiments, in three groups:

- *Reference Query (RQ):* accesses the reference of an `Integer` object;
- *Type Query (TQ):* accesses the type held by an `Integer` object;
- *Value Query (VQ):* accesses the value that an `Integer` object holds;



The result on the right is divided into three groups as above. In each group, postfix 1 denotes 100% random access, 2 denotes 25% random access, and 3 denotes sequential access. For the TQ experiment, our benchmark applies `instanceof` operator to the object. To avoid source-level compiler optimization performed by modern Java compilers such as transforming expression `x instanceof Integer` to a no-op if `x` is only assigned to hold an `Integer` object, our micro-benchmark assigns objects of different types to reference `x`, and the `instanceof` operator cannot be optimized away through standard points-to analysis.

RQ and TQ are both more efficient than VQ. According to the runtime semantics of object-oriented programs, RQ only entails a stack access, whereas VQ includes access to the heap, a much more expensive operation.

```
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. 1. Object-Centric Data Grouping

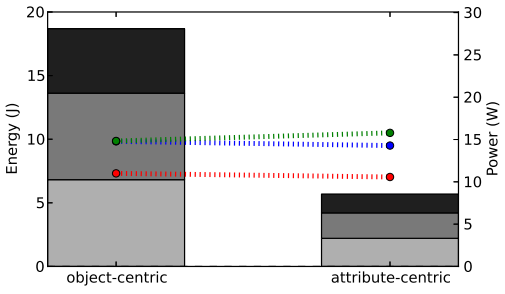
```
class Main {
  int[] a = ..; int[] b = ..; int[] c = ..; int[] d = ..; int[] e = ..;
  void calc() {
    for (int i = 0; i < N; i++) {
      e[i] = a[i] * b[i] * c[i] * d[i];
}}}
```

Fig. 2. Attribute-Centric Data Grouping

Less obvious is the case of TQ. On one hand, the type of an object is stored as object metadata, whose access also requires heap access. As a result, TQ is more expensive than RQ. On the other hand, all objects of the same type share the same metadata representing the type, and repeated queries of the same type yield high cache hits. As a result, TQ is cheaper than VQ.

3.3 Data Organization

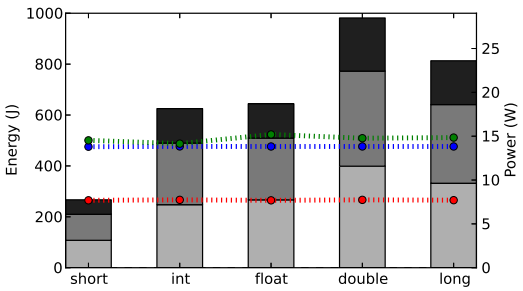
In the next experiment, we consider two programs in Figure 1 and Figure 2. Functionally equivalent, the first *object-centric* program accesses a large array (of size N=50,000,000) of objects with 5 fields, and the second *attribute-centric* program accesses 5 primitive arrays.



As shown here, 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 garbage collection. Another plausible cause is that there is no guarantee that objects in the array are allocated in contiguous space on the heap. As a result, even though Fig. 1 may be cache-friendly for retrieving the 5 fields for the same object, it may incur more cache misses when the entire array is traversed.

3.4 Data Precision Choices

We next analyze the impact of data precision choices on energy consumption. Our micro-benchmark performs the multiplication of two 1000 × 1000 matrices. For our experiments, we vary the matrix element type, declared with the `short`, `int`, `float`, `double`, and `long` types respectively for each variation of the benchmark. In our environment, `short/int/float/double/long` data types are 16/32/32/64/64 bits respectively. To set a fair comparison, we pre-fill matrices with `double` values through random number generation. All other variations of the benchmark pre-fill their matrix data through data conversion from the `double` matrix. In other words, all experiments operate on matrices of comparable values, only with different precisions. Our reported results exclude the pre-filling/converting stage above.

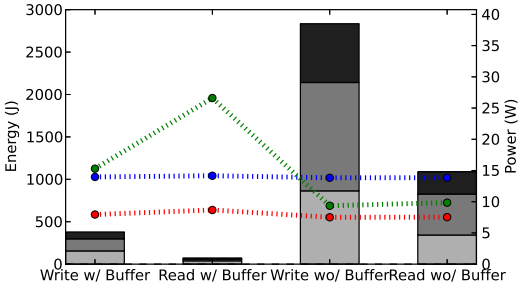


Our experiments show that energy consumption grows with the number of bits (a) among the non-floating point data types, as reflected by the relative standings between `short`, `int`, and `long`. (b) among floating point data types, as seen in the relative standings between `float` and `double`. Both are consistent with architecture-level comparisons, where instructions operating on more bytes/words are more expensive.

It is however unreliable to use the number of bits to cross-compare between non-floating point types and floating point types. Programs with floating point types involve the use of FPUs. Based on our experience, one must be cautious to draw generalizations from cross-comparisons between FPU-intensive programs and those otherwise. For instance, the two 32-bit types used in our benchmarking — `int` and `float` — appear to incur similar amounts of energy consumption. As we shall see in the Section 5 however, the two may also lead to drastically different consumptions. It is a reminder for energy-conscious programmers who wish to save energy by modifying their `float`-precision program to one with `int` precision — the strategy may or may not be effective.

3.5 Data I/O Configurations

Finally, we analyze the energy behaviors of I/O operations. We construct two micro-benchmarks that read and write 50MB data from/to a file, using `FileInputStream` and `FileOutputStream` objects respectively. For the read benchmark, we create two variations, one with buffering through the use of the `BufferedInputStream` object, and the other without. Similarly, the write benchmark has two variations. The one with buffering uses the `BufferedOutputStream` object.



As the figure reveals, buffering has significant impact on improving energy efficiency. Indeed, buffer removal in essence disables bulking of I/O operations, so its effect on energy consumption is dramatic. Furthermore, observe that data output is significantly more energy-consuming than data input. Third, the power consumption of unbuffered file access (about 10W) is lower than that of buffered access. Unbuffered file access leads to a much higher level of I/O intensity. As a result, CPU is more likely to remain idle, and more likely to be scaled down by Linux's default power management strategy, the `ondemand` governor.

RQ1 Summary: *Random access, object-centric data organization, unbuffered I/O consume significantly more energy. The energy consumption for memory read and write are on par, but file write is significantly more expensive than file read. Data types with more bits tend to consume more energy, but there is no simple generalization to cross-compare FPU-intensive types and those that are not.*

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 [11] 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 $E = P * t$.

Result Summary. We have conducted the same experiments reported 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 Section 8 for information). Figure 3 and Figure 4 report selected results. 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 consumptions 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. This is a sober reminder of the applicability of DVFS as an energy management strategy: whereas downscaling can be effective in some scenarios, blind DVFS is likely to fall short in goals. This somewhat pessimistic conclusion does have a positive "coda" — thanks to the difference between micro-benchmarks and real-world applications — a discussion we will continue in Section 5.

Furthermore, observe that we have adopted a very narrow view to equate "being favorable" as being able to save energy. As an example beyond this view,

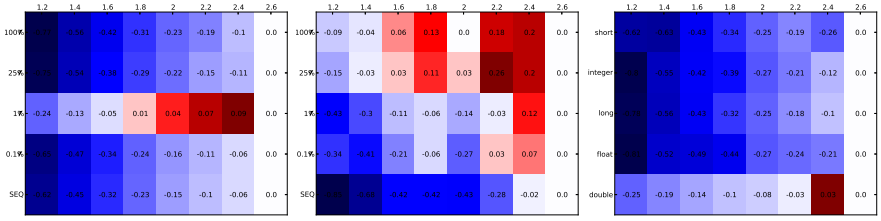


Fig. 3. Selected Energy Results of DVFS Combined with Data Access, Data Representation, and Data Precision. (Labels on top are CPU frequencies, and labels to the left are random/sequential access patterns. All data are normalized energy consumption against the 2.6Ghz data of the same row. Red indicates savings, whereas Blue indicates loss. The darker the Red shade, the more "favorable" the configuration is, *i.e.*, greater energy savings. The darker the Blue shade, the more "unfavorable" the configuration is *i.e.*, greater energy loss.)

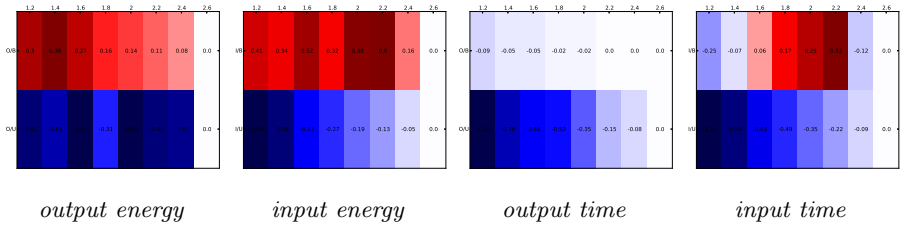


Fig. 4. DVFS and Data I/O (`O`: Output, `I`: Input, `B`: Buffered, `U`: Unbuffered)

running CPUs at the lowest frequency may reduce heat dissipation, and improve the reliability of program execution.

Overall, our results can serve as a "lookup" 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 randomly access an array and query the value held by the array element object (*VQ+Randomness*) with a fixed energy budget, she can look up the results from Figure 3, and run the program either at 2GHz or at 2.6Ghz. The former may reduce heat dissipation, whereas the latter may produce results faster.

Specific Findings. In Figure 3, observe that for *VQ+Randomness*, 100% random access or 25% random access 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 saving and performance loss. The more random access patterns react to DVFS more gracefully because random access leads to more cache misses and instruction pipeline stalls. As a result, the CPU more

frequently "waits for" data fetch. When the CPU frequencies are lowered, the relative impact on performance is smaller.

The most encouraging results come from Figure 4. 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.

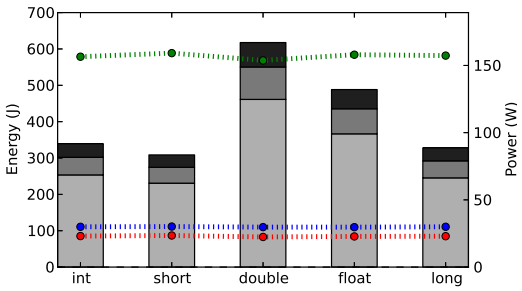
RQ2 Summary: *Blindly downscaling CPU frequency often leads to increased energy consumption and performance loss. Downscaling can play a prominent role in the energy optimization of I/O-intensive benchmarks.*

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[4].

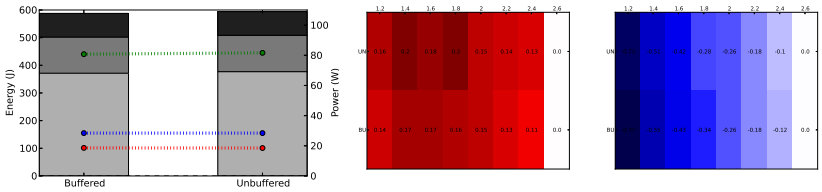
Sunflow renders a set of images using ray tracing, a CPU-intensive benchmark. The original program represents rendering data in type `double`. We performed our experiments by varying the data types appearing in the rendering method from `double` to `short`, `int`, `float`, and `long`. The rest of the source code remained unchanged.

The results here confirm some patterns from micro-benchmarking: for instance, `short` still consumes less energy than `int`, and `float` is still cheaper than `double`. The figure here highlights the non-comparability between floating point types and non-floating point types. The rendering process of Sunflow involves complex floating point operations (such as division), leading to heavy overhead on FPU operations (such as rounding [8]). In other words, these heavy operations significantly outpace their `short`/`int`/`long` counterparts in execution time, and subsequently energy consumption.



Another observation is that the difference in energy consumption of `short`, `int` and `long` is not as drastic as the one in micro-benchmarking. The same also holds for the difference between `float` and `double`. Real-world programs such as

[4] http://www.dacapobench.org



Buffered vs. Unbuffered *DVFS Energy* *DVFS Time*

Fig. 5. Xalan Results

Sunflow are more likely to lead to instruction pipeline stalls (due to branching, synchronization, *etc*) than micro-benchmarks, and we speculate these additional stalls may help mask some of the difference.

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 on the overall energy consumption of the application. In this example, the energy-aware programmer needs to balance the trade-off between energy efficiency and accuracy.

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 use a buffer. We add buffers to two program points in the `XSLTBench` class. With this modification, we observed an energy saving of 4.29%. Execution time kept roughly the same. The first figure in Figure 5 shows the results. On one hand, the savings here are not as "dramatic" as what micro-benchmarking showed. On the other hand, real-world applications often consume more energy, so a small percentage of savings can still make a difference (4.29% for Xalan implies more than 20J). The insights from micro-benchmarking guide us to identify and perform this optimization.

We also studied the impact of DVFS on Xalan, with results shown in the same figure. In all configurations, executing Xalan can lead to energy savings than running it at 2.6Ghz. This is consistent with our findings in the micro-benchmarking, because Xalan does perform significant I/O operations on files. DVFS may be useful for Xalan when one is willing to trade performance for energy savings.

The power consumptions of Sunflow and Xalan also deserve attention. Different applications operate on very different power levels: the CPU power for Sunflow nearly doubles that for Xalan. Our power analysis is consistent with the established fact that Sunflow is a CPU-intensive benchmark.

Finally, the gap between CPU power consumption and Uncore/DRAM power consumption is much larger than those in micro-benchmarks. This is good news for CPU-centric energy management strategies such as DVFS: they may sometimes be ineffective for micro-benchmarks because Uncore/DRAM power consumption has a large proportion to offset the savings from CPUs, the proportional offset is smaller when we apply these strategies to real-world benchmarks. To validate this, we conducted the data precision experiment of Sunflow over different CPU frequencies and produced a counterpart of the *Precision* heat map

of Fig. 3. As it turns out, unlike all cells are blue in the *Precision* heat map of Fig. 3, most cells are red for SUNFLOW. In other words, 2.6GHz is not the most energy-saving frequency for data precision choices. Readers can find the detailed data of this result in our online repository.

6 Threats to Validity

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 the energy consumption of OS execution, VM execution, and 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 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 the application, 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. Second, there are other possible data manipulations beyond the scope of this paper. With our tool, we expect similar analysis can be conducted in the future 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

Application-level energy management. 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. Some focus on the design of new programming models, with examples such as Green [1], Energy Types [4], and Eco [23]. In these systems, recurring patterns of energy management tasks are incarnated as first-class citizens. Approximated programming [3] 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 [22], system call level [6], and bytecode level [20]. Recent progress also includes fine-grained measurement for Android programs [10,14], 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.*, [12,21]); its precision and reliability has been extensively studied [9].

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 [17], VM services [2,12], cloud offloading [13], and refactoring [19]. 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 distinctive 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 caches and DRAM.

The focus of this paper lies upon "charting" the optimization space. In the future, we are interested in applying the findings in this paper — together with the library we developed — to application-level energy optimization. Two directions that appear to fit nicely with our study are (1) energy co-optimization through program refactoring and deployment-site configuration; (2) energy optimization through search-based software engineering, such as applying the data-oriented characterizations described in the paper as the dimensions of search space, and `jRAPL` as a tool, for software energy optimization.

A full set of the experimental results, the source code of `jRAPL`, the benchmarks, and all raw data, can be found online at `http://kliu20.github.io/jRAPL/`.

Acknowledgments. We thank the anonymous reviewers for their high-quality and thorough comments and suggestions. We also thank Jianfei Hu, Amol Patil, and Harry Xu for useful discussions. This work is partially supported by US NSF CCF-1054515 and CAPES-Brazil.

References

1. W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.
2. 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.
3. M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.
4. M. Cohen, H. Zhu, S. Emgin, and Y. Liu. Energy types. In *OOPSLA*, 2012.
5. H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *ISLPED*, 2010.
6. M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *MobiSys*, 2011.
7. 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.
8. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.
9. 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.
10. S. Hao, D. Li, W. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, 2013.
11. M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. IEEE Symposium*, 1994.
12. Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *OOPSLA*, pages 329–344, 2014.
13. Y. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, 2013.
14. D. Li, S. Hao, W. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *ISSTA*, 2013.
15. Yu David Liu. Energy-efficient synchronization through program patterns. In *Proceedings of GREENS'12*, 2012.
16. G. Pinto, F. Castor, and Y. Liu. Mining questions about software energy consumption. In *MSR*, 2014.
17. G. Pinto, F. Castor, and Y. Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, 2014.
18. H. Ribic and Y. Liu. Energy-efficient work-stealing language runtimes. In *ASPLOS*, 2014.
19. C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *ESEM*, 2014.
20. C. Seo, S. Malek, and N. Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *PerCom*, 2008.
21. Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. In *ICPE*, 2013.
22. 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.
23. Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *ICSE'15*, May 2015.