ENERGY-EFFICIENT JAVA APPLICATIONs: EMPIRICAL STUDIES AND RUNTIME DESIGN

BY

KENAN LIU

B.S., Southwestern University of Finance and Economics, Year 2011
M.S., Binghamton University, Year 2013

DISSERTATION

Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2021

Accepted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2021

April 30, 2021

Yu David Liu, Chair and Principal Advisor
Department of Computer Science, Binghamton University

Dmitry Ponomarev, Committee Member
Department of Computer Science, Binghamton University

Leslie C Lander, Committee Member
Department of Computer Science, Binghamton University

Yu Chen, Outside Examiner
Department of Electric & Computer Engineering, Binghamton University

ABSTRACT

Energy efficiency is increasingly becoming critical for computer systems for at least two reasons. First, widely used platforms such as Android phones, Apple Watch and drones, are powered by battery, and battery life has a direct impact on the usability of these systems. On the higher end, electricity bills are a significant part of operational cost for data centers. The environmental impact of electricity consumption at this scale is not sustainable.

There is a growing interest to study energy consumption on the *application* level. Unlike lower-level energy management strategies that often happen "under the hood," *application-level energy management* has some unique advantages. In a nutshell, applications and their runtime systems can be viewed as a white box, whose structural features may be considered for energy optimization. This dissertation makes three contributions in application-level energy management.

First, we study the energy impact of alternative data management choices by programmers, such as data access patterns, data precision choices, and data organization. we further 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.

Second, we conduct an empirical investigation of 16 Java Collection implementations (13 thread-safe, 3 non-thread-safe) grouped under 3 commonly used forms of data structures (lists, sets, and maps). Using micro- and real world-benchmarks, we

show that small design decisions can greatly impact energy consumption. For example, different implementations of the same thread-safe collection can have widely different energy consumption behaviors. This variation also applies to the different operations that each collection implements, e.g., a collection implementation that performs traversals very efficiently can be more than an order of magnitude less efficient than another implementation of the same collection when it comes to insertions.

Third, we introduce VINCENT, a novel Java Virtual Machine (JVM) for improving the energy efficiency of Java applications. The key insight of VINCENT is that the programming abstraction of methods may serve as an ideal granularity for phase-based energy management, and specifically, DVFS. VINCENT is a multi-pass feedback-directed optimizer with a focus on applying DVFS to hot methods. We show that VINCENT can lead to significant energy savings to Java applications without any modification to either the underlying systems or the application source code.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Energy efficiency plays an critical role for computer systems of both mobile devices and data centers. Battery life of mobile devices and electricity bills of data centers increasingly becoming one of the critical investigation areas across the compute stack. Energy-efficient computing is an active direction of research with results from many computer science areas. The most established area is hardware solutions. For example, how to save energy by designing low power CPUs and caches? Further up on the compute stack, there are also many other system level solutions. For example, there are some mobile device operating systems designed for smart energy management. In recent years, a growing interest is to study energy consumption on the application level. Real word applications are often very large with very complex structures. First, applications are viewed as a white box, whose structural features may be considered for energy optimization. Second, the knowledge of software developers and their design choices can influence energy efficiency. Programmers need to understand the energy behaviors at different levels of software granularities in order to make judicious design decisions in order to improve the energy efficiency. If energy aware programmers wish to make their applications more energy efficient, where should they start?

First, an emerging direction in programming is approximate programming. The premise is programmers can accept different versions of the same program but with different quality of service. For example, the programmer is willing to use single floating

point numbers when the battery is low. Otherwise double should be used. To effective, a crucial question that needs to be answered is how much energy savings could there be when the data type is switched from double to float.

Second, despite Java thread-safe collections of characteristics of the performance, scalability and thread-safety are widely discussed by software developers. The energy efficiency of those collections are still not widely considered in the application design and implementation. That is, the performance is not always the only indicator for even the energy consumption of different implementation of the same collection. For some cases, power consumption plays an important role in the energy consumption. This is especially true for the thread-safe collections – the synchronized locks are commonly used in the collection implementations, which could cause the other threads to wait for the resources and turn to be idle state so the power consumption of the core would be reduced.

Finally, energy optimization should not be studied in isolation. What are the cross layer effects? For example: a classic hardware level energy management technique is dynamic voltage and frequency scaling (DVFS). The unknown is how hardware features interact with application level energy management features. The application runtime, e.g. JVM, Android runtime and etc. serve the excellent candidate for the cross layer between the lower layers and the higher layers. That is, the runtime has accessibility of method specific information and its respective invocation frequency. As a result, unlike its neighbor layers, the runtime has ability to manage the energy consumption with the boundary of programming method.

1.1 Contributions

To understand the energy consumption behavior of the applications to better serve the application level energy optimization at runtime. We have developed a energy profiling Java APIs to let us enable to refine the energy analysis. We further discuss the interac-

tive energy consumption behaviors between application and hardware. As a result, we propose a novel application runtime energy optimization solution.

Data-Oriented Characterization of Application-Level Energy Optimization: we have designed a new methodology of energy measurement – `jRAPL` [1] It is an open source fined grained energy measurement java library running on Intel CPUs with Running Average Power Limit (RAPL) [27]. Instead of traditional meter-based energy profiling, which is hard to synchronize the beginning/end of measurement with the specific program execution stage, `jRAPL` enables us to profile the energy consumption behavior of the program with finer granularity without synchronization needed. We collect the energy profiling information to show: (i) How the application-level data management impact energy consumption (ii) How does application-level energy management interact with hardware-level energy management. We apply the energy optimization strategies to the real-world benchmarks, SUNFLOW and XALAN, from the well-known DaCapo suite[2] [12]. It shows the energy consumption can be reduced by 4.29% without performance loss.

A Comprehensive Study on the Energy Efficiency of Java Thread-Safe Collections: We further explore the energy optimization space on the choice of Java thread-safe collections. By energy profiling of 13 thread-safe collections, we talk the impact of energy consumption base on four points. (i) Different implementation of the same collection (ii) Different operations in the same implementation of a collection (iii) Energy consumption scalability of collections in the aspect of multithreading. (iv) Energy consumption of different collection configurations and usages. With the knowledge of the energy optimization space, we made the small changes in both micro-benchmarks and DaCapo benchmark suite by replacing the collections to be more energy efficient col-

[1] `http://kliu20.github.io/jRAPL/`

[2] `http://www.dacapobench.org/`

lections. We got energy consumption improved by 2x in our micro-benchmarks, and by 10%, in DaCapo benchmark suite.

Vincent: Energy-Efficient Hot Methods for Java Applications: We propose a novel application runtime-based energy optimizer in a JVM, called Vincent. Instead of the existing application runtime-based energy profiling and optimization methodology, which applies for the specific service of application runtime [1], [91], [45], Vincent built on top of JVM is aware of the programming abstraction of methods. It would allow Vincent can align the method-grained energy profiling with the boundary of methods optimization. Thanks to the Just-In-Time (JIT) design, Vincent is aware of the method-grained hotspot to limit the overhead of the energy profiling and optimization.

1.2 Outline

Rest of the dissertation is organized as follows. In Chapter 2, We present the Intel RAPL tenique. The energy profiling technique that jRAPL built on top of. We further discuss Dynamic Voltage and Frequency Scaling (DVFS), the classic energy management strategy. Then we would introduce the JIT componnet of JikesRVM [3, 2], and Adaptive Optimization System (AOS) [5] of JikesRVM for the background of the instrumentation and hot method selection Vincent built on top of. In Chapter 6, we present the related work. In Chapter 3, we present data-Oriented characterization of application-level energy optimization. In Chapter 4, we present the energy efficiency impact of Java thread-safe collections. In Chapter 5, we present Vincent, a JVM-based energy-efficient system for Java application. In Chapter 7, we conclude the dissertation and discuss the future directions.

Chapter 2

Background

Energy profiling and management strategies are the most basic questions of energy efficient computing that need to be answered. In this chapter, we introduce the background concepts related to the energy profiling and management strategy we use. Furthermore, we introduce JIT compiler which is generally used in JVM. Then using AOS as an example to show how JikesRVM recognizes the hot methods. After the hot methods get selected, how the optimizing compilers and Intermediate Representation interoperate.

2.1 RAPL

RAPL is an Intel technology which collects accumulated energy information and places it in a set of non-architectural registers called Machine-Specific Register (MSRs). As Figure 2.1 [1] (edit from Intel Power Governor Documentation) shows, it records the energy consumption from three different domains, e.g. CPU core, CPU uncore(L3 cache, on-chip GPUs, and interconnects), package and DRAM. RAPL has ability to do energy management through writing the bits to some certain domains of registers. Since this dissertation only uses energy profiling part of RAPL. When we talk about jRAPL, it would be equivalent to energy profiling.

[1] `https://software.intel.com/content/www/us/en/develop/articles/intel-power-governor.html`

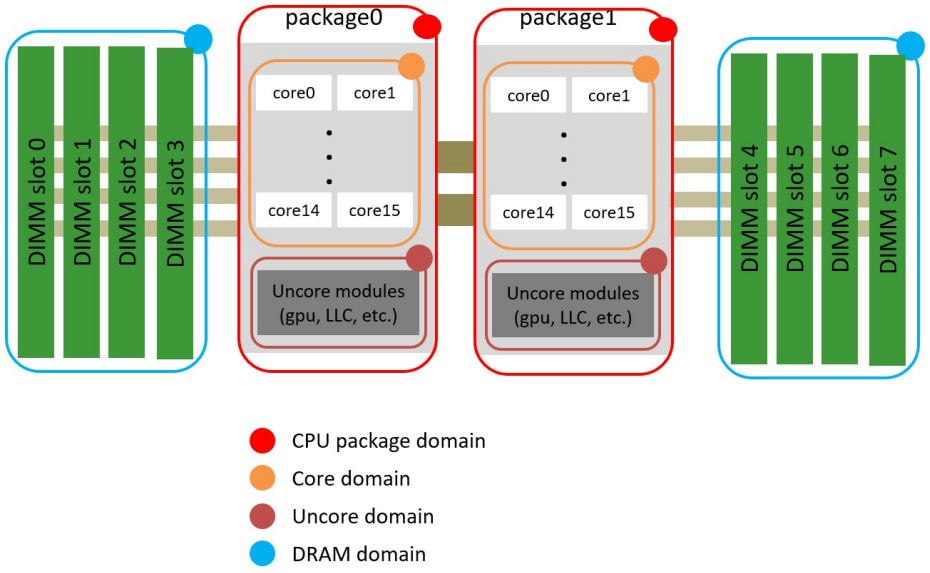


Figure 2.1: RAPL Domains

2.2 DVFS

DVFS [44] 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 $E = P * t$.

2.3 JVM Hot Method Selection

Hot method selection is the essential component of JIT which is commonly supported by JVM-based platform. As Figure 2.2 from [5] shows, AOS serves as a online feedback-directed optimizing component in JikesRVM [3, 2] that is used for hot method selection

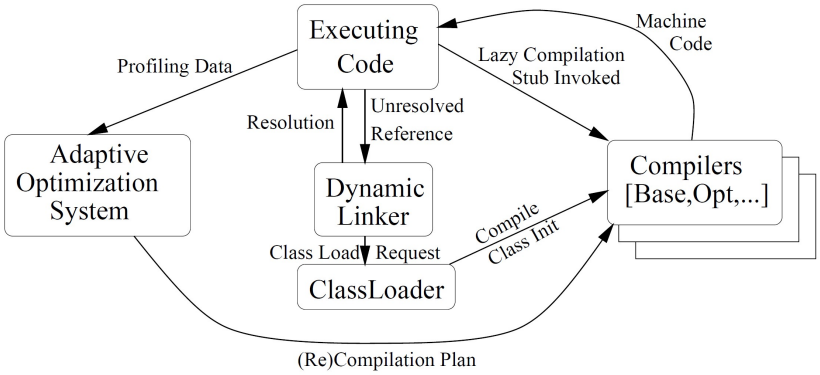


Figure 2.2: Methods Compilation Workflow

and optimizing recompilation decision making. In order to profile the method execution with low cost, AOS takes samples of the method execution and push the count to the corresponding priority queue. When the threshold reached, only the high prioritized methods would be considered as the hot method for the optimizing recompilation uses.

2.4 Instrumentation In JIT

Instrumentation has been widely used to add to the existing program in profiling and tracing. Instrumentation can be injected in the source code directly or in the different compilation stages with IR or binary. Different with instrumentation injection in the static compiler or interpreter, instrumentation injection in JIT can get the profiling information on-the-fly and makes advantage of the hot methods selection features in JIT. As Figure 2.3 [2] shows, JikesRVM has three levels of IR, High Level IR (HIR), Low Level

[2] `https://www.jikesrvm.org/Resources/Presentations/` (Jikes RVM Optimizing Compiler Intermediate Code Representation)

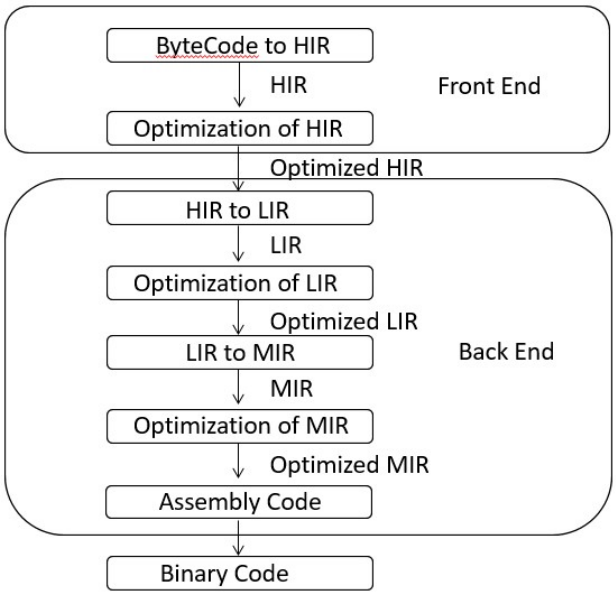


Figure 2.3: Optimizing Compiler Architecture

IR (LIR) and Machine Specific IR (MIR). Each level would optimize and translate the method from higher level IR to the lower level IR. In JikesRVM, the instrumentation can be injected to any level of IR dynamically. In this dissertaion, we injected our energy profiling and energy optimizing instrumentation to HIR to LIR. Once the method is picked as hot method, the instrumentation would be injected and carried all the way to the program ends.

Chapter 3

Data-Oriented Characterization of Application-Level Energy Optimization

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.*, [44, 27]) and system-level approaches (*e.g.*, [34, 77]), energy optimization is gaining momentum at the level of application development [7, 20, 24, 50, 60, 95]. 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 [68] 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 [18, 69], and the guidelines are often anecdotal or incorrect [68]. 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 chapter 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 [88] or bytecode [40, 53] 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) [44], 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 chapter 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.

3.1 Methodology

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

3.1.1 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) [27] 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.

3.1.2 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 [69]. 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.

3.2 Application-Level Energy Management

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

3.2.1 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 Fig. 3.1 shows[1] 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 $\times$ 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.

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

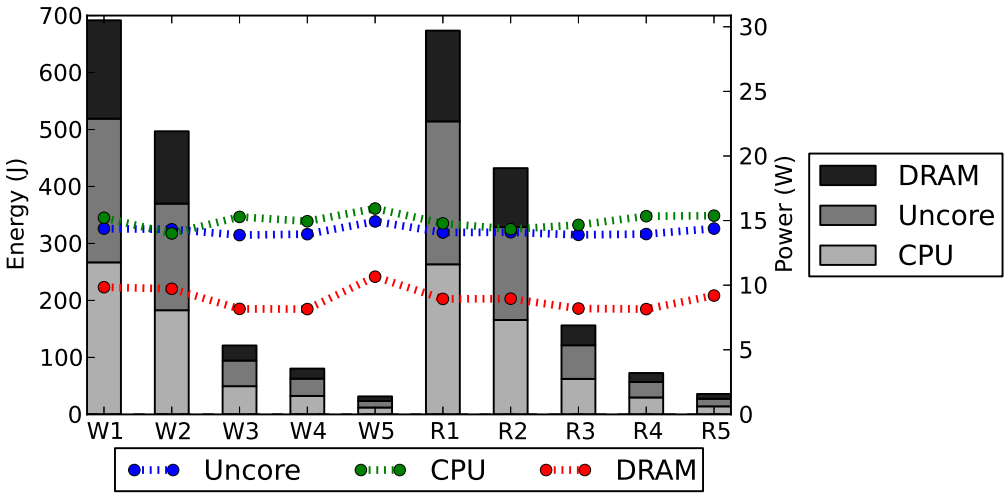


Figure 3.1: Read and Write Access With Different Randomness

3.2.2 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 3.2.1, 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.

As shown in Fig. 3.2 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 3.2.1, 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.

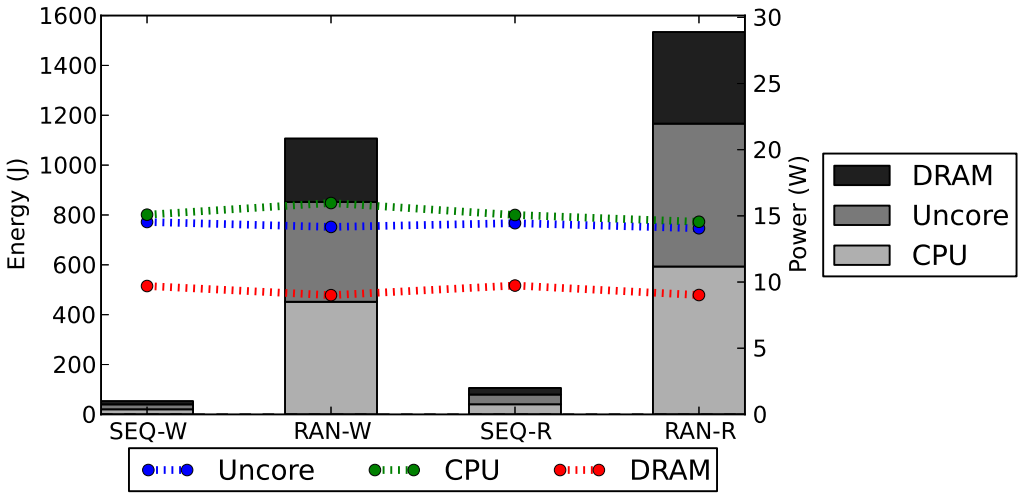


Figure 3.2: Sequential Read/Write VS Random Read/Write

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 Fig. /reffig:ref-type-value-random 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

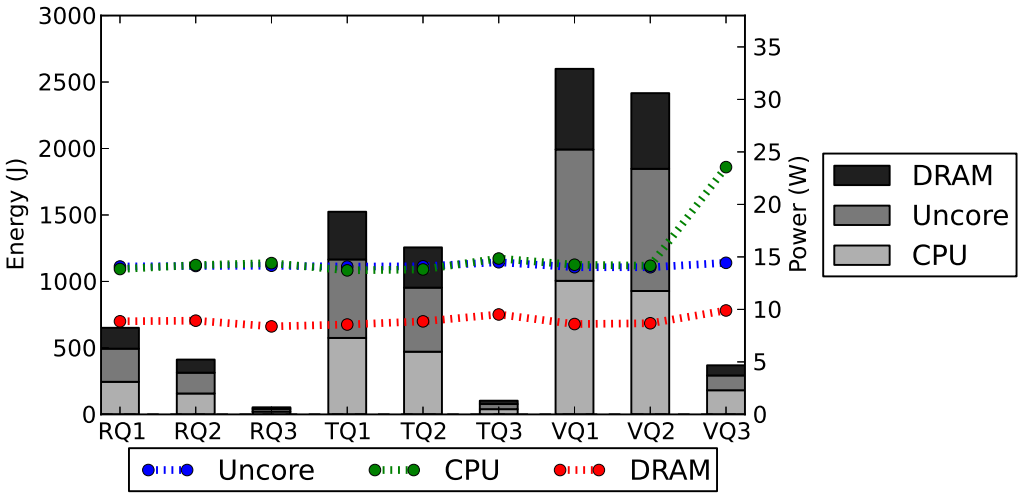


Figure 3.3: Reference/Type/Value Query with different Randomness

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.

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.

3.2.3 Data Organization

In the next experiment, we consider two programs in Figure 3.4 and Figure 3.5. 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.

```
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
          ;
}}}
```

Figure 3.4: 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];
}}}
```

Figure 3.5: Attribute-Centric Data Grouping

As Fig. 3.6 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. 3.4 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.

3.2.4 Data Precision Choices

We next analyze the impact of data precision choices on energy consumption. Our micro-benchmark performs the multiplication of two 1000 $\times$ 1000 matrices. For our

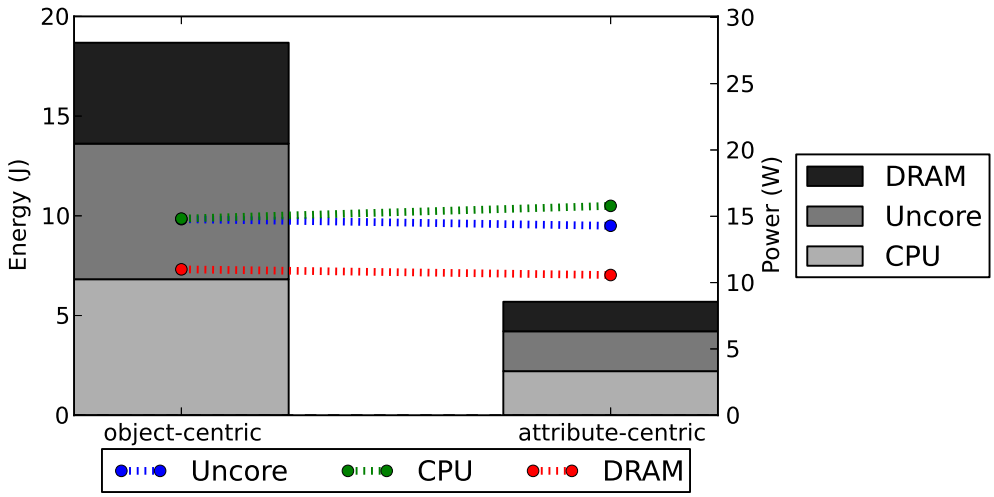


Figure 3.6: Energy Behavior Under Different Data Organizations.

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.

As shown in Fig. 3.7, 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

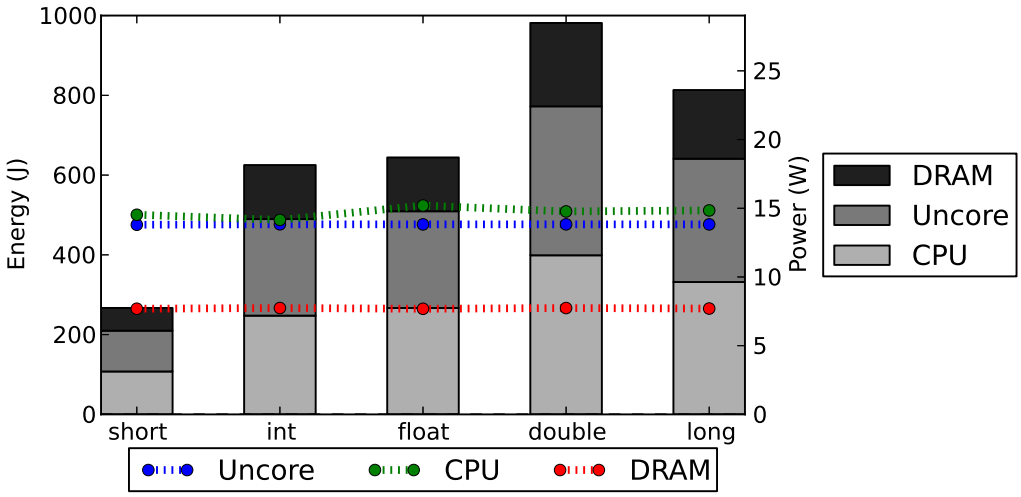


Figure 3.7: Energy Behavior Under Different Data Types

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 3.4 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.

3.2.5 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 Fig. 3.8 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

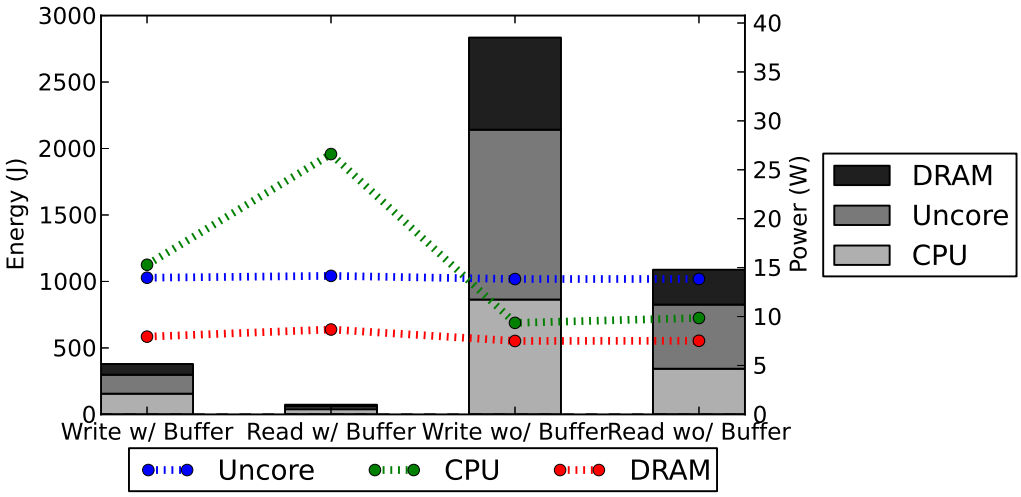


Figure 3.8: Energy behaviors of data I/O operations

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.*

3.3 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 [44] 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 3.6 for information). Figure 3.9 and Figure 3.10 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 3.4.

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

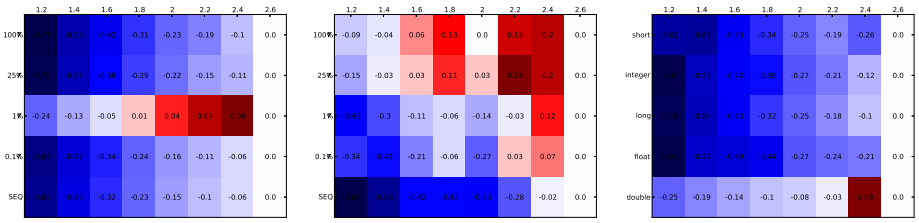


Figure 3.9: Selected Energy Results of DVFS Combined with Data Access, Data Representation, and Data Precision. (Labels on top are CPU frequencies, and labels to the left are random/sequential access patterns. All data are normalized energy consumption against the 2.6Ghz data of the same row. Red indicates 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.)

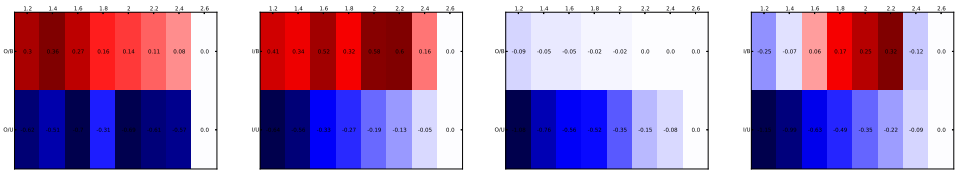


Figure 3.10: DVFS and Data I/O (`O`: Output, `I`: Input, `B`: Buffered, `U`: Unbuffered)

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 3.9, 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 3.9, 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 3.10. 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.*

3.4 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. The original program represents rendering data in type `double`. As the Fig. 3.11 We

[2] `http://www.dacapobench.org`

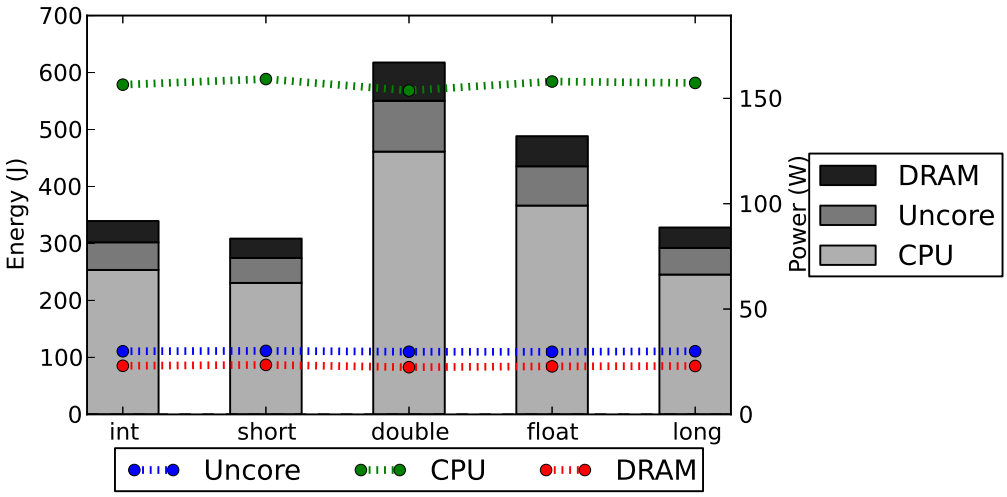


Figure 3.11: SUNFLOW: energy behaviors under different data precision choices

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 [37]). 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 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.

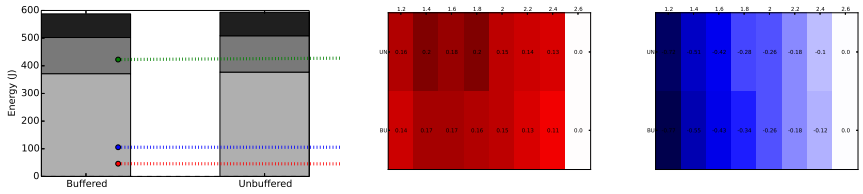


Figure 3.12: XALAN Results

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 3.12 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. Differ-

ent 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. 3.9. As it turns out, unlike all cells are blue in the *Precision* heat map of Fig. 3.9, 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.

3.5 Threats to Validity

Internal factors: First, accessing MSRs also consumes energy (see discussions in Section 3.1.1). 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 chapter. 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.

3.6 Chapter Summary

In this chapter, 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 chapter lies upon "charting" the optimization space. In the future, we are interested in applying the findings in this chapter — 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/`.

Chapter 4

A Comprehensive Study on the Energy Efficiency of Java Thread-Safe Collections

4.1 Introduction

A question that often arises in software development forums is: "since Java has so many collection implementations, which one is more suitable to my problem?"[1]. Answers to this question come in different flavors: these collections serve for different purposes and have different characteristics in terms of performance, scalability and thread-safety. Developers should consider these characteristics in order to make judicious design decisions about which implementation best fits their problems. In this study, we consider one additional attribute: *energy efficiency*.

Traditionally addressed by hardware-level (e.g. [22, 28]) and system-level approaches (e.g. [77, 10]), energy optimization is gaining momentum in recent years through application-level software engineering techniques (e.g. [26, 51, 79]). The overarching premise of this crescent direction is that the high-level knowledge from software engineers on application design and implementation can make significant impact on energy consumption, as confirmed by recent empirical studies [54, 74]. The space for application-level energy optimization, however, is diverse. Developers sometimes rely on conventional wisdom, consult software development forums and blogs, or simply search online for

[1] `http://stackoverflow.com/search?q=which+data+structure+use+java+is:question`

"tips and tricks." Many of these guidelines are often anecdotal or even incorrect [72].

In this chapter, we elucidate one important area of the application-level optimization space: the energy consumption of different Java thread-safe collections running on parallel architectures. This is a critical direction at the junction of data-intensive computing and parallel computing, which deserves more investigation due to at least three reasons:

- Collections are one of the most important building blocks of computer programming. Multiplicity, a collection may hold many pieces of data items, is the norm of their use, and it often contributes to significant memory pressure, and performance problems in general, of modern applications where data are often intensive [92, 14].
- Not only high-end servers but also desktop machines, smartphones and tablets need concurrent programs to make best use of their multi-core hardware. A CPU with more cores (say 32) often dissipates more power than one with fewer cores (say 1 or 2) [56].
- Mainstream programming languages often provide a number of implementations for the same collection and these implementations have potentially different characteristics in terms of energy efficiency [41].

The current work is an broader extension of a preliminary study [71]. It is aimed at evaluating the performance and energy consumption characteristics of 16 Java collection implementations (13 thread-safe, 3 non-thread-safe) grouped by 3 well-known interfaces: `List`, `Set`, and `Map`. Through extensive experiments conducted in a multi-core environment, we correlate energy behaviors of different thread-safe implementations of Java collections and their knobs. Our research is motivated by the following questions:

RQ1. Do different implementations of the same collection have different impacts on energy consumption?

RQ2. Do different operations in the same implementation of a collection consume energy differently?

RQ3. Do collections scale, from an energy consumption perspective, with an increasing number of concurrent threads?

RQ4. Do different collection configurations and usages have different impacts on energy consumption?

In order to answer **RQ1** and **RQ2**, we select and analyze the behaviors of three common operations — traversal, insertion and removal — for each collection implementation. To answer **RQ3**, we analyze how different implementations scale in the presence of multiple threads. In this experiment, we cover the spectrum including both under-provisioning (the number of threads exceeds the number of CPU cores) and over-provisioning (the number of CPU cores exceeds the number of threads). In **RQ4**, we analyze how different configurations — such as the load factor and the initial capacity of the collection — impact energy consumption.

We analyze energy-performance trade-offs in diverse settings that may influence the high-level programming decisions of energy-aware programmers. We cross-validade the main findings into two well-known benchmarks (XALAN and TOMCAT). To gain confidence in our results in the presence of platform variations and measurement environments, we employ two machines with different architectures (a 32-core AMD vs. a 16-core Intel). We further use two distinct energy measurement strategies, relying on an external energy meter, and internal Machine-Specific Registers (MSRs), respectively.

Our study produces a list of interesting findings, some of which are not obvious. To highlight one of them, our experiments show that simple small changes have the potential of improving the energy consumption by 2x, in our micro-benchmarks, and by 10%, in our real-world benchmarks.

4.2 Study Setup

In this section we describe the benchmarks that we analyzed, the infrastructure and the methodology that we used to perform the experiments.

4.2.1 Benchmarks

The benchmarks used in this study consist of 16 commonly used collections (13 thread-safe, 3 non-thread-safe) available in the Java programming language. Our focus is on the thread-safe implementations of the collection. Hence, for each collection, we selected a single non-thread-safe implementation to serve as a baseline. For each implementation, we analyzed insertion, removal and traversal operations. We grouped these implementations by the logical collection they represent, into three categories:

`Lists (java.util.List)`: `Lists` are ordered collections that allow duplicate elements. Using this collection, programmers can have precise control over where an element is inserted in the list. The programmer can access an element using its index, or traverse the elements using an `Iterator`. Several implementations of this collection are available in the Java language. We used `ArrayList`, which is not thread-safe, as our baseline. We studied the following thread-safe `List` implementations: `Vector`, `Collections.synchronizedList()`, and `CopyOnWriteArrayList`. The main difference between `Vector` and `Collections.synchronizedList()` is their usage pattern in programming. With `Collections.synchronizedList()`, the programmer creates a wrapper around the current `List` implementation, and the data stored in the original `List` object does not need to be copied into the wrapper object. It is appropriate in cases where the programmer intends to hold data in a non-thread-safe `List` object, but wishes to add synchronization support. With `Vector`, on the other hand, the data container and the synchronization support are unified so it is not possible to keep an underlying structure (such as `LinkedList`) separate from the object managing the

synchronization.

`CopyOnWriteArrayList` creates a copy of the underlying `ArrayList` whenever a mutation operation (e.g. using the `add` or `set` methods) is invoked.

`Maps (java.util.Map)`: `Maps` are objects that map keys to values. Logically, the keys of a map cannot be duplicated. Each key is uniquely associated with a value. An insertion of a (key, value) pair where the key is already associated with a value in the map results in the old value being replaced by the new one. Our baseline thread-unsafe choice is `LinkedHashMap`, instead of the more commonly used `HashMap`. This is because the latter sometimes caused non-termination during our experiments[2]. Our choice of thread-safe `Map` implementations includes `Hashtable`, `Collections.synchronizedMap()`, `ConcurrentSkipListMap`, `ConcurrentHashMap`, and `ConcurrentHashMapV8`. The difference between `ConcurrentHashMap` and `ConcurrentHashMapV8` is that the latter is an optimized version released in Java 1.8, while the former is the version present in the JDK until Java 1.7. While all `Map` implementations share similar functionalities and operate on a common interface, they are particularly known to differ in the order of element access at iteration time. For instance, while `LinkedHashMap` iterates in the order in which the elements were inserted into the map, a `Hashtable` makes no guarantees about the iteration order.

`Sets (java.util.Set)`: `Sets` model the mathematical set abstraction. Unlike `Lists`, `Sets` do not count duplicate elements, and are not ordered. Thus, the elements of a set cannot be accessed by their indices, and traversals are only possible using an `Iterator`. Among the available implementations, we used `LinkedHashSet`, which is not thread-safe, as our baseline. Our selection of thread-safe `Set` implementations includes `Collections.synchronizedSet()`, `ConcurrentSkipListSet`,

[2] A possible explanation can be found here: `http://mailinator.blogspot.com/2009/06/beautiful-race-condition.html`

ConcurrentHashSet, CopyOnWriteArraySet, and ConcurrentHashSetV8. It should be noted that both ConcurrentHashSet and ConcurrentHashSetV8 are not top-level classes readily available in the JDK library. Instead, they are supported through the returned object from Collections.newSetFromMap(new ConcurrentHashMap<String,String>()) or the analogous implementation Collections.newSetFromMap(new ConcurrentHashMapV8<String,String>()) respectively. The returned Set object observes the same ordering as the underlying map.

4.2.2 Experimental Environment

To gain confidence in our results in the presence of platform variations, we run each experiment on two significantly different platforms:

- System#1: A 2×16-core AMD Opteron 6378 processor (Piledriver microarchitecture), 2.4GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 32KB per core, L2 with 256KB per core, and L3 20480 (Smart cache). It is running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64), and Oracle HotSpot 64-Bit Server VM (build 21) JDK version 1.7.0_11.

- System#2: A 2×8-core (32-cores when hyper-threading is enabled) Intel(R) Xeon(R) E5-2670 processor, 2.60GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 48KB per core, L2 with 1024KB per core, and L3 20480 (Smart cache). It is running Debian 6 (kernel 3.0.0-1-amd64) and Oracle HotSpot 64-Bit Server VM (build 14), JDK version 1.7.0_71.

When we performed the experiments with Sets and Maps, we employed the jsr166e library[3], which contains the ConcurrentHashMapV8 implementation. Thus, these experiments do not need to be executed under Java 1.8.

[3]Available at: http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166e/

We also used two different energy consumption measurement approaches. For `System#1`, energy consumption is measured through current meters over power supply lines to the CPU module. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second and the unit of the current sample is *deca-ampere* (10 ampere). Since the supply voltage is stable at 12V, energy consumption is computed as the sum of current samples multiplied by $12 \times 0.01 \times 10$. We measured the "base" power consumption of the OS when there is no JVM (or other application) running. The reported results are the measured results *modulo* the "base" energy consumption.

For `System#2`, we used `jRAPL` [58], a framework that contains a set of APIs for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) [28] 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 processors. RAPL-enabled architectures monitor the energy consumption information and store it in Machine-Specific Registers (MSRs). Due to architecture design, the RAPL support for `System#2` can access CPU core, CPU uncore data (*i.e.*, caches and interconnects), and in addition DRAM energy consumption data. RAPL-based energy measurement has appeared in recent literature (e.g. [47, 86]); its precision and reliability have been extensively studied [39].

As we shall see in the experiments, DRAM power consumption is nearly constant. In other words, even though our meter-based measurement strategy only considers the CPU energy consumption, it is still indicative of the relative energy consumptions of different collection implementations. It should be noted that the stability of DRAM power consumption through RAPL-based experiments does not contradict the established fact that the energy consumption of memory systems is highly dynamic [11]. In that context, memory systems subsume the entire memory hierarchy, and most of the variations are caused by caches [65] — part of the "CPU uncore data" in our experi-

ments.

All experiments were performed with no other load on the OS. We conform to the default settings of both the OS and the JVM. Several default settings are relevant to this context: (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 10 times within the same JVM; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs. We chose the last three runs because, according to a recent study, JIT execution tends to stabilize in the latter runs [74]. Hyper-threading is enabled and turbo Boost feature is disabled on `System#2`.

4.3 Study Results

In this section, we report the results of our experiments. Results for **RQ1** and **RQ2** are presented in Section 4.3.1, describing the impact of different implementations and operations on energy consumption. In Section 4.3.2 we answer **RQ3** by investigating the impact of accessing data collections with different numbers of threads. Finally, in Section 4.3.3 we answer **RQ4** by exploring different "tuning knobs" of data collections.

4.3.1 Energy Behaviors of Different Collection Implementations and Operations

For **RQ1** and **RQ2**, we set the number of threads to 32 and, for each group of collections, we performed and measured insertion, traversal and removal operations.

- For the insertion operation, we start with an empty data collection, and have each thread insert 100,000 elements. Hence, at the end of the insertion operation, the total number of elements inside the collection is 3,200,000. To avoid duplicate

elements, each insertion operation adds a `String` object with value *thread-id* + *"-"* + *current-index*.

- For the traversal operation, each thread traverses the entire collection generated by the insertion operation, *i.e.*, over 3,200,000 elements. On `Sets` and `Maps`, we first get the list of keys inserted, and then we iterate over these keys in order to get their values. On `Lists`, the traversal operation is performed using a top-level loop over the collection, accessing each element by its index using the `E get(int i)` method of each collection class, where `E` is the generic type (instantiated to `String` in our experiments).

- For the removal operation, we start with the collection with 3,200,000 elements, and remove the elements one by one. For `Maps` and `Sets`, the removals are based on keys, and we remove until the collection becomes empty. On `Lists`, however, the removal operation is based on indexes, and occurs *in-place* — that is, we do not traverse the collection to look up for a particular element before removal.

  Here, according to the `List` documentation, the `E remove(int i)` method "*removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.*"[4]. As we shall see, removal operations on `Lists` are excessively expensive. In order to make it feasible to perform all experiments, we chose to remove only half of the elements.

Lists. Figure 4.1 shows the energy consumption (bars) and power consumption (lines) results of our `List` experiments. Each bar represents one `List` implementation. The three graphs at top of the figure are collected from `System#1`, whereas the three graphs in the bottom are from `System#2`. We do not show the figures for

[4]Documentation available at `http://docs.oracle.com/javase/7/docs/api/java/util/List.html#remove(int)`

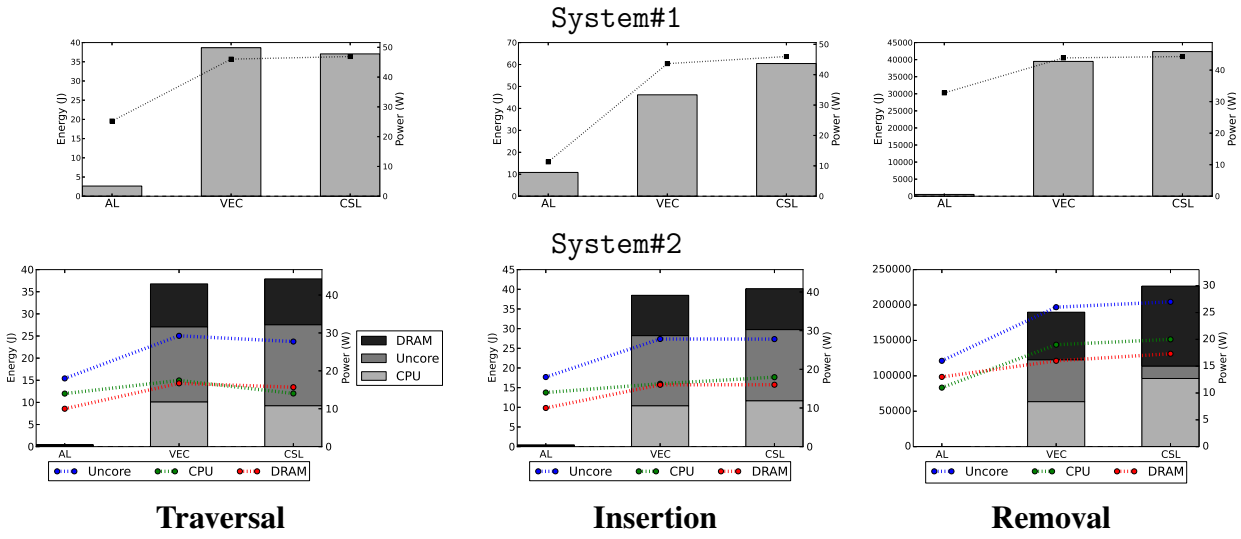


Figure 4.1: Energy and power results for traversal, insertion and removal operations for different `List` implementations. Bars denote energy consumption and lines denote power consumption. `AL` means `ArrayList`, `VEC` means `Vector`, and `CSL` means `Collections.synchronizedList()`.

`CopyOnWriteArrayList` because the results for insertion and removal are an outlier and would otherwise skew the proportion of the figures.

First, we can observe that synchronization does play an important role here. As we can see, `ArrayList`, the non-thread-safe implementation, consumes much less energy than the other ones, thanks to its lack of synchronization. `Vector` and `Collection.synchronizedList()` are similar in energy behaviors. The greatest difference is seen on insertion, on `System#1`, in which the former consumed about 24.21% less energy than the former. `Vector` and `Collection.synchronizedList()` are strongly correlated in their implementations, with some differences. While both of them are thread-safe on insertion and removal operations, `Collection.synchronizedList()` is not thread-safe on traversals, when performing through an `Iterator`, whereas `Vector` is thread-safe on the `Iterator`. `CopyOnWriteArrayList`, in contrast, is thread-safe in all operations. However, it does not need synchronization on traversal operations, which makes this implementation

more efficient than the thread-safe ones (it consumes 46.38x less energy than `Vector` on traversal).

Furthermore, different operations can have different impacts. As we can see on traversal, the `Vector` implementation presents the worst result among the benchmarks: it consumes 14.58x more energy and 7.9x more time than the baseline on `System#1`(84.65x and 57.99x on `System#2`, respectively). This is due both `Vector` and `Collection.synchronizedList()` implementations need to synchronize in traversal operations. As mentioned contrast, the `CopyOnWriteArrayList` implementation is more efficient than the thread-safe implementation.

For insertion operations, `ArrayList` consumes the least energy for both `System#1` and `System#2`. When comparing the thread-safe implementations, `Collections.synchronizedList()` consumes 1.30x more energy than `Vector` (1.24x for execution time) on `System#1`. On `System#2`, however, they consume barely the same amount of energy (`Collections.synchronizedList()` consumes 1.01x more energy than `Vector`). On the other hand, `CopyOnWriteArrayList` consumes a total of 6,843.21 J, about 152x more energy than `Vector` on `System#1`. This happens because, for each new element added to the list, the `CopyOnWriteArrayList` implementation needs to synchronize and create a fresh copy of the underlying array using the `System.arraycopy()` method. As discussed elsewhere [74, 29], even though the `System.arraycopy()` behavior can be observed in sequential applications, it is more evident in highly parallel applications, when several processors are busy making copies of the collection, preventing them from doing important work. Although this behavior makes this implementation thread-safe, it is ordinarily too costly to maintain the collection in a highly concurrent environment where insertions are not very rare events.

Moreover, removals usually consumes much more energy than the other operations. For instance, removal on `Vector` consumes about 778.88x more energy than insertion on `System#1`. Execution time increases similarly, for instance, it took about 92 seconds

to complete a removal operation on `Vector`. By way of contrast, insertions on a `Vector` takes about 1.2 seconds. We believe that several reasons can explain this behavior. First, the removal operations need to compute the size of the collection in each iteration of the for loop and, as we shall see in Section 4.3.4, such naive modification can greatly impact both performance and energy consumption. The second reason is that each call to the `List.remove()` method leads to a call to the `System.arrayCopy()` method in order to resize the `List`, since all these implementations of `List` are built upon arrays. In comparison, insertion operations only lead to a `System.arrayCopy()` call when the maximum number of elements is reached.

Power consumption also deserves attention. Since `System.arrayCopy()` is a memory intensive operation, power consumption decreases, and thus, execution time increase. Moreover, for most cases, power consumption follows the same shape of energy. Since energy consumption is the product of power consumption and time, when power consumption decreases and energy increase, execution time tends to increase. This is what happens on removal on `System#2`. The excessive memory operations on removals, also observed on DRAM energy consumption (the black top-most part of the bar), prevents the CPU to do useful work, which increases the execution time.

We also observed that the baseline benchmark on `System#2` consumes the least energy when compared to the baseline on `System#1`. We atribute that to our energy measurement approaches. While RAPL-based measurement can be efficient in retrieving only the necessary information (for instance, package energy consumption), our hardware-based measurement gathers energy consumption information pertaining to everything that happens in the CPU. Such noise can be particularly substantial when the execution time is small.

For all aforementioned cases, we observed that energy follows the same shape as time. At the first impression, this finding might seem to be "boring". However, recent studies have observed that energy and time are often not correlated [55, 74, 89], which

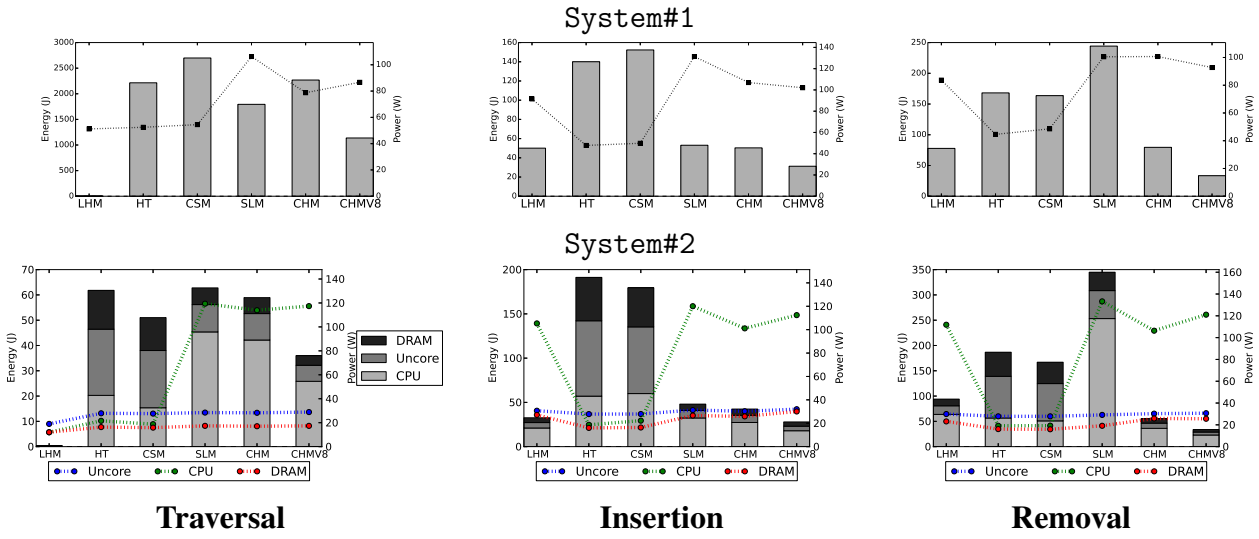


Figure 4.2: Energy and power results for traversal, insertion and removal operations for different `Map` implementations. Bars mean energy consumption and line means power consumption. `LSM` means `LinkedHashMap`, `HT` means `Hashtable`, `CSM` means `Collections.synchronizedMap()`, `SLM` means `ConcurrentSkipListMap`, `CHM` means `ConcurrentHashMap`, and `CHMV8` means `ConcurrentHashMapV8`.

is particularly true for concurrent applications. For this set of benchmarks, however, we believe that developers can safely use time as a proxy for energy, which can be a great help when refactoring an application to consume less energy.

Ultimately, although we have found some differences in the results, both `System#1` and `System#2` presented a compelling uniformity.

Maps. Figure 4.2 presents a different picture for the `Map` implementations. For the `LinkedHashMap`, `Hashtable`, and `Collections.synchronizedMap()` implementations, energy follows the same curve as time, for both traversal and insertion operations, on both `System#1` and `System#2`. Surprisingly, however, the same cannot be said for the removal operations. Removal operations on `Hashtable` and `Collections.synchronizedMap()` exhibited energy consumption that is proportionally smaller than their execution time for both systems. Such behavior is due to a drop on power consumption. Since such collections are single-lock based, for each removal operation, the other threads need to

wait until the underling structure is rebuilt. This synchronization prevents the collection to speed-up, and also decreases power usage.

On the other hand, for the `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` implementations, more power is being consumed behind the scenes. As we mentioned that energy consumption is the product of power consumption and time, if the benchmark receives a 1.5x speed-up but, at the same time, yields a threefold increase in power consumption, energy consumption will increase twofold. This scenario is roughly what happens in traversal operations, when transitioning from `Hashtable` to `ConcurrentHashMap`. Even though `ConcurrentHashMap` produces a speedup of 1.46x over the `Hashtable` implementation on `System#1`, it achieves that by consuming 1.51x more power. As a result, `ConcurrentHashMap` consumed slightly more energy than `Hashtable` (2.38%). On `System#2`, energy consumption for `Hastable` and `ConcurrentHashMap` are roughly the same. This result is relevant mainly because several textbooks [67], research papers [30] and internet blog posts [36] suggest `ConcurrentHashMap` as the *de facto* replacement for the old associative `Hashtable` implementation. Our result suggests that the decision on whether or not to use `ConcurrentHashMap` should be made with care, in particular, in scenarios where the energy consumption is more important than performance. However, the newest `ConcurrentHashMapV8` implementation, released in the version 1.8 of the Java programming language, handles large maps or maps that have many keys with colliding hash codes more gracefully. On `System#1`, `ConcurrentHashMapV8` provides performance savings of 2.19x when compared to `ConcurrentHashMap`, and energy savings of 1.99x in traversal operations (these savings are, respectively, 1.57x and 1.61x in insertion operations, and 2.19x and 2.38x in removal operations). In addition, for insertions and removals operations on both systems, `ConcurrentHashMapV8` has performance similar or even better than the not thread-safe implementation.

`ConcurrentHashMapV8` is a completely rewritten version of its predecessor. The

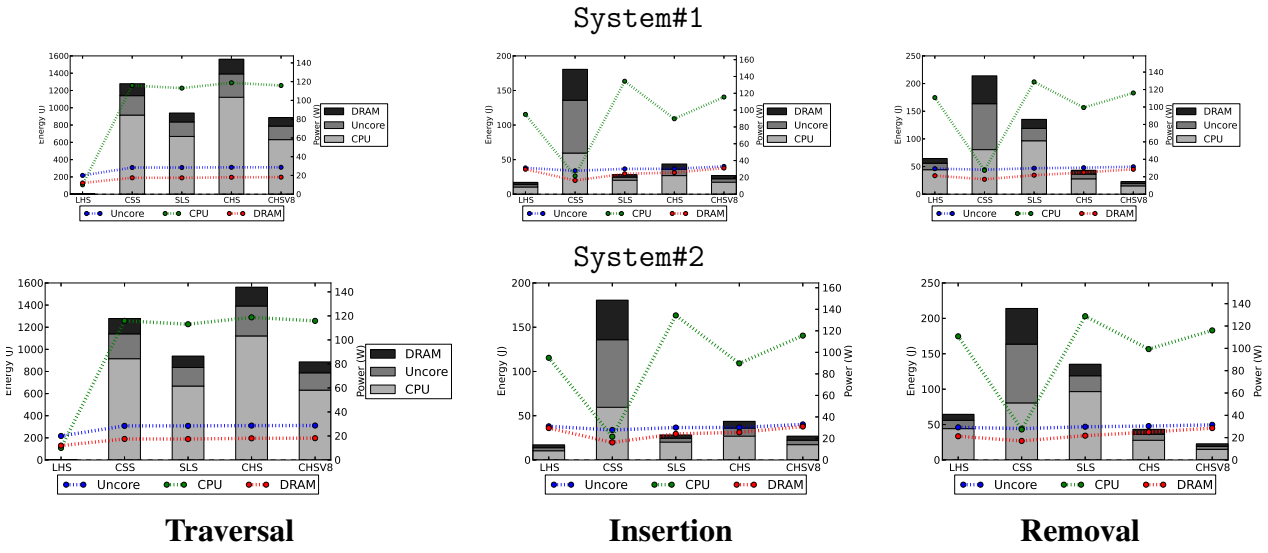


Figure 4.3: Energy and power results for traversal, insertion and removal operations for different `Set` implementations. Bars mean energy consumption and lines mean power consumption. `LSH` means `LinkedHashSet`, `CSS` means `Collections.synchronizedSet()`, `SLS` means `ConcurrentSkipListSet`, `CHS` means `ConcurrentHashSet`, and `CHSV8` means `ConcurrentHashSetV8`.

primary design goal of this implementation is to maintain concurrent readability (typically on the `get()` method, but also on `Iterators`) while minimizing update contention. This map acts as a binned hash table. Internally, it uses tree-map-like structures to maintain bins containing more nodes than would be expected under ideal random key distributions over ideal numbers of bins. This tree also requires an additional locking mechanism. While list traversal is always possible by readers even during updates, tree traversal is not, mainly because of tree-rotations that may change the root node and its links. Insertion of the first node in an empty bin is performed with a Compare-And-Set operation. Other update operations (insertional, removal, and replace) require locks. Locking support for these locks relies on builtin "synchronized" monitors.

Sets. Figure 4.3 shows the results of our experiments with `Set`. We did not present the results for `CopyOnWriteHashSet` in this figure because it exhibited a much higher energy consumption, which made the figure difficult to read. First, for all of the imple-

mentations of Set, we can observe that energy consumption follows the same behavior of power on traversal operations for both System#1 and System#2. However, for insertion and removal operations, they are not always proportional.

Notwithstanding, an interesting trade-off can be observed when performing traversal operations. As expected, the non-thread-safe implementation, LinkedHashSet, achieved the least energy consumption and execution time results, followed by the CopyOnWriteArraySet implementation. We believe that the same recommendation for CopyOnWriteArrayList fits here: this collection should only be used in scenarios where reads are much more frequent than insertions. For all other implementations, the ConcurrentHashSetV8 presents the best results among the thread-safe ones. Interestingly, for traversals, ConcurrentHashSet presented the worst results, consuming 1.23x more energy and 1.14x more time than Collections.synchronizedSet() on System#1 (1.31x more energy and 1.19x more time on System#2).

Another interesting result is observed with ConcurrentSkipListSet, which consumes only 1.31x less energy than a Collections.synchronizedSet() on removal operations on System#1, although it saves 4.25x in execution time. Such energy consumption overhead is also observed on System#2. Internally, ConcurrentSkipListSet relies on a ConcurrentSkipListMap, which is non-blocking, linearizable, and based on the compare-and-swap (CAS) operation. During traversal, this collection marks the "next" pointer to keep track of triples (predecessor, node, successor) in order to detect when and how to unlink deleted nodes. Also, because of the asynchronous nature of these maps, determining the current number of elements (used in the Iterator) requires a traversal of all elements.

These behaviors are susceptible to create the energy consumption overhead observed in Figure 4.3.

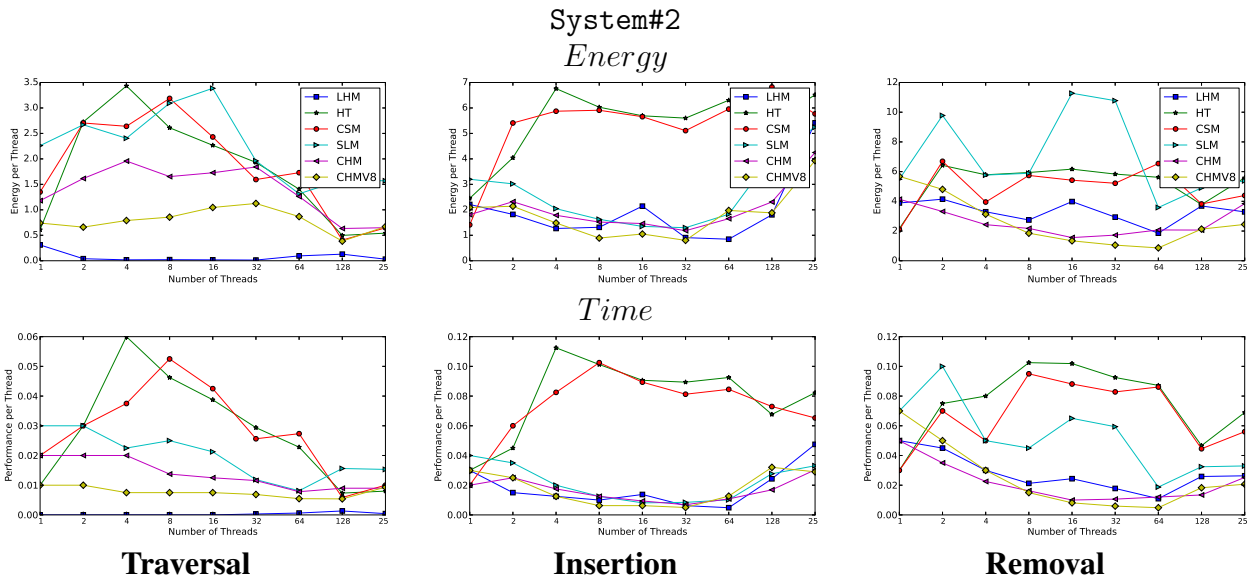


Figure 4.4: Energy consumption and execution time in the presence of concurrent threads (X axis: the number of threads, Y axis: energy consumption normalized against the number of element accesses, in joules per 100,000 elements)

4.3.2 Energy Behaviors with Different Number of Threads

In this group of experiments, we aim to answer **RQ3**. For this experiment, we chose `Map` implementations only, due to presence of both single-lock and high-performatic implementations. We vary the number of threads (1, 2, 4, 8, 16, 32, 64, 128, and 256 concurrent threads) and study how such variations impact energy consumption. An increment in the number of threads also increments the total number of elements inside the collection. Since each thread inserts 100,000 elements, when performing with one thread, the total number of elements is also 100,000. When performing with 2 threads, the final number of elements is 200,000, and so on. To give an impression on how `Map` implementations scale in the presence of multiple threads, Figure 4.4 demonstrates the effect of different thread accesses on benchmarks.

In this figure, each data point is normalized by the number of threads, so it represents the energy consumption per thread, per configuration. Generally speaking, `Hashtable`

and Collections.synchronizedMap() scale up well. For instance, we observed a great increment of energy consumption when using Collections.synchronizedMap() when we move from 32 to 64 threads performing traversals, but this trend can also observed for insertions and removals. Still on traversals, all Map implementations greatly increase the energy consumed as we add more threads. Also, all thread-safe implementations have their own "15 minutes of fame". Despite the highly complex landscape, some patterns do seem to recur. For instance, even though ConcurrentHashMapV8 provides the best scalability among the thread-safe collection implementations, it still consumes about 11.6x more energy than the non-thread-safe implementation. However, the most interesting fact is the peak of ConcurrentSkipListMap, when performing with 128 and 256 threads. As discussed earlier, during traversal, ConcurrentSkipListMap marks or unlinks a node with null value from its predecessor (the map uses the nullness of value fields to indicate deletion). Such mark is a compare-and-set operation, and happens every time it finds a null node. When this operation fails, it forces a re-traversal from caller.

For insertions, we observed a great disparity; while Hashtable and Collections.synchronizedMap() scale up well, ConcurrentSkipListMap, ConcurrentHashMap and ConcurrentHashMapV8 scale up very well. One particular characteristic about ConcurrentHashMapV8 is that the insertion of the first element in an empty map employs compare-and-set operations. Other update operations (insert, delete, and replace) require locks. Locking support for these locks relies on builtin "synchronized" monitors. When performing using from 1 to 32 threads, they have energy and performance behaviors similar to the non-thread-safe implementation. Such behavior was previously discussed in Figure 4.2.

For removals, interestingly, both ConcurrentHashMap and ConcurrentHashMapV8 scale better than all other implementations, even the non-thread-safe implementation, LinkedHashMap. ConcurrentSkipListMap, on the

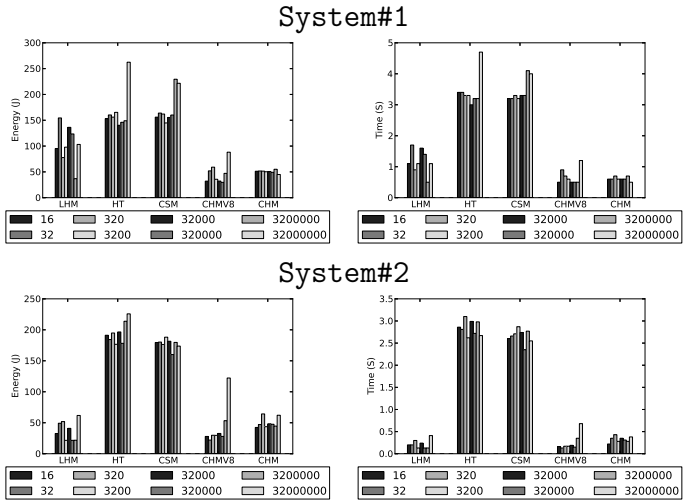


Figure 4.5: Energy and performance variations with different initial capacities.

other hand, presents the worst scenario, in particular with 16, 32 and 128 threads, even when compared to the single-lock implementations, such as `Hashtable` and `Collections.synchronizedMap()`.

4.3.3 Collection configurations and usages

We now focus on **RQ4**, studying the impact of different collection configurations and usage patterns on program energy behaviors. The `Map` implementations have two important "tuning knobs": the *initial capacity* and *load factor*. The capacity is the total number of elements inside a `Map` and the initial capacity is the capacity at the time the `Map` is created. The default initial capacity of the `Map` implementations is only 16 locations. We report a set of experiments where we configured the initial capacity to be 32, 320, 3,200, 32,000, 320,000, and 3,200,000 elements — the last one is the total number of elements that we insert in a collection. Figure 4.5 shows how energy consumption behaves using these different initial capacity configurations.

As we can observe from this figure, the results can vary greatly when using different initial capacities, in terms of both energy consumption and execution time. The

most evident cases are when performing with a high initial capacity in `Hashtable` and `ConcurrentHashMap`. `ConcurrentHashMapV8`, on the other hand, presents the least variation on energy consumption.

The other tuning knob is the load factor. It is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of elements inside a `Map` exceeds the product of the load factor and the current capacity, the hash table is rehashed; that is, its internal structure is rebuilt. The default load factor value in most `Map` implementation is 0.75. It means that, using initial capacity as 16, and the load factor as 0.75, the product of capacity is 12 (16 * 0.75 = 12). Thus, after inserting the 12th key, the new map capacity after rehashing will be 32. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. Figure 4.6 shows how energy consumption behaves using different load factors configurations. We perform these experiments only with insertion operations[5].

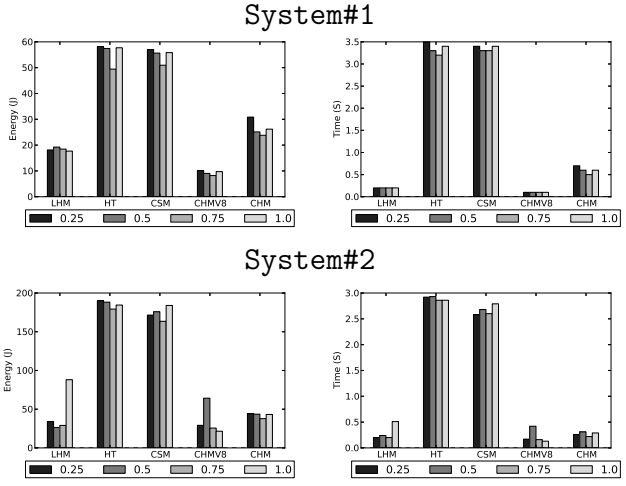


Figure 4.6: Energy and performance variations with different load factors.

From this figure we can observe that, albeit small, the load factor also influences

[5]We did not performed experiments with `ConcurrentSkipListMap` because it does not provide access to initial capacity and load factor variables.

both energy consumption and time. For instance, when using a load factor of 0.25, we observed the most energy inefficient results on `System#1`, except in one case (the energy consumption of `LinkedHashMap`). On `System#2`, the 0.25 configuration was the worst in three out of 5 of the benchmarks. We believe they is due to the successive rehashing operations that must occur. Generally speaking, the default load factor (.75) offers a good tradeoff between performance, energy, and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry, which can reflect in most of the `Map` operations, including `get()` and `put()`). It is possible to observe this cost when using a load factor of 1.0, which means that the map will be only rehashed when the number of current elements reaches the current maximum size. The maximum variation was found when performing operations on a `Hastable` on `System#1`, in the default load factor, achieving 1.17x better energy consumption over the 0.25 configuration, and 1.09x in execution time.

4.3.4 The Devil is in the Details

In this section we further analyze some implementation details that can greatly increase energy consumption.

Upper bound limit. We also observed that, on traversal and insertion operations, when the upper bound limit needs to be computed in each iteration, for instance, when using

```
for(int i=0; i<list.size(); i++) { ... }
```

the `Vector` implementation consumed about twice as much as it consumed when this limit is computed only once on (1.98x more energy and 1.96x more time), for instance, when using

```
int size = list.size();
for(int i=0; i < size; i++) { ... }
```

When this limit is computed beforehand, energy consumption and time drop by half. Such behavior is observed on both `System#1` and `System#2`. We believe it happens mainly because for each loop iteration, the current thread needs to fetch the `list.size()` variable from memory, which would incur in some cache misses. When initializing a `size` variable close to the loop statement, we believe that such variable will be stored in a near memory location, and thus, can be fetched all together with the remaining data. Using this finding, well-known IDEs, such as Eclipse and IntelliJ, can take advantage of it and implement refactoring suggestions for developers. Currently, the Eclipse IDE does not provide such feature. Also, recent studies have shown that programmers are more likely to follow IDE tips [64]. One concern, however, is related to removal operations. Since removal on `Lists` shift any subsequent elements to the left, if the limit is computed beforehand, the `i++` operation will skip one element.

Enhanced for loop. We also analyzed traversal operations when the programmer iterates using an *enhanced for loop*, for instance, when using

```
for (String e: list) { ... }
```

which is translated to an `Iterator` at compile time. Figure 4.7 shows the results. In this configuration, `Vector` need to synchronize in two different moments: during the creation of the `Iterator` object, and in every call of the `next()` method. By contrast, the `Collections.synchronizedList()` does not synchronize on the `Iterator`, and thus has similar performance and energy usage when compared to our baseline, `ArrayList`. On `System#1`, energy decreased from 37.07J to 2.65J, whereas time decreased from 0.81 to 0.10. According to the `Collections.synchronizedList()` documentation, the programmer must ensure external synchronization when using `Iterator`.

Removal on objects. When using `Lists`, instead of perform removals based on the indexes, one can perform removals based on object instances, for instance, when using

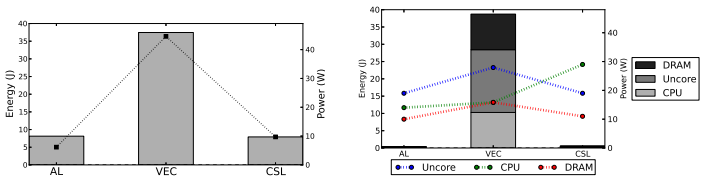


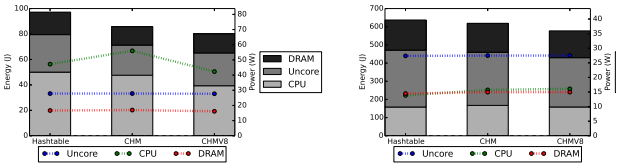
Figure 4.7: Traversal operations using the `get()` method. We use the same abbreviations of Figure 4.1.

```
for (int i = 0; i < threads; i++) {
  for (int j = 0; j < list.size(); j++) {
    boolean b = list.remove(i + "-" + j);
} }
```

we observed an increment on energy consumption of 39.21% on `System#1` (32.28% on execution time). This additional overhead is due to the traversal needed for this operations. Since the collection does not know in which position the given object is placed, it needs to traversal and compare each element until it finds the object – or until the collection ends.

4.4 Case Study

In this section we apply our findings into real-world benchmarks. We select benchmarks that made several use of `Hashtables`. We chose the `Hashtable` implementation because it presents one of the greatest different in terms of energy consumption, when compared to its newest version, `ConcurrentHashMap` (see Figure 4.2). The first selected benchmark is XALAN, from the well-known DaCapo suite [12]. This benchmark transforms XML documents into HTML. It performs reads and writes from input/output channels, and it has 170,572 lines of Java code. We chose this benchmark because it employs more than 300 `Hashtables`.



XALAN TOMCAT

Figure 4.8: Energy and performance variations with different initial capacities.

Since the DaCapo framework is not under active development, we used the BOA infrastructure [32] to select *active*, *i.e.*, have at least one commit in the last 12 months, *non-trivial*, *i.e.*, have at least 500 commits in the whole software history, benchmarks that use at least 50 `Hashtables`. We found a surprisingly number of 151 projects that fit on this criteria. Among the projects, we selected TOMCAT, which is an open-source web server. In the selected version 8.5, which is one of the latest one, it has 188,645 lines of Java code. Also, one of its previous version (6.0) is also part of the DaCapo framework. In this study we analyzed the most recent one.

For each benchmark, we performed our experiments by changing the `Hashtable` instance to `ConcurrentHashMap` and `ConcurrentHashMapV8`. The rest of the source code remained unchanged. We ran the benchmarks on `System#1`, with 32 concurrent threads. Figure 4.8 shows the results.

The results here confirm some patterns from micro-benchmarking. Both XALAN and SUNFLOW present an improvement in energy consumption when migrating from `Hashtable` to `ConcurrentHashMap`. XALAN, in particular, presented an improvement of 12.21% when varying from `Hashtable` to `ConcurrentHashMap`, and 17.82% when varying from `Hashtable` to `ConcurrentHashMapV8`. This is particularly due to its CPU intensive nature, which is also observed with the green line. SUNFLOW, on the other hand, spends most of its computational time with logging and network operations — which are intrinsically IO operations. Despite of it, it is still capable of yielding a 9.32% energy saving when moving from `Hashtable` to `ConcurrentHashMapV8`.

These results suggests that even small changes have the potential of improving the energy consumption of a non-trivial software system. It is important to observe that the degree of energy saving is related to the degree of use of an inefficient collection (e.g. `Hashtable`). Therefore, benchmarks that make heavily use of single lock based collections are more likely to have high energy gains.

Finally, although both `Hashtable` and `ConcurrentHashMap` share the same interface, it is not always straightforward to perform this transformation, due to at least three reasons:

1. `Hashtable` and `ConcurrentHashMap` do not obey the same hierarchy. For instance, `Hashtable` inherits from `Dictionary` and implements `Cloenable`, while `ConcurrentHashMap` does not. It means that operations such as `hash.clone()` raises a compile error when changing the instance of the `hash` variable.

2. Third party libraries often require implementations instead of interfaces. If a method is expecting a `Hashtable` instead of, say, a `Map`, a `ConcurrentHashMap` needs to be converted to `Hashtable`, decreasing its effectiveness.

3. Programmers often use methods that are present only in the concrete implementation, not in the interface (e.g. `rehash`). This creates a strong tie between the client code with the concrete implementation, hampering the transition from `Hashtable` to `ConcurrentHashMap`.

4.5 Threat to Validity

Internal factors. First, the elements which we used are not randomly generated. We chose to not use random number generators because they can greatly impact the performance and energy consumption of our benchmarks. We observed standard deviation of over 70% between two executions when using the random number generators. We mitigate this problem by combining the index of the for loop plus the thread id that inserted

the element. This approach also prevents compiler optimizations that may happen when using only the index of the for loop as the element to be inserted in the collection.

External factors. First, our results are limited by our selection of benchmarks. Nonetheless, our corpus spans a wide spectrum of collections, ranging from lists, sets, and maps. Second, there are other possible collections implementations beyond the scope of this chapter. With our methodology, we expect similar analysis can be conducted by others. 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 [74]. We experienced differences in standard deviation of over 30% when comparing the warmup run (first 3 executions) and later runs, but less than 5% when comparing the last 3 runs.

4.6 Chapter Conclusions

In this chapter, we presented an empirical study that investigates the impacts of using different collections on energy usage. As subjects for the study, we analyzed the main methods of 16 types of commonly used collection (13 thread-safe, 3 non-thread-safe) in the Java language. Some of the findings of this study include: (1) Different operations of the same implementation also have different energy footprints. For example, a removal operation in a `ConcurrentSkipListMap` can consume more than 4 times of energy than an insertion to the same collection. Also, for `CopyOnWriteArraySet`, an insertion consumes three order of magnitude more than a read. (2) Small changes have the potential of improving the energy consumption by 2x, in our micro-benchmarks, and by 10%, in our real-world benchmarks.

Chapter 5

Vincent: Energy-Efficient Hot Methods for Java Applications

5.1 Introduction

The carbon footprint of data centers has recently received significant scrutiny [63]. After mobile workloads, server-type workloads once again place energy-efficient and sustainable computing in the spotlight. This cross-sectioned design goal can be addressed at many layers of the computing stack. Among them, a relatively less explored approach is to study the energy impact of *managed runtimes*, a middle layer between high-level applications and low-level systems. Outnumbered by approaches from its neighbor layers, existing runtime-level system designs [23, 19, 1, 45, 78] and empirical studies [91, 33, 81, 73, 70, 38, 75, 48] nonetheless demonstrate the critical role of this layer on energy optimization. Relative to lower-layer techniques on hardware design (e.g., [21]) and OS design (e.g., [93]), a runtime-level approach has the benefit of guiding energy optimization with runtime-specific information unavailable to the underlying systems. Relative to higher-layer techniques such as energy-aware programming models [85, 8, 80, 43, 25, 9, 49, 42, 61, 96, 17], a runtime-level approach can work with legacy code written in existing programming models and languages, arguably easier for adoption. In a nutshell, the managed runtime — strategically positioned in between the lower layers and the higher layers — can often combine the benefits of both sides of its neighbors on the computing stack.

At their essence, all runtime-based approaches are motivated by the same question: what information uniquely available in the runtime can be harvested to guide energy optimization? As examples, existing efforts have relied on thread and synchronization states (e.g., [1]), uust-in-time (JIT) compilation strategies (e.g., [91]), and garbage collector (GC) designs (e.g., [45]) to inform energy optimization.

5.1.1 VINCENT

VINCENT is a novel energy optimizer at the level of the JVM. It relies on two simple facts of the JVM: (i) the JVM is aware of the boundary of programming abstractions such as methods; (ii) the JVM is aware of how often a method is used. Both pieces of information are readily available among existing JVMs, good news for the adoption of our approach. To the best of our knowledge however, this is the first time they are used to guide energy optimization.

The premise of VINCENT is that each method as a logical unit of the program behavior can serve as an ideal granularity for energy optimization. For example, the method `Matrix4.transformP` in a ray-tracing benchmark `sunflow` may carve out the boundary of a CPU-intensive computation, and the method `PSStream.write` in a file processing benchmark `fop` may demarcate an I/O-intensive computation. Well known in classic techniques such as DVFS [44, 15], understanding *phased behaviors* [83, 84, 46] — an application may go through phases of different levels of CPU intensity — is vital for energy optimization. For example, running an I/O-intensive program fragment at a lower CPU frequency can often save energy without hampering performance.

VINCENT features *method-grained energy optimization*: it associates the boundary of energy phases with the boundary of methods for energy optimization. Operationally, it relies on profiling to assign desirable CPU frequencies to the most energy-consuming methods, and dynamically adjust CPU frequencies accordingly in production runs. Only *hot methods* deemed by the JVM are candidates for energy optimization. This

design decision is aligned with the fact that hot methods are frequently executed, and any improvement to their energy behavior may have an amplified effect. Practically, focusing on hot methods also allows VINCENT to piggyback on existing JIT design, where information on hot methods is readily available.

In contrast, the state-of-the-art approach for phase identification relies on dynamic monitoring system-level information, such as hardware performance counters (e.g., cache misses). A classic example of this approach is the `ONDEMAND` governor, often the default power manager across Linux versions. This governor continuously predicts the level of CPU intensity demanded by the application and dynamically adjusts the CPU frequency to meet the demand. This approach is blind to the logical structure of the running application, and fundamentally reactive: it uses the level of CPU intensity at the *current* time interval to set the CPU frequency for the *next* time interval. Whereas the reactive approach is effective when the application is stable within a phase, it loses its effectiveness when phase changes.

5.1.2 Contributions

VINCENT a novel feedback-directed energy optimizer with a focus on applying DVFS to hot methods. To the best of our knowledge, it is the only runtime-level solution that performs DVFS at the granularity of methods. This chapter makes the following contributions:

- the novel philosophy of method-grained energy optimization at the level of managed runtimes;
- the specification of the feedback-directed energy optimization through instrumentation inside the JVM;
- the implementation and evaluation of method-grained DVFS on top of JikesRVM [3, 2], which demonstrates its effectiveness in improving the energy efficiency of

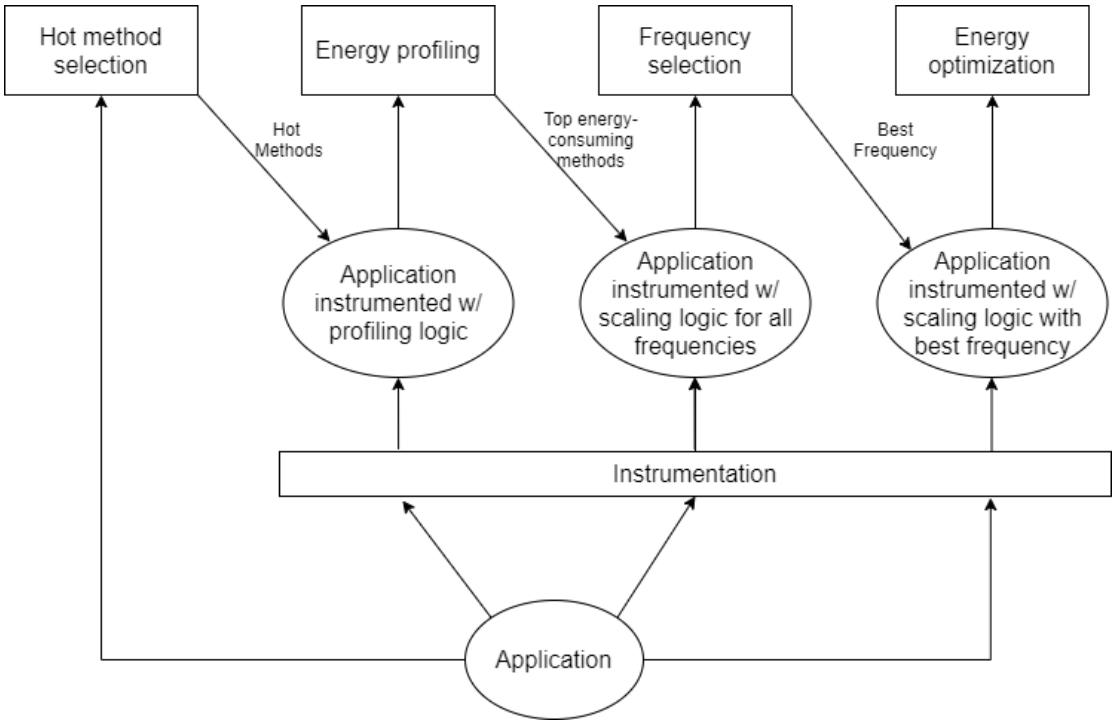


Figure 5.1: VINCENT Design and Workflow

Java applications against existing power managers relying on dynamic monitoring.

VINCENT is an open-source project. Its source code and all raw experimental data can be found online [1].

5.2 VINCENT Design

In this section, we describe the design of VINCENT, with a high-level description in § 5.2.1, followed by an algorithm specification in § 5.2.2.

[1] The Middleware'21 CFP explicitly discourages any URL sharing, so we elide our anonymous site here to err on the safe side. Reviewers can request the link through the PC chairs as intermediaries at any time during the review period, if that is approved by the PC chairs.

5.2.1 VINCENT Design

The system components and the workflow of VINCENT are shown in Fig. 5.1. On the high level, VINCENT is a feedback-directed optimizer that conceptually consists of 4 passes:

- **Hot Method Selection**: VINCENT first obtains a list of *hot methods*.
- **Energy Profiling**: VINCENT profiles the energy consumption of hot methods under the default `ONDEMAND` governor. It ranks their energy consumption, and reports a list of *top energy-consuming methods* as the output of this pass.
- **Frequency Selection**: For each top energy-consuming method, VINCENT observes the energy consumption of the application when the execution of this method is scaled to each CPU frequency, which we call a *configuration*. VINCENT determines the best CPU frequency for each top energy-consuming method by ranking the energy efficiency of different configurations.
- **Energy Optimization**: VINCENT runs the application when the execution of each top energy-consuming method is scaled to its best CPU frequency.

The workflow described here is conceptual. For instance, the VINCENT system combines hot method selection and energy profiling in one single run, with the former achieved by reusing the results identified by JIT. The core design elements are the algorithms for energy profiling (used by passes 1 and 2) and method-based scaling (used by passes 3 and 4). These details will be provided in § 5.2.2.

The key observation over this workflow is that VINCENT places the spotlight on *methods*: in each of the workflow tasks, the unit of processing — be it selection, profiling, or optimization — is *at the granularity of methods*. For this reason, despite that the general workflow is expected as a feedback-directed optimizer, it represents a sig-

nificant departure from the state of the art of optimizing the energy consumption of applications.

Algorithm 1 Thread Bookkeeping and Timer Thread Loop

```
 1: typedef T {
 2:     vtimer: int // VINCENT timer
 3:     skipCount: int // # of invocations to skip
 4:     sampleCount: int // # of samples to collect
 5:     edata: EDATA // energy profiling data
 6:     gov: GOVERNOR // saved governor
 7:     freq: FREQ // saved CPU frequency
 8: }
 9: const EPOCH // unit of time in the JVM
10: const SKIPNUM // skipped samples between collections
11: const SAMPLENUM // collected samples per timer interval
 1: ts: T[THREADNUM] // running threads
 2: procedure TIMER
 3:     while TRUE do
 4:         SLEEP(EPOCH)
 5:         for each t ∈ ts do
 6:             t.vtimer++
 7:         end for
 8:     end while
 9: end procedure
```

5.2.2 VINCENT Specification

We now specify the algorithm implemented by VINCENT. We first describe the top-level thread bookkeeping in § 5.2.2. For the key system components described in the previous section, hot method selection is largely achieved by building on top of existing JIT infrastructure, a topic we will defer as an implementation detail (§ 5.3). The rest of the system components are enabled by instrumentation, which we describe in this section as the profiling algorithm (§ 5.2.2) and the scaling algorithm (§ 5.2.2).

Algorithm 2 Profiling Algorithm

```
 1: typedef LOG {
 2:   mn: MNAME // method name
 3:   edata: EDATA // data
 4: }
 5: typedef CVAL enum { TAKE, SKIP, LAST }
 6: typedef EDATA float
 7: const PN // profiling timer factor (relative to EPOCH)
 8: l: LOG[LOGNUM]
 9: procedure PROLOGUEPROFILE()
10:     t ← CURRENTTHREAD()
11:     if COUNTER(t, PN) == TAKE then
12:         t.edata ← READENERGY()
13:     end if
14:     if COUNTER(t, PN) == LAST then
15:         t.edata ← ⊥
16:     end if
17: end procedure
18: procedure EPILOGUEPROFILE()
19:     t ← CURRENTTHREAD()
20:     if COUNTER(t, PN) == TAKE or LAST then
21:         energy ← READENERGY()
22:         if t.edata ≠ ⊥ then
23:             l ←+ LOG(THISM, DIFF(energy, t.edata))
24:         else
25:             t.edata ← energy
26:         end if
27:     end if
28:     if COUNTER(t, PN) == LAST then
29:         t.edata ← ⊥
30:     end if
31: end procedure
32: function COUNTER(t: T, factor: int): CVAL
33:     if t.vtimer >= factor then
34:         t.skipCount ← t.skipCount − 1
35:         if t.skipCount == 0 then
36:             t.skipCount ← SKIPNUM
37:             t.sampleCount ← t.sampleCount − 1
38:             if t.sampleCount == 0 then
39:                 t.vtimer ← 0
40:                 t.sampleCount ← SAMPLENUM
41:                 return LAST
42:             end if
43:             return TAKE
44:         end if
45:     end if
46:     return SKIP
47: end function
```

Algorithm 3 Scaling Algorithm

```
 1: enum GOVERNOR {USERSPACE, ONDEMAND, ...}
 2: const SN // scaling timer factor (relative to EPOCH)
 3: procedure PROLOGUESCALE(f : FREQ))
 4:     t ← CURRENTTHREAD()
 5:     if COUNTER(t, SN) == TAKE then
 6:         t.gov ← GETGOVERNOR()
 7:         if t.gov == USERSPACE then
 8:             t.freq ← GETFREQ()
 9:         else
10:             SETGOVERNOR(USERSPACE)
11:         end if
12:         SETFREQ(f)
13:     end if
14:     if COUNTER(t, SN) == LAST then
15:         SETGOVERNOR(ONDEMAND)
16:     end if
17: end procedure
18: procedure EPILOGUESCALE()
19:     t ← CURRENTTHREAD()
20:     if COUNTER(t, SN) == TAKE then
21:         if t.gov ≠ ⊥ then
22:             SETGOVERNOR(t.gov)
23:             if t.gov == USERSPACE then
24:                 SETFREQ(t.freq)
25:             end if
26:         end if
27:     end if
28:     if COUNTER(t, SN) == LAST then
29:         SETGOVERNOR(ONDEMAND)
30:     end if
31: end procedure
```

Thread Bookkeeping

Algorithm 1 overviews the bookkeeping in a multi-threading environment. Here, all threads visible to the JVM (other than the timer thread itself) are maintained in a global structure *ts*, a collection of threads of type T. Each thread contains thread-local bookkeeping information; in particular, note that *vtimer* manages the elapse of time, incremented by the unit `EPOCH` As profiling and scaling belong to different passes of VINCENT and do not share the same runtime, *vtimer* is used for both runs. The thread-local fields used only for profiling and those only for scaling are illustrated with GREEN box and RED box respectively. The specific meanings of the constants and the fields in T other than *vtimer* will be detailed in the rest of this section.

The timer thread is defined as an infinite loop. When the JVM timer interrupt happens at the rate of `EPOCH`, the *vtimer* associated with each thread is incremented.

Profiling Instrumentation

Recall that the goal of profiling is to identify the top energy-consuming methods. The raw energy consumption maintained by the underlying hardware (see § 5.3) is *accumulative*, i.e., reported as monotonically increasing values. To determine the energy consumption of a method, we conceptually need to "diff" the raw energy reading obtained at the beginning of the method execution, and one obtained at the end of the method execution.

Challenges and Strawman Solutions A root hurdle against energy profiling is that obtaining a raw energy reading from the underlying hardware incurs a non-trivial overhead, often taking tens of microseconds to complete. As a result, standard solutions known to be effective for execution time profiling may not be ideal for energy profiling, which we now briefly review.

A strawman solution naively adapted from execution time profiling is to instru-

ment the begin (i.e., *prologue* in JVM terminology) and the end (i.e., *epilogue* in JVM terminology) of every hot method, where a raw energy reading is taken each time the prologue and epilogue is encountered. The energy consumption of a method can thus be the difference between the two readings. Unfortunately, thanks to the non-trivial overhead with raw energy readings, this approach may incur prohibitively high overhead (10x-200x in our preliminary experiments), severely altering the program behavior. In other words, the instrumented run may produce the result no longer representative of the original benchmark's energy behavior. Oberve that even instrumenting each hot method "one at a time" does not solve the problem. The hot methods are "hot" for a reason: they are frequently called, and the per-call overhead may rapidly accumulate.

A second strawman solution is to perform sampling at fixed time intervals. For example, assume the JVM has just taken an energy sample of $90J$ at the beginning of its 100th time interval. After one time interval elapses, it takes another energy sample of $90.25J$, and the epilogue of a method is encountered. The approach can thus attribute $0.25J$ to that method. This approach however may lead to over-attribution: $0.25J$ is attributed to one method encountered at the end of the time interval, but many other methods may have contributed to the energy consumption during the interval. This sampling approach is widely used for execution time sampling, because precision can be improved by shortening the time interval. For energy profiling however, the room for shortening the time interval is limited due to the overhead of raw energy readings.

Delimited Sampling with VINCENT The energy profiler of VINCENT is a hybrid of the two strawman solutions above, which we call *delimited sampling*. Similar to the first strawman approach, VINCENT takes energy readings when the method prologue and the method epilogue are encountered, and computes the difference of the two. VINCENT however does not take energy readings at every encounter of the prologue or the epilogue. Instead, the number of energy readings at the method prologue/epilogue are *bounded for each interval*, similar to the second strawman approach.

As seen in Algorithm 2, each hot method is instrumented with a pair of methods, with PROLOGUEPROFILE inserted before the entry point of the method body, and EPILOGUEPROFILE inserted after each exit point of the method body. Auxiliary function READENERGY obtains a raw energy sample from the underlying hardware (a value of EDATA type). Binary function DIFF computes the difference of two raw energy samples, and function CURRENTTHREAD returns the current thread of the execution, of type `T`. Constant `THISM` is the name of the instrumented method, an implementation detail we clarify in § 5.3. Sampling happens within the function of COUNTER, which we describe shortly.

An observable reader may notice that we do not match the method name between the prologue and the epilogue. This is intentional: in a sampling-based approach, the method encountered during the prologue and one encountered during the epilogue may not match. The key insight here is that we are not attempting to replicate the first strawman approach, but to avoid the overattribution problem in the second strawman approach. The philosophy here is *refutation*: if a prologue (of any method) is encountered before the epilogue of the method m of our interest, we know the energy consumption incurred before the prologue encounter must not be due to m, thanks to how call stacks are structured.

Counter-based Sampling Our description so far can be *conceptually* viewed as taking two energy readings — one at the prologue and the other at the epilogue — for each time interval. VINCENT extends from this conceptual view by adopting counter-based sampling [4], allowing multiple (but still bounded) pairs of energy readings to be collected within a time interval. The benefit of counter-based sampling in improving the accuracy of sampling is well documented, especially for complex call graphs where methods are of variant lengths. Specific to energy optimization, this means that VINCENT cares about both longer but slightly less frequently invoked (but still hot) methods and shorter but more frequently invoked methods, as long as they incur high

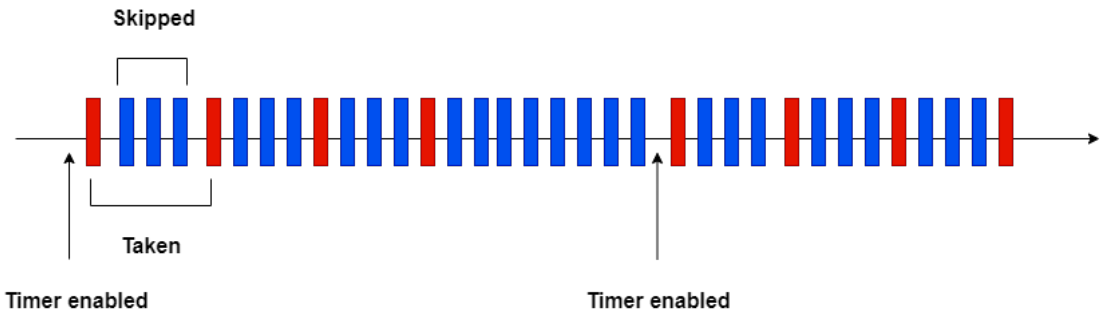


Figure 5.2: Counter-based Sampling where `SKIPNUM` = 3 and `SAMPLENUM` = 4

energy consumption.

As shown in Fig. 5.2, the counter-based approach alternates between taking samples and skipping samples within each time interval. This is achieved through maintaining two counters: the number of samples to take (`sampleCount` in our algorithm) and the number of samples to skip between two samples (`skippedCount` in our algorithm).

In Algortihm 2, counter-based sampling is captured by function COUNTER, at Lines 32-47. Here, the profiling time interval is set as `PN` $\times$ `EPOCH`; recall that *vtimer* is incremented at each VM `EPOCH`, so `PN` is the "slowdown" factor of profiling relative to the top-level timer loop. Constants `SAMPLENUM` and `SKIPNUM` represent the number of samples to take and skip, respectively, within each profiling time interval.

The COUNTER function may return one of the 3 values: `TAKE` (indicating a sample should be taken), `SKIP` (indicating a sample should not be taken), and `LAST` (indicating one last sample should be taken for each time interval). The `LAST` value plays a role of re-initializing the environment for the next time interval. For profling, this means to reset the *edata* field.

Scaling Instrumentation

Algorithm 3 defines the instrumentation-based algorithm for CPU scaling. Convenience function GETGOVERNOR retrieves the current governor (power manager) from the un-

BENCHMARK	ONDEMAND T	ONDEMAND E	POWERSAVE T	POWERSAVE E	PERFORMANCE T	PERFORMANCE E
sunflow	2539.80 ± 50.36	62.82 ± 1.81	3838.80 ± 49.29	67.63 ± 0.74	1879.00 ± 41.97	71.16 ± 2.23
pmd	2140.53 ± 116.00	42.11 ± 3.23	4594.33 ± 65.79	43.81 ± 2.37	2090.60 ± 104.51	46.02 ± 5.61
avrora	5459.87 ± 55.53	75.34 ± 23.99	7156.60 ± 47.30	76.44 ± 15.36	3340.93 ± 50.91	83.11 ± 13.64
jython	2740.13 ± 204.96	46.32 ± 4.52	6159.33 ± 178.22	61.03 ± 3.89	2695.87 ± 132.91	63.81 ± 5.81
fop	939.60 ± 108.65	19.52 ± 2.95	2043.40 ± 47.22	19.53 ± 1.68	929.93 ± 78.09	16.68 ± 3.67
antlr	1029.40 ± 84.35	18.45 ± 2.39	2166.73 ± 94.17	22.68 ± 1.18	996.20 ± 49.01	21.21 ± 2.04
bloat	3657.53 ± 145.75	62.35 ± 5.16	8157.20 ± 105.74	80.02 ± 3.61	3735.13 ± 151.93	93.22 ± 6.03
luindex	887.00 ± 45.19	14.49 ± 1.38	1710.00 ± 29.82	16.12 ± 1.06	852.73 ± 50.74	16.80 ± 1.53

Table 5.1: Benchmark Statistics (T indicates Time in milliseconds and E indicates energy consumption in joules.)

derlying system, which can either be USERSPACE (i.e., with frequencies manually set by the user) or ONDEMAND. Function SETGOVERNOR sets the governor to its argument value. Function GETFREQ retrieves the current CPU frequency, whereas SETFREQ sets the CPU frequency to its argument value.

Recall that the scaling instrumentation is used for VINCENT's passes of frequency selection or energy optimization. The instrumentation is only applied to the hot top-energy consuming methods. When the application is bootstrapped, VINCENT sets the governor to ONDEMAND. When a top energy-consuming method is encountered at its PROLOGUESCALE, the governor and the CPU frequency are set according to the need of frequency selection or energy optimization. At this point, the governor to be used is USERSPACE, *a la* the convention of Linux. VINCENT in addition preserves the governor/frequency context, i.e., the settings of governor/frequency before the PROLOGUESCALE is encountered. The EPILOGUESCALE recovers the preserved context.

Just as profiling, counter-based sampling is also at work during scaling. Note that profiling and scaling do not have to follow the same rate. Constant SN adjusts the rate for scaling. In addition, note that when we reach the LAST sample in each time interval, the governor is reset to ONDEMAND.

5.3 Implementation and Experimental Settings

Hardware/OS/VM Setup We evaluated VINCENT on a dual socket Intel E5-2630 v4 2.20 GHz CPU server, with 10 cores in each socket and 64 DDR4 RAM. Hyper-threading is enabled. In total, we have 20 physical cores and 40 virtual cores. The machine runs Debian 4.9 OS, Linux kernel 4.9. For profiling based on individual CPU frequencies and the DVFS-based optimization, we explored all CPU frequencies that can be stably supported by our hardware, ranging from 2.2GHz to 1.2GHz, with the decrement of 0.1Ghz. For the rest of the chapter, we use `F1` to refer to 2.2Ghz, `F2` for 2.1Ghz, `F3` for 2.0Ghz, . . . , `F11` for 1.2Ghz.

We rely on RAPL [27] and a Java-based tool jRAPL [59] to obtain raw energy readings. The energy consumption reported by RAPL is accumulative. Each energy sample – as shown of the `EDATA` type in the algorithm specification — is the sum of energy readings from all sockets; and each socket-wise reading consists of energy consumption for the CPU cores, the uncore (cache, TLB, etc), and the DRAM. Specific to our environment, this means we collect and sum up $2 \times 3 = 6$ raw readings for each energy sample.

We implemented VINCENT on JikesRVM version 3.1.4. The hot method selection is built on top of the Adaptive Optimization System (AOS) [5] of JikesRVM.

Hot Method Selection We rely on the JIT component of JikesRVM for hot method selection, where the runtime compiler is set as the optimizing compiler. The hot method selection process in JikesRVM is adaptive, so is the process of profiling based on them. Whenever a new method is identified as hot, VINCENT's profiler will instrument it dynamically and perform its profiling upon identification.

One design consideration was whether we should exclude very short methods such as getters and setters from the hot methods. Intuitively, if such methods were subjected to scaling, the scaling overhead might well offset the benefit of setting the method to the

desirable frequency. Fortunately, the top energy-consuming methods identified by VINCENT's energy profiler (as seen in § 5.4) appear to rarely include them. In other words, these very short methods, even though hot from the perspective of invocation counts, rarely accumulate enough energy consumption to become top energy-consuming methods. As a result, we choose to keep our design simple, and do not alter the hot method selection logic in JikesRVM.

Algorithm Implementation The prologue and epilogue program fragments for profiling and optimization we specified in the previous section are inserted as IR instrumentation through `hir2lir`. Recall that we need to obtain the "this method" information (`THISM` in Algorithm 2). This is implemented through the optimizing compiler. As the method signature is carried with the IR, VINCENT stores the method information when instrumentation is added.

In the top-level timer loop, the interval `EPOCH` is identical to the default time interval of AOS, 4ms. Unless otherwise noted, we set the time interval for both profiling and scaling at 8ms, i.e., `PN` = 2 and `SN` = 2. Within each time interval, counter-based sampling is at work for both profiling and scaling. Unless otherwise noted, parameter `SAMPLENUM` is set at 16. In both scenarios, `SKIPNUM` = 7. The fact the skipped number of samples should be an odd number is well known in counter-based sampling [4].

All energy readings are stored as a C array and printed after the experiments end for posterior analysis.

Benchmarking and Experimental Setup We evaluate VINCENT with benchmarks in the Dacapo suite [13], arguably the most widely used benchmark suite for multi-threaded Java applications. Our benchmarks by default come from the last version of Dacapo known to work with JikesRVM, Dacapo MR2. Dacapo has a more recent release, Dacapo 9.12-bach, and we successfully ported some benchmarks in this version — `sunflow`, `luindex`, and `avrora` specifically — to work with JikesRVM. The rest

of porting was unsuccessful because JikesRVM cannot support some advanced Java features that appeared in the later versions of benchmarks.

Baselines We choose 3 baselines that rely on dynamic monitoring for energy optimizations; all 3 are available in Linux as governors. Among them, `ONDEMAND` is the most widely used governor, the default of many Linux installations. `POWERSAVE` and `PERFORMANCE` are variants of `ONDEMAND` in Linux, with a bias toward selecting lower CPU frequencies, and a bias toward selecting higher CPU frequencies, respectively. The baseline execution time and energy consumption of each benchmark while running with the 3 Linux governors can be found in Table 5.1.

Throughout this chapter, all experiment results are collected by running each benchmark 20 times in a hot run, and reporting the average of the last 15 runs.

5.4 VINCENT Evaluation

In this section, we evaluate the effectiveness of VINCENT. We aim at answering the following questions: (**Q1**) Do different methods and different choices of CPU frequencies for DVFS have different impacts on energy consumption and EDP? (**Q2**) What impact do sampling settings in the effectiveness of VINCENT? (**Q3**) How is VINCENT compared against different existing power management strategies? We answer each of these questions in each subsection below.

5.4.1 Method-Grained Energy Optimization

Energy Profiling The first step of the VINCENT lifecycle is energy profiling. Fig. 5.3 shows the top-5 energy-consuming methods for selected benchmarks. For example, it shows the method `PSRenderer.renderDisplaySpace` consumes around 33% of energy consumption due to hot methods for `fop`. These top-ranked methods appear to be consistent with our manual inspection of the benchmarks.

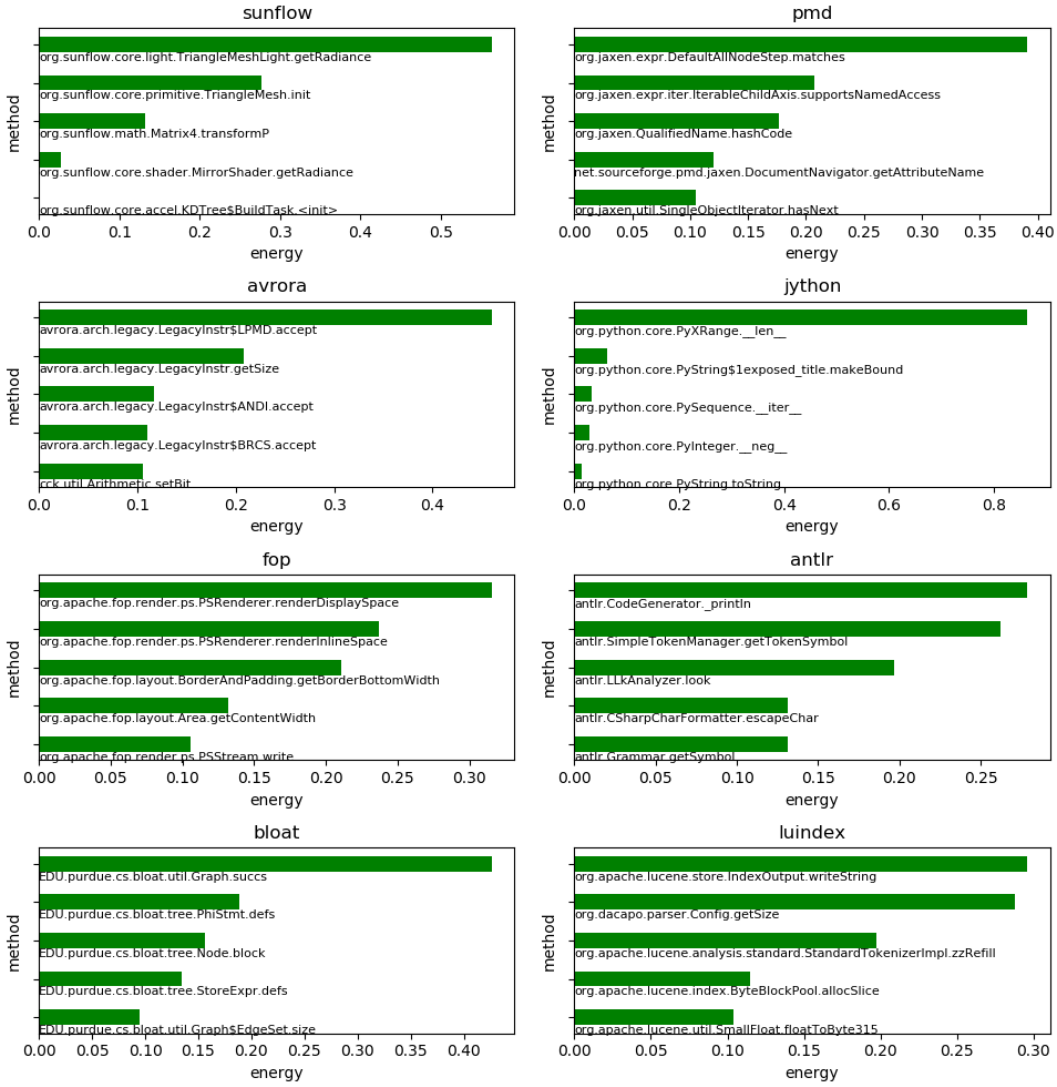


Figure 5.3: Top Energy-Consuming Methods According to VINCENT Energy Profiling (For each benchmark, the top-5 energy consuming methods are shown. The length of each bar represents its normalized energy consumption relative to the overall energy consumption. The name below each bar is the name of the top consuming method.

Very short methods indeed appear in the top energy-consuming methods, but this is relatively rare. For example, `pmd`'s top-consuming method, `DefaultAllNodeStep.matches` only contains a simple boolean return as its method body. As we shall see soon, these methods are indeed unfriendly for DVFS (see § 5.3). That being said, the vast majority of methods identified by VINCENT's profiling phase are indeed methods of reasonable length (in terms of execution time), and they respond to DVFS.

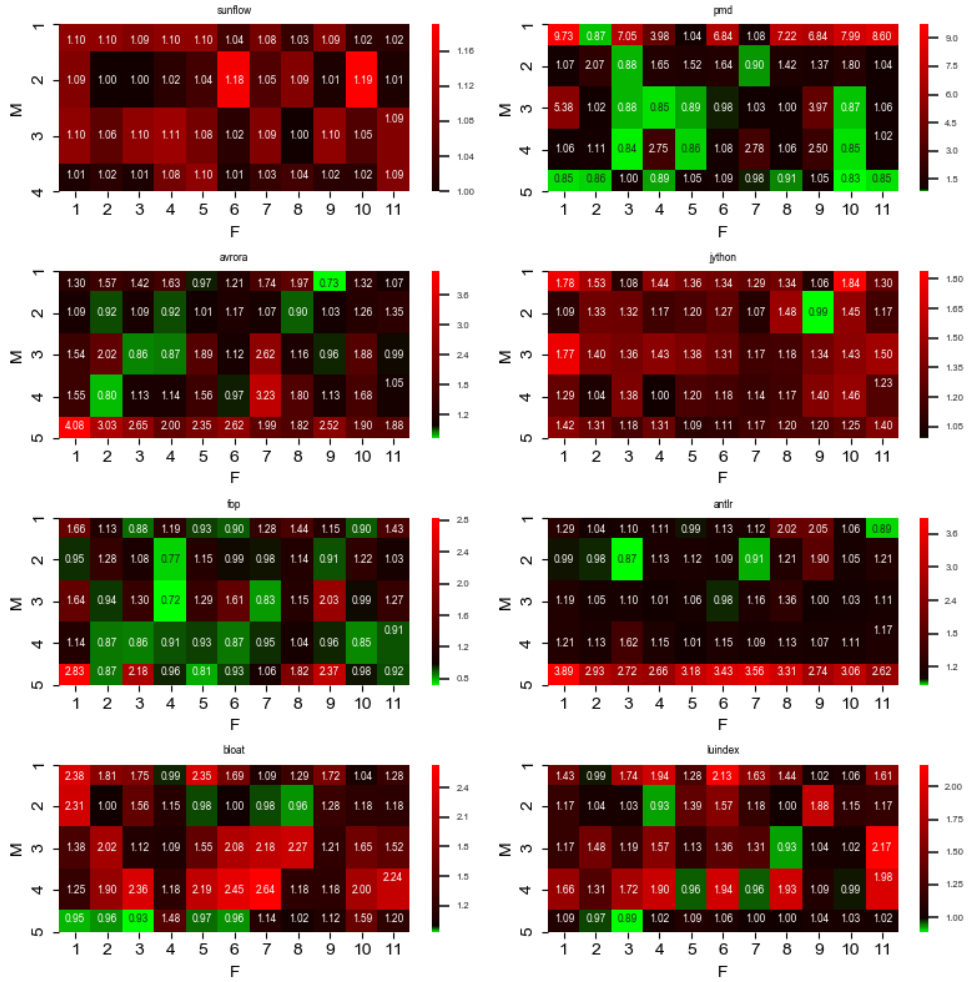


Figure 5.4: VINCENT Energy Consumption Normalized Against the ONDEMAND Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has energy consumption v, normalized against that of the ONDEMAND run. If $v < 1$, the VINCENT incurs less energy than the ONDEMAND run.)

For VINCENT, the energy profiling results are intermediate. The effectiveness of identifying top energy-consuming methods will impact the effectiveness of energy optimization, which we describe next.

The Impact on Energy Consumption We now describe the effectiveness of VINCENT energy optimization against the ONDEMAND baseline, i.e., when the application is running with the ONDEMAND governor in place throughout its execution. We show the

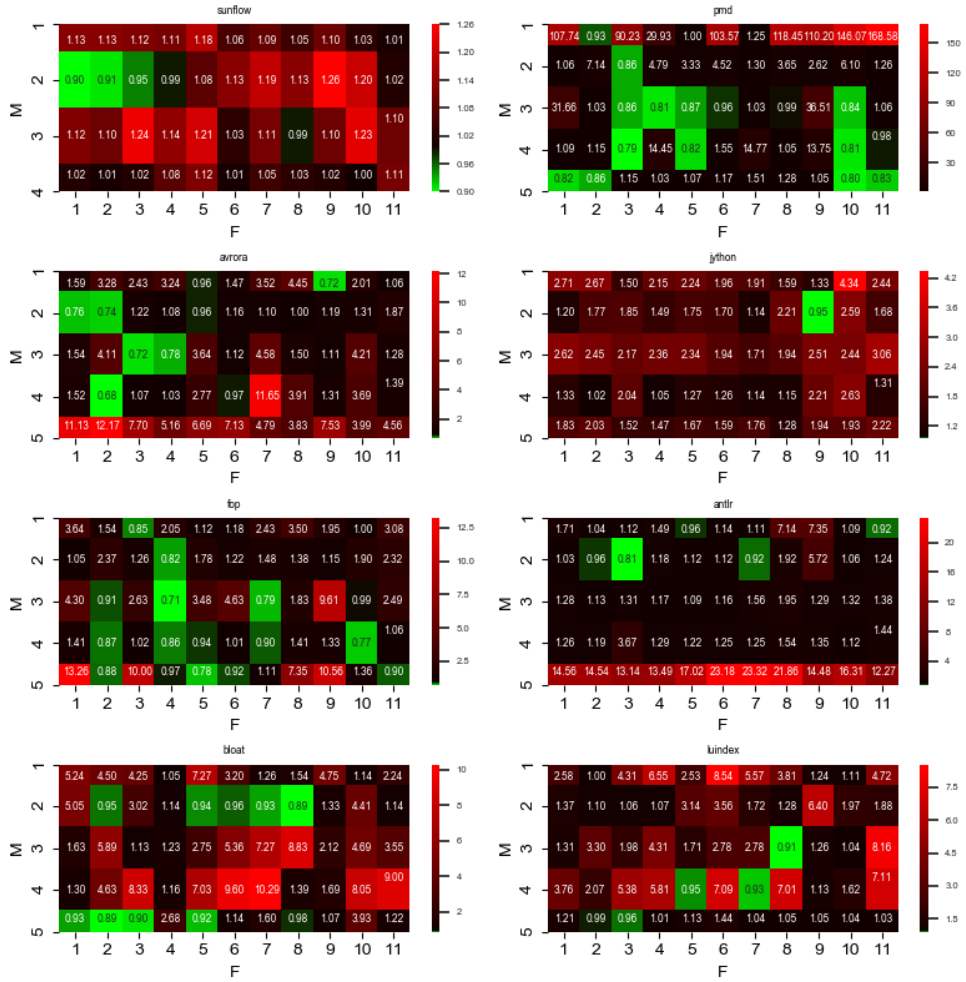


Figure 5.5: VINCENT EDP Normalized Against the ONDEMAND Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has EDP v, normalized against that of the ONDEMAND run. If $v < 1$, the VINCENT is more energy-efficient than the ONDEMAND run w.r.t. EDP.)

energy consumption results of VINCENT in Fig. 5.4. In each figure, a heat map is used for each benchmark to show the result of running it with VINCENT where one of the top-consuming methods (Y axis) is subjected to DVFS at a particular frequency level (X axis). The value carried in each cell in the heatmap is normalized against the ONDEMAND run. Each green cell indicates an energy-friendly configuration, i.e., the energy consumption for VINCENT is smaller than that of the ONDEMAND run. All benchmarks are shown with 5 top-consuming methods except `sunflow`, which we only show 4 because

the 5th energy-consuming method consumes little energy, as shown in Fig. 5.3.

Method-grained energy optimization is effective in reducing energy consumption for all benchmarks (but one): there exists at least one configuration within the benchmark whose normalized energy consumption is less than 1. For example, when VINCENT runs `antlr` at the third highest CPU frequency (2.0Ghz) for its second most energy-consuming method, `SimpleTokenManager.getTokenSymbol`, the normalized EDP is 0.87, indicating that VINCENT can save energy by 13% than running `antlr` with the `ONDEMAND` governor. As each green cell in the heatmap indicates a configuration with energy savings relative to `ONDEMAND`, energy optimization opportunities widely exist across benchmarks.

Indeed, not every benchmark can benefit from method-grained energy optimization. Benchmark `sunflow` has all normalized energy consumption values greater than 1 for all VINCENT configurations, indicating the `ONDEMAND` execution indeed consumes less energy than VINCENT. The same applies to nearly all `jython` configurations. Both benchmarks are consistently CPU-intensive, meaning that the `ONDEMAND` governor is likely to operate the CPUs at the highest frequencies at most times. In this case, DVFS has limited choices: if it scales the CPU down, the CPU-intensive application may run significantly slower, negatively impacting energy consumption because the latter is the accumulated power consumption *over time*; if it scales the CPU up, the power consumption may increase, ultimately impacting the energy consumption as well.

In contrast, memory-intensive or I/O-intensive benchmarks respond well with VINCENT. This is consistent with our general understanding of DVFS: these benchmarks often have latency due to memory round-trips or I/O requests, and scaling down the CPU frequency may have limited impact on execution time while reducing the power consumption significantly. For example, there are benefits for reducing energy consumption for many configurations of `pmd` (AST-based program analysis), `avrora` (simulation), `fop` (file transformation), and `luindex` (data indexing). All are centric to data

processing, and most benchmarks have I/Os.

The same benchmarks also have less predictable performance due to the increased unpredictability with memory or I/O interactions. As a result, the benchmarks that respond to VINCENT well (those with more green cells) are also the ones more likely to have configurations whose behavior defies patterns, i.e., some random red cells among them.

Finally, relatively short methods (such as the top-consuming method of `pmd` and `bloat`) indeed respond to DVFS poorly: the overhead of DVFS significantly outweighs its benefit. As we can see, energy consumption may deteriorate significantly for them, sometimes near 10x.

The Impact on EDP Fig. 5.5 shows VINCENT's impact on energy consumption.

One interesting observation is that DVFS may play different roles for different benchmarks in balancing the trade-off between energy consumption and execution time: sometimes the reduction of EDP is due to reduced energy consumption, whereas at other times, EDP may reduce due to reduced execution time.

Take `sunflow` for instance. Recall earlier that its energy heatmap revealed that reducing the energy consumption of `sunflow` is challenging (all cells in the energy consumption heatmap are red), but observe that VINCENT may in fact improve the energy efficiency of `sunflow` in terms of EDP: by scaling the CPU frequency to the highest while executing its method `TriangleMesh.init`, the normalized EDP may reach 0.90, i.e., a 10% reduction than that of `ONDEMAND`. Here, VINCENT primarily plays the role of improving the performance: as `sunflow` is a CPU-intensive benchmark, DVFS plays the role of speeding up its execution; the shortened execution time contributes to the reduce EDP.

Overall, we find VINCENT an effective solution to reducing EDP as well as reducing energy consumption. Occasionally, it is even more effective for the former than the latter: when we correlate Fig. 5.4 and Fig. 5.5, the best configuration for a bench-

mark often exhibits a lower normalized value in Fig. 5.5 than in Fig. 5.4. As energy optimization is well known to be a trade-off between maximizing energy savings and minimizing performance loss, an EDP-friendly solution is of practical importance.

5.4.2 Sampling Settings

All experimental results we have shown so far are based on the setting where each optimization sampling interval is set at 8ms, and within each interval, 16 samples are taken. In other words, `EPOCH` $\times$ `PN` $= 8$ and `SAMPLENUM` $= 16$. We now evaluate VINCENT under different settings of sampling, with the energy consumption and EDP results shown in Fig. 5.6 and Fig. 5.7 respectively.

The primary observation is that the results are generally stable when the same benchmark is optimized under different sampling settings. The dominating factor of effectiveness — in terms of both energy consumption and EDP — remains to be the benchmark itself. For example, `pmd`, `avrora`, and `fop` can lead to the most energy savings and the least EDPs, and these facts remain true across all sampling settings. A general level of stability highlights that it is the principle of VINCENT — method-grained energy optimization — that leads to promising results, not the specifics of tuning.

On the other hand, these figures reveal that tuning sampling settings may indeed have small but noticeable impact on the effectiveness of VINCENT. As different choices of sampling settings represent different trade-offs between overhead and accuracy, the variation is not surprising; indeed, tuning has been a classic component in JVM optimization. For some settings – e.g., 8ms/8 and 8ms/16 for `avrora` energy consumption and 4ms/32 and 8ms/8 for `jython` EDP — the difference in effectiveness may be as much as 10%-12%. There appears to be no generalizable trend in terms of the selection of the sampling interval and the number of samples within each sample. For example, in terms of energy consumption (Fig. 5.6), `jython` optimization appears to more effective with 4ms-interval settings, while `antlr` optimization appears to be more effective

with 8ms-interval settings. In principle, the shorter the interval is and the more the samples are, the more likely VINCENT would be able to accurately perform DVFS at the boundary of methods. However, shorter intervals and more samples also imply more overhead, incurred both in terms of the sampling algorithm itself, and the overhead of DVFS.

5.4.3 Alternative Baselines

We have so far compared our results with the `ONDEMAND` governor, arguably the most widely used DVFS-enabled energy optimization based on dynamic monitoring. In this section, we now look at other important governors as baselines.

In Fig. 5.8, we show the relative effectiveness of VINCENT against alternative governors. For example, the height of `sunflow` EDP bar against the `ONDEMAND` governor is 0.86, meaning that among all CPU frequencies, all selected methods, and all sampling rate settings, the VINCENT configuration with the least EDP is 14% less than that of the `ONDEMAND` run for `sunflow`. For the same benchmark, its EDP bar against the `POWERSAVE` governor is 0.52, meaning that the VINCENT configuration with the least EDP is 48% less than that of the `POWERSAVE` run. In other words, `POWERSAVE` is a relatively less effective power governor for `sunflow` than `ONDEMAND` in terms of EDP, and neither is as effective as VINCENT.

Across the benchmarks, a trend is that the `POWERSAVE` baseline fares poorly relative to `ONDEMAND`, and much worse than VINCENT. Relatively, `POWERSAVE` is slightly worse than the `ONDEMAND` governor in terms of energy consumption, but it may significantly increase the execution time of benchmarks, ultimately leading to poor EDPs.

VINCENT is also more effective than the `PERFORMANCE` governor. Note that in the last row of Fig. 5.8, all normalized energy results are significantly less than 1. All but one (`sunflow`) benchmarks also have EDP results less than 1. The most revealing fact about the `PERFORMANCE` governor is that it may reduce the execution time of CPU-

intensive benchmarks. Recall that when VINCENT is compared against the ONDEMAND governor in terms of the execution time (the last figure in the first row), the VINCENT runs of sunflow and jython can lead to shorter execution time than the runs with the ONDEMAND governor. This however is not true when VINCENT is compared against the PERFORMANCE governor: the VINCENT runs of sunflow and jython are slightly slower than the runs with the PERFORMANCE governor (the last figure in the last row). The PERFORMANCE governor however is not as effective for memory-intensive or I/O-intensive benchmarks. The somewhat surprising fact is that the VINCENT runs for these benchmarks can in fact have a small but noticeable reduction in the execution time than their counterpart PERFORMANCE runs.

5.4.4 Multi-Method Optimization

As a part of the design space optimization, we further constructed experiments where multiple methods are subject to DVFS at the same time. Concretely, for benchmarks that have at least two methods that show favorable EDP configurations (normalized EDP < 1), we pick two methods whose least EDPs among all configurations are the smallest. We perform DVFS of both methods at the same time, adjusting the frequencies according to their respective "least EDP" configurations.

Unfortunately, the results do not show improvement. In fact, the 3 most promising benchmarks (i.e., with multiple EDP$<$1 configurations spanning different methods as shown in Fig. 5.5), pmd, avrora, and fop produced normalized EDP as 2.01, 1.77, and 1.60, respectively. The root cause is that when multiple methods are subjected to DVFS at the same time, the chance of concurrent DVFS requests increases significantly. As CPU hardware must serialize DVFS requests — in other words, frequency adjustment hardware is the "shared" resource — there is a noticeable slowdown for all benchmarks. An extensive increase in execution time is bad news for energy efficiency.

We think this negative result is interesting in its own right. It is a conscious reminder

that an overdesign may hamper effectiveness. VINCENT, as it turns out, is most effective when we keep it simple: method-grained energy optimization with a focus on the most impactful method in an application.

5.4.5 Summary

Fig 5.9 summarizes the average of VINCENT normalized energy/EDP/time against different baselines, across all benchmarks. On average, VINCENT can reduce energy consumption by 14.9%, EDP by 21.1%, and execution time by 12.5% against the `ONDEMAND` baseline. Its relative effectiveness against the `POWERSAVE` baseline is even more dramatic, with an EDP reduction of 63.0%. The drastic frequency downscaling in `POWERSAVE` may save *power*, but it is ineffective in energy optimization. On average, VINCENT's performance is on par with the `PERFORMANCE` baseline, with a negligible execution time reduction of 2.5%. Its effectiveness in energy and EDP reduction is similar to the result against the `ONDEMAND` baseline.

5.5 Related Work

A relatively small number of JVM-centric solutions exist for energy optimization. Chen et al. [23] relies on garbage collection tuning to save memory system energy consumption in JVMs. Cao et al. [19] improves the energy efficiency of JVM by assigning JVM services to small cores on asymmetric hardware. `DEP+BURST` [1] is a performance predictor and energy management system where JVM features such as synchronization, inter-thread dependencies, and store bursts, are taken into account for performance/energy prediction. Hussein et al. [45] investigates the energy impact of garbage collector design in the Android runtime. They proposed some extensions to improve the energy efficiency of asynchronous GC in Android. Overall, a common theme in existing work is to focus on JVM services (such as GC and thread management), but none considers energy optimization at the granularity of programming abstractions. Our work comple-

ments existing work with a fine-grained method-based approach for energy optimization. For unmanaged language runtimes, Hermes [78] is an energy-efficient solution built on top of Cilk. It performs DVFS based on the dependencies between thief threads and victim threads in a work stealing runtime.

Empirical studies often illuminate the energy consumption (and performance) of managed language runtimes. An early study by Vijaykrishnan et al. [91] focuses on the energy consumption impact on the memory hierarchy (cache and main memory) by JIT-enabled Java applications. Esmaeilzadeh et al. [33] studies energy efficiency with a focus on diverse configurations of workload and hardware. Sartor and Eeckhout [81] illuminates the performance of Java applications, with a focus on mapping Java application threads and JVM threads to multi-core hardware. Despite that their focus is on performance, DVFS is extensively used in their design space exploration, such as running GC threads at different CPU frequencies. Pinto et al. [73] studies the impact of energy consumption when alternative thread management designs in Java are used, such as different settings of the thread pool. Specific to ForkJoin [52], a study [70] also explored the impact of work stealing on the performance and energy trade-off in Java runtimes. The energy impact of different choices of Java collection classes — such as an `ArrayList` or a `HashMap` — were also a subject of studies [38, 75]. Kambadur [48] takes a cross-layer approach to surveying the energy management solutions, studying the interface and interaction of different hardware/OS/compiler configurations.

Energy profiling is more commonly conducted at the system level (e.g., [66, 35]), rather than at the boundary of programming abstractions such as methods. Chappie [6] supports method-grained energy profiling. It adopts an approach with fixed time intervals, a necessary design choice when there is no JVM modification. VINCENT is fundamentally a JVM-centric approach. It takes advantage of the JVM support such as instrumentation to enable delimited sampling. To VINCENT, energy profiling is an intermediate step for energy optimization, which Chappie does not support.

Energy optimization at the boundary of programming abstractions is more common in energy-aware programming languages [85, 8, 80, 43, 25, 9, 49, 42, 61, 96, 17]. For example, Eon [85] allows alternative data flow paths to be selected for different energy need. Green [8] and LAB [49] select alternative algorithm-specific parameters based on energy and QoS need. Energy Types [25] introduces DVFS at the boundary of methods based on phase information declared by programmers or inferred by the compiler. Ent [16] relies on hybrid type checking to select alternative programming abstractions (methods and objects) for message dispatch. VINCENT works with the existing programming model of Java; it is an effort on runtime design instead of programming model design.

5.6 Chapter Conclusion

VINCENT is a method-grained energy optimizer residing inside the JVM. It identifies the top energy-consuming methods in the Java runtime, and performs feedback-directed optimization guided by DVFS. Our experiments show VINCENT can reduce the energy consumption and improve the energy efficiency of Java applications. Broadly, VINCENT belongs to a small number of energy optimization approaches that take advantage of the information available to the application runtime. It requires no modification to the underlying OS/hardware, and no programmer effort.

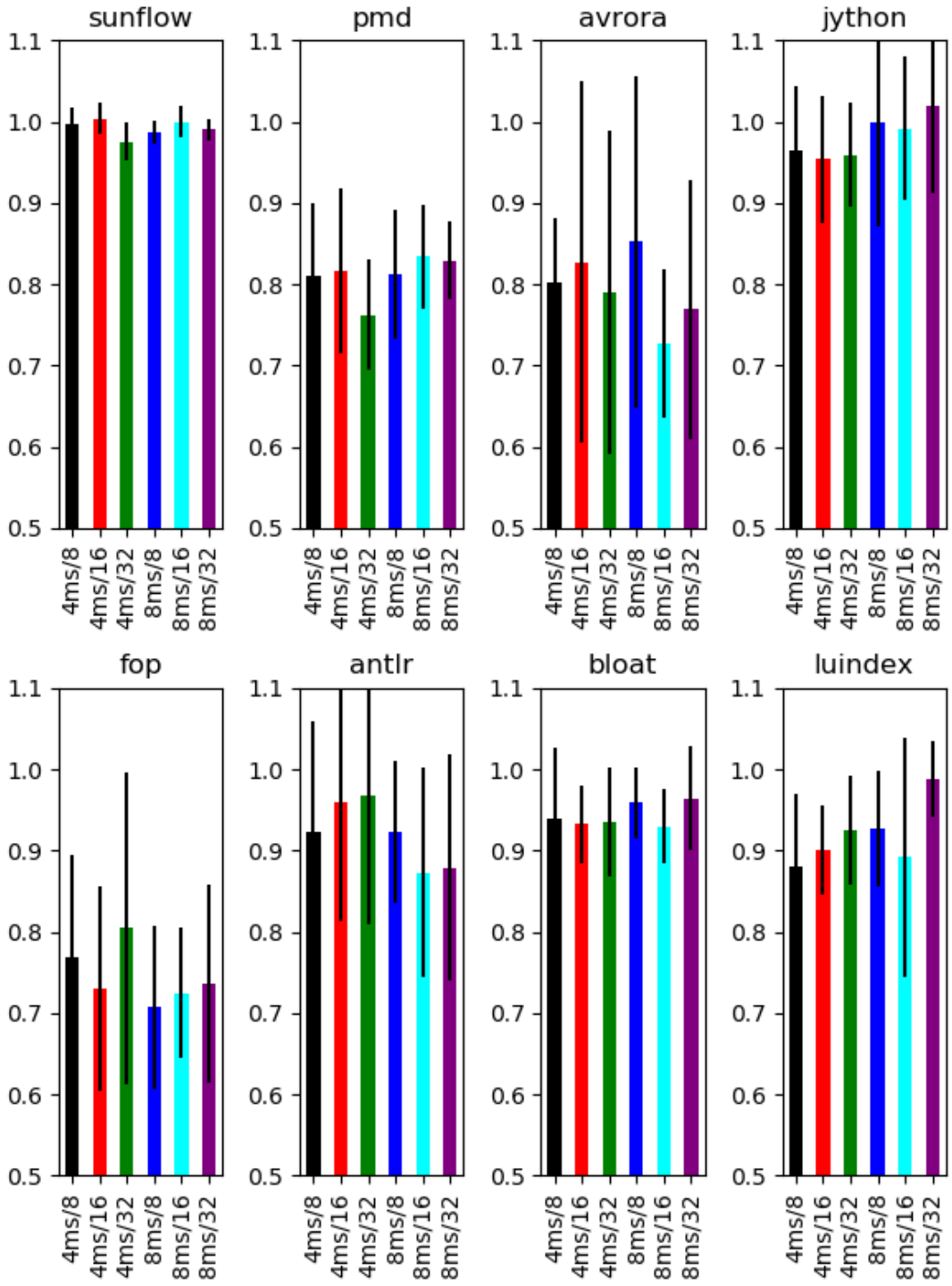


Figure 5.6: VINCENT Least Energy Consumption Under Different Sampling Settings Against the ONDEMAND Baseline (Under each bar, "Xms/Y" means EPOCH $\times$ SN $=$ X and SAMPLENUM $=$ Y. Each bar shows the least energy consumption among all combinations of methods and CPU frequencies for that benchmark, i.e., among all cells in a heatmap such as produced in Fig. 5.4. Each result is the average of multiple runs according to the discussion in § 5.3 under that setting. The thin bar on each bar shows standard deviation. For all bars, shorter is better.)

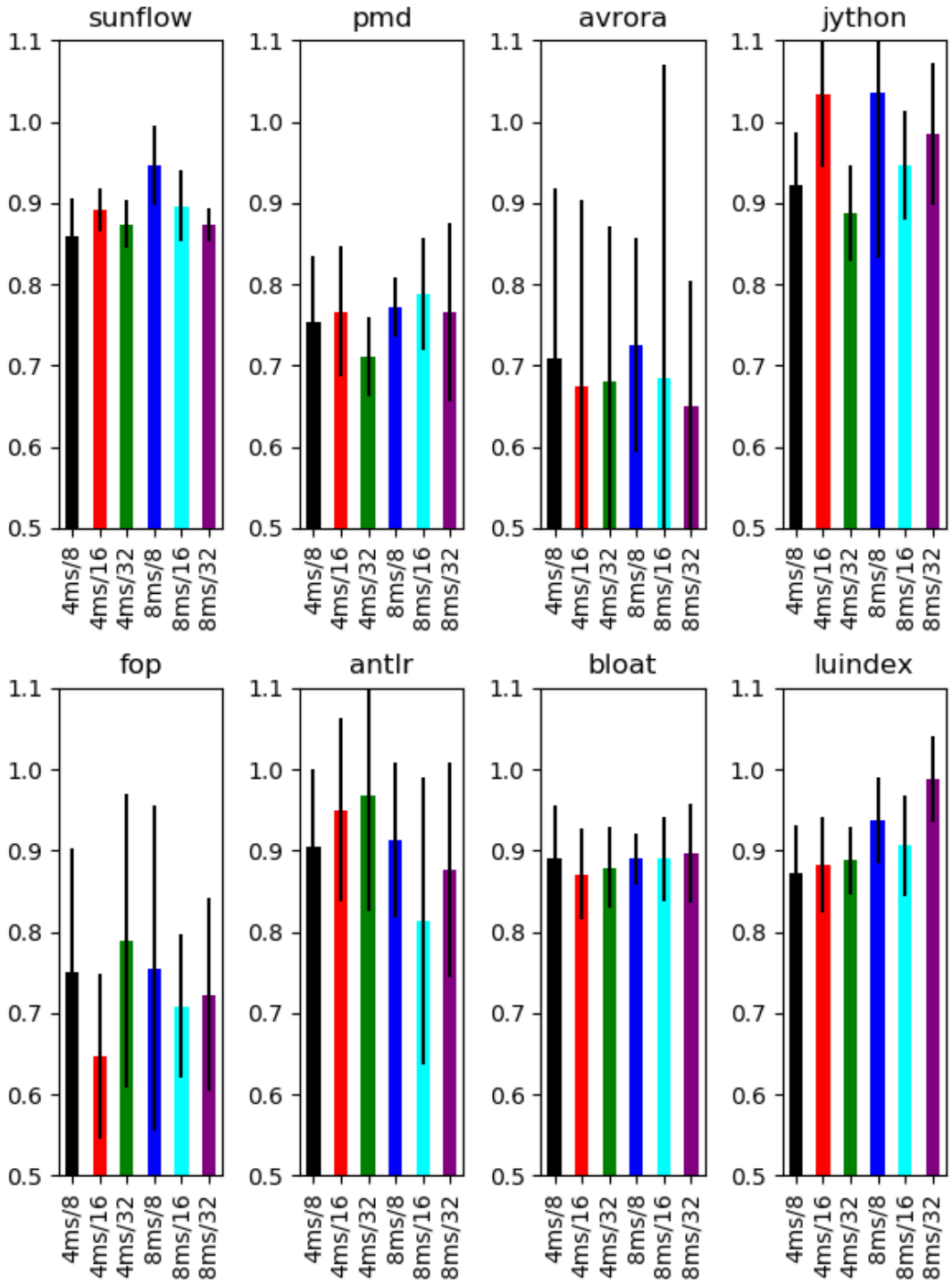


Figure 5.7: VINCENT Least EDP Under Different Sampling Settings Against the ONDEMAND Baseline (All legends are identical to those in Fig. 5.6. For all bars, shorter is better.)

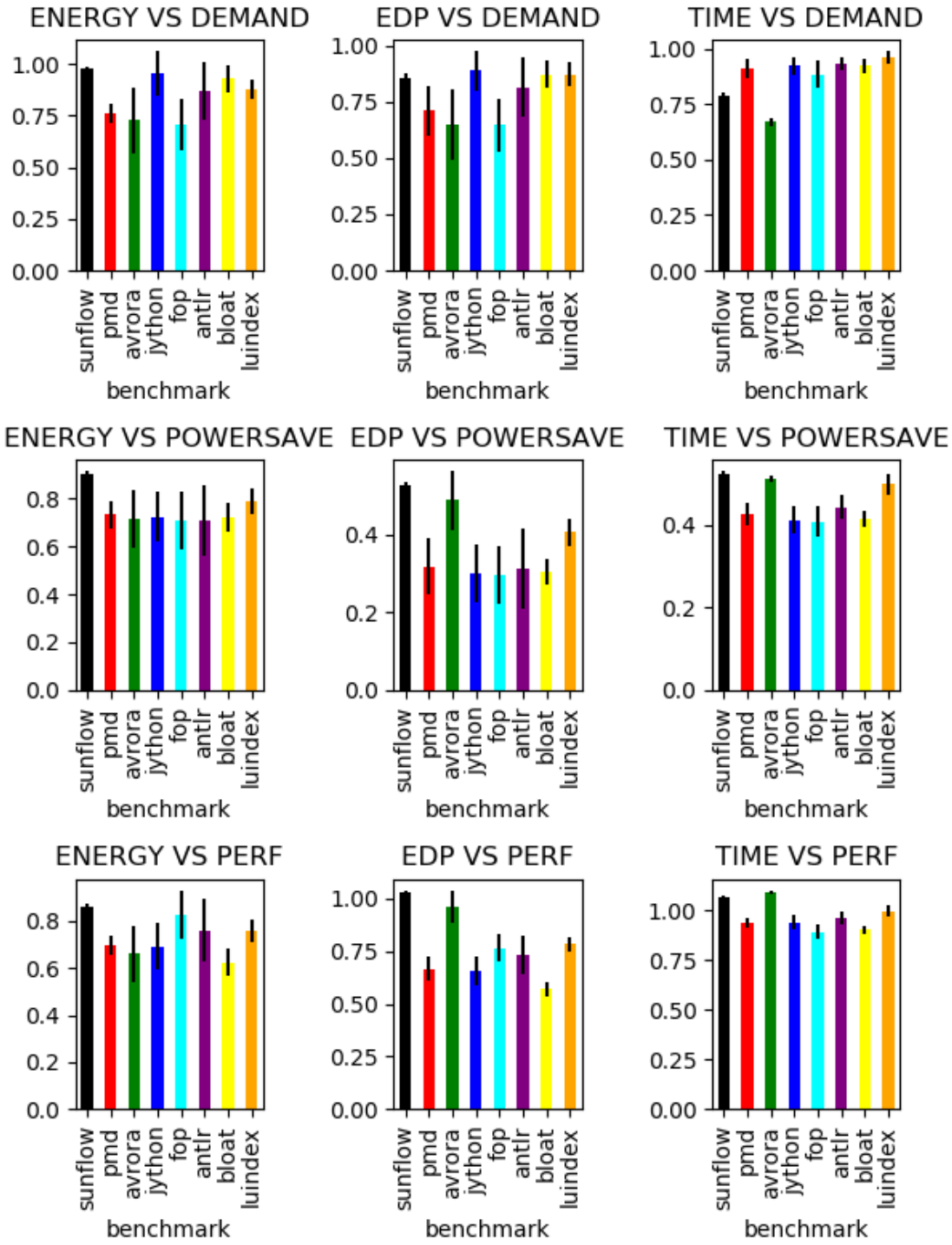


Figure 5.8: VINCENT Best Results against Different Governor Baselines (The first row shows results normalized against the `ONDEMAND` governor. The second row shows results normalized against the `POWERSAVE` governor. The third row shows results normalized against the `PERFORMANCE` governor. For all bars, shorter is better.)

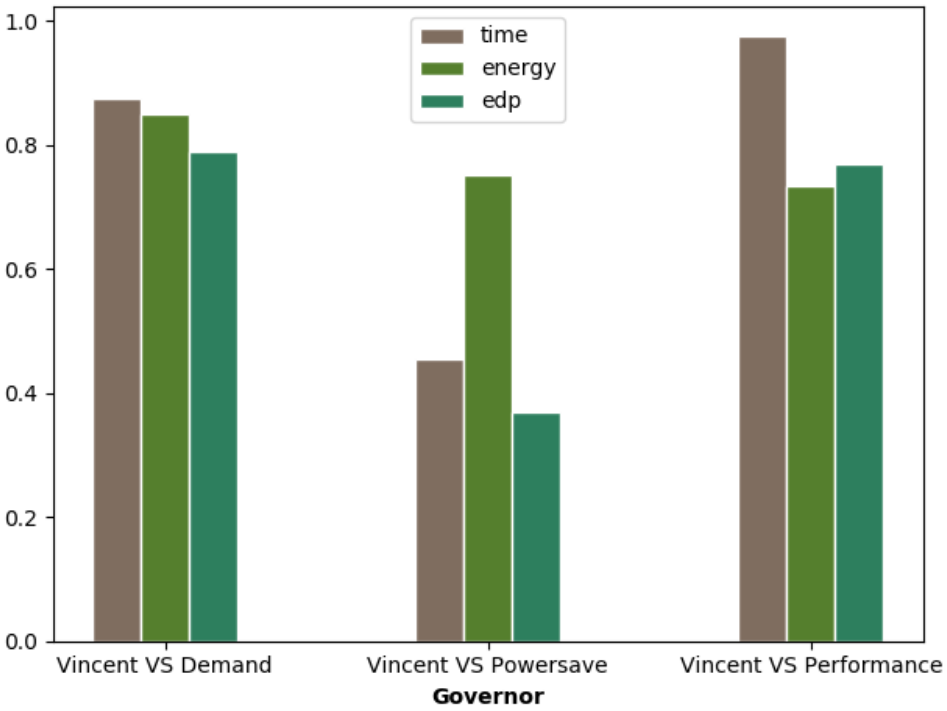


Figure 5.9: A Summary of Results with Different Governor Baselines (In each group, the energy/EDP/time data are normalized with their corresponding data under a built-in governor based on dynamic monitoring. For all bars, being shorter means VINCENT is more effective than the built-in governor.)

Chapter 6

Related Work

Energy-efficiency is an active research area many layers of compute stack. This chapter, we overview energy efficiency techniques in other layers.

6.1 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 [7], Energy Types [24], and Eco [95]. In these systems, recurring patterns of energy management tasks are incarnated as first-class citizens. Approximated programming [20] 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.

6.2 Energy Profiling

Energy profiling is a broad area of research. Prior work has attempted to model energy consumption at the individual instruction level [88], system call level [31], and

bytecode level [82]. Recent progress also includes fine-grained profiling for Android programs [40, 53], with detailed energy profiling of different hardware components such as camera, Wi-Fi and GPS. RAPL-based energy profiling has appeared in recent literature (*e.g.*, [47, 87]); its precision and reliability has been extensively studied [39].

Energy profiling is more commonly conducted at the system level (e.g., [66, 35]), rather than at the boundary of programming abstractions such as methods. Chappie [6] supports method-grained energy profiling. It adopts an approach with fixed time intervals, a necessary design choice when there is no JVM modification. VINCENT is fundamentally a JVM-centric approach. It takes advantage of the JVM support such as instrumentation to enable delimited sampling. To VINCENT, energy profiling is an intermediate step for energy optimization, which Chappie does not support.

6.3 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 [69], VM services [18, 47], cloud offloading [50], and refactoring [79]. 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.

Empirical studies often illuminate the energy consumption (and performance) of managed language runtimes. An early study by Vijaykrishnan et al. [91] focuses on the energy consumption impact on the memory hierarchy (cache and main memory) by JIT-enabled Java applications. Esmaeilzadeh et al. [33] studies energy efficiency with a focus on diverse configurations of workload and hardware. Sartor and Eeckhout [81] illuminates the performance of Java applications, with a focus on mapping Java application threads and JVM threads to multi-core hardware. Despite that their focus is

on performance, DVFS is extensively used in their design space exploration, such as running GC threads at different CPU frequencies. Pinto et al. [73] studies the impact of energy consumption when alternative thread management designs in Java are used, such as different settings of the thread pool. Specific to ForkJoin [52], a study [70] also explored the impact of work stealing on the performance and energy trade-off in Java runtimes. The energy impact of different choices of Java collection classes — such as an `ArrayList` or a `HashMap` — were also a subject of studies [38, 75]. Kambadur [48] takes a cross-layer approach to surveying the energy management solutions, studying the interface and interaction of different hardware/OS/compiler configurations. Li *et al.* [54] presented an evaluation of a set of programming practices suggested in the official Android developers web site. They observed that some practices such as the network packet size can provide impressive energy savings, while others, such as limiting memory usage, had minimal impact on energy usage. Vallina-Rodriguez *et al.* [90] surveys the energy-efficient software-centric solutions on mobile devices, ranging from operating system solutions to energy savings via process migration to the cloud and protocol optimizations.

Java collections are the focus of several studies [92, 76, 57], although only few of them are interested in analyzing their energy consumption [71, 62, 41]. In a previous effort, we have presented a preliminary study on the energy consumption of the Java thread-safe collections [71]. This current study greatly expands the previous study, attempting to answer four different research questions. The preliminary paper only addressed two of these research questions. Also, in this current submission, we employ two different energy profiling methods in two different machines, instead of just one, as reported in the initial study. Finally, we also applied our findings into two real-world, non-trivial benchmarks.

To the best of our knowledge, two other studies intersect with our goal of understanding the energy consumption of Java collections [62, 41]. SEEDS [62] is a general

decision-making framework for optimizing software energy consumption. The authors demonstrated how SEEDS can identify energy-inefficient uses of Java collections, and help automate the process of selecting more efficient ones. The focus of their work is the methodology itself, not the energy footprint of Java collections. The closest study was conducted by Hasan *et al.* [41]. In this study, the authors also investigated collections grouped with the same interfaces (`List`, `Set`, and `Map`). However, they do not focus on concurrent collections neither perform experiments in a multi-core environment. Therefore their study can be seen as complementary to ours.

6.4 JVM-Centric Solutions

A relatively small number of JVM-centric solutions exist for energy optimization. Chen et al. [23] relies on garbage collection tuning to save memory system energy consumption in JVMs. Cao et al. [19] improves the energy efficiency of JVM by assigning JVM services to small cores on asymmetric hardware. `DEP+BURST` [1] is a performance predictor and energy management system where JVM features such as synchronization, inter-thread dependencies, and store bursts, are taken into account for performance/energy prediction. Hussein et al. [45] investigates the energy impact of garbage collector design in the Android runtime. They proposed some extensions to improve the energy efficiency of asynchronous GC in Android. Overall, a common theme in existing work is to focus on JVM services (such as GC and thread management), but none considers energy optimization at the granularity of programming abstractions. Our work complements existing work with a fine-grained method-based approach for energy optimization. For unmanaged language runtimes, Hermes [78] is an energy-efficient solution built on top of Cilk. It performs DVFS based on the dependencies between thief threads and victim threads in a work stealing runtime. Zhang*et al.* [94] presented a mechanism for automatically refactoring an Android app into one implementing the on-demand computation offloading design pattern, which can transfer some computation-intensive

tasks from a smartphone to a server so that the task execution time and battery power consumption of the app can be reduced significantly. Cao *et al.* [18] described how different VM services (such as the Just-In-Time compiler, interpretation and/or the garbage collector) impact energy consumption.

Chapter 7

Conclusions and Future Directions

Energy optimization is an active direction in many computer science areas including hardware level, system level and programming model. However, energy optimization in application runtime is less considered in this area. With the knowledge of energy consumption behaviors, application runtime has the advantange of the ability of accessing the method-grained program behavior that other compute stack doesn't have.

7.1 Summary of Contributions

We implemented a software-based energy profiling library `jRAPL`. Different with the traditional meter-based energy profiling, jRAPL is able to allow developers customize the synchronization of the energy profiling scope. It greatly improves the flexibility of energy optimization in application level.

For the data-oriented characterization of application, We first explored How data management features could impact the energy consumption from 5 perspective: (i) Data access pattern (ii) Data representation (iii) Data organization (iv) Data precision (v) Data I/O strategies. We further explore how application-level energy management interact with hardware-level energy management from both micro-benchmarks and real-world application benchmark perspectives.

We also studied the energy efficiency of Java thread-safe collections and explored

the energy optimization possibilities in the real world application. We investigated 13 thread-safe and 3 non thread-safe Java collections and chart the energy optimization possibilities. The small changes have the potential of optimizing the energy consumption from both micro-benchmarks and real-world benchmarks.

We proposed Vincent, an JVM-based energy optimizing runtime system. Vincent is a novel runtime-based energy optimizer for method-grained energy profiling and optimization. Vincent has unique advantages compared with other layers of energy optimization. Relative to lower-level techniques such as hardware design and OS design, Vincent has ability to access the application runtime specific information which is unavailable to underlying systems. Relative to higher layer energy optimization techniques such as energy-aware programming models, Vincent have an advantage of backward compatibility with legacy code.

Bibliography

[1] S. Akram, J. B. Sartor, and L. Eeckhout. Dep+burst: Online dvfs performance prediction for energy-efficient managed language execution. *IEEE Transactions on Computers*, 66(4):601–615, 2017.

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. *SIGPLAN Not.*, 34(10):314–324, October 1999.

[3] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, 2000.

[4] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*, pages 51–62, 2005.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Lan-*

guages, and Applications, OOPSLA '00, page 47–65, New York, NY, USA, 2000. Association for Computing Machinery.

[6] Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, and Yu David Liu. Calm energy accounting for multithreaded java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 976–988, 2020.

[7] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[8] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pages 198–209.

[9] T. Bartenstein and Y. Liu. Green streams for data-intensive software. In *ICSE*, 2013.

[10] Thomas W. Bartenstein and Yu David Liu. Rate types for stream programs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 213–232, 2014.

[11] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(3):299–316, June 2000.

[12] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas

VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 169–190. ACM, 2006.

[13] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[14] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. A bloat-aware design for big data applications. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 119–130, 2013.

[15] T.D. Burd and R.W. Brodersen. Energy efficient cmos microprocessor design. In *HICSS'95*, pages 288–297 vol.1, 1995.

[16] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 217–232, 2017.

[17] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. Stochastic energy optimization for mobile GPS applications. In *Proceedings of the 2018 ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 703–713, 2018.

[18] 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.

[19] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, page 225–236, USA, 2012. IEEE Computer Society.

[20] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

[21] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power cmos digital design. *IEEE JOURNAL OF SOLID STATE CIRCUITS*, 27:473–484, 1995.

[22] AP. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, Apr 1992.

[23] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Trans. Embed. Comput. Syst.*, page 27–55, November 2002.

[24] M. Cohen, H. Zhu, S. Emgin, and Y. Liu. Energy types. In *OOPSLA*, 2012.

[25] Michael Cohen, Haitao Steve Zhu, Senem Ezgi Emgin, and Yu David Liu. Energy types. In *OOPSLA '12*.

[26] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *OOPSLA'12*, pages 831–850, 2012.

[27] H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *ISLPED*, 2010.

[28] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanaa, and Christian Le. RAPL: memory power estimation and capping. In *Proceedings of the 2010 International Symposium on Low Power Electronics and Design, 2010, Austin, Texas, USA, August 18-20, 2010*, pages 189–194, 2010.

[29] Mattias De Wael, Stefan Marr, and Tom Van Cutsem. Fork/join parallelism in the wild: Documenting patterns and anti-patterns in java programs using the fork/join framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 39–50, 2014.

[30] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, 2009.

[31] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *MobiSys*, 2011.

[32] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE'13, pages 422–431, May 2013.

[33] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the Sixteenth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, page 319–332, New York, NY, USA, 2011. Association for Computing Machinery.

[34] 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.

[35] X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou. E-android: A new energy profiling tool for smartphones. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 492–502, June 2017.

[36] Biran Goetz. Java theory and practice: Concurrent collections classes. `http://www.ibm.com/developerworks/java/library/j-jtp07233/index.html`. Accessed: 2014-09-29.

[37] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[38] Irene Lizeth Manotas Gutiérrez, Lori L. Pollock, and James Clause. SEEDS: a software engineer's energy-optimization decision support framework. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 503–514, 2014.

[39] 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.

[40] S. Hao, D. Li, W. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, 2013.

[41] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 2016 International Conference on Software Engineering*, ICSE '16.

[42] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214.

[43] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*.

[44] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. IEEE Symposium*, 1994.

[45] Ahmed Hussein, Mathias Payer, Antony L. Hosking, and Christopher A. Vick. Impact of GC design on power and performance for android. In Dalit Naor, Gernot Heiser, and Idit Keidar, editors, *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, pages 13:1–13:12. ACM, 2015.

[46] Canturk Isci and Margaret Martonosi. Identifying program power phase behavior using power vectors. In *In Workshop on Workload Characterization*, 2003.

[47] Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *OOPSLA*, pages 329–344, 2014.

[48] Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 329–344, New York, NY, USA, 2014. Association for Computing Machinery.

[49] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA '13*, pages 661–676.

[50] Y. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, 2013.

[51] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, pages 170–179, 2013.

[52] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, page 36–43, New York, NY, USA, 2000. Association for Computing Machinery.

[53] D. Li, S. Hao, W. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *ISSTA*, 2013.

[54] Ding Li and William G.J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *GREENS*, 2014.

[55] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, 2013.

[56] Jian Li and José F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(4):397–422, December 2005.

[57] Yu Lin and Danny Dig. Check-then-act misuse of java concurrent collections.

In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, ICST, 2013.

[58] Kenan Liu, Gustavo Pinto, and David Liu. Data-oriented characterization of application-level energy optimization. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, FASE'15, 2015.

[59] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *FASE 2015*, April 2015.

[60] Yu David Liu. Energy-efficient synchronization through program patterns. In *Proceedings of GREENS'12*, 2012.

[61] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 575–585, 2015.

[62] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *ICSE*, 2014.

[63] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[64] Emerson Murphy-Hill, Rahul Jiresal, and Gail C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 42:1–42:11, 2012.

[65] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, 1984.

[66] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, 2012.

[67] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[68] G. Pinto, F. Castor, and Y. Liu. Mining questions about software energy consumption. In *MSR*, 2014.

[69] G. Pinto, F. Castor, and Y. Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, 2014.

[70] Gustavo Pinto, Anthony Canino, Fernando Castor, Guoqing (Harry) Xu, and Yu David Liu. Understanding and overcoming parallelism bottlenecks in forkjoin applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 765–775, 2017.

[71] Gustavo Pinto and Fernando Castor. Characterizing the energy efficiency of java's thread-safe collections in a multicore environment. In *Proceedings of the SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS)*, SEPS '14, 2014.

[72] Gustavo Pinto, Fernando Castor, and Yu Liu. Mining questions about software

energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, 2014.

[73] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA '14*.

[74] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 345–360, 2014.

[75] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A comprehensive study on the energy efficiency of java thread-safe collections. In *International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[76] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. A large-scale study on the usage of java's concurrent programming constructs. *Journal of Systems and Software*, (0):–, 2015.

[77] H. Ribic and Y. Liu. Energy-efficient work-stealing language runtimes. In *ASPLOS*, 2014.

[78] Haris Ribic and Yu David Liu. Energy-efficient work-stealing language runtimes. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 513–528, 2014.

[79] C. Sahin, L. Pollock, and J. Clause. How do code refactorings affect energy usage? In *ESEM*, 2014.

[80] 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'11*.

[81] Jennfer B. Sartor and Lieven Eeckhout. Exploring multi-threaded java application performance on multicore hardware. OOPSLA '12, page 281–296, 2012.

[82] C. Seo, S. Malek, and N. Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *PerCom*, 2008.

[83] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[84] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[85] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07*, pages 161–174.

[86] Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 15–26, 2013.

[87] Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. In *ICPE*, 2013.

[88] 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.

[89] Anne E. Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444 – 449, 2013. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[90] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, 15(1):179–198, First 2013.

[91] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, JVM 2001*, Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, JVM 2001. USENIX Association, 2001. Funding Information: This research is supported in part by grants from NSF CCR-0073419, Pittsburgh Digital Greenhouse and Sun Microsystems.; 1st Java Virtual Machine Research and Technology Symposium, JVM 2001 ; Conference date: 23-04-2001 Through 24-04-2001.

[92] Guoqing Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26, 2013.

[93] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *In Proceedings of the USENIX Annual Technical Conference*, pages 43–56, 2003.

[94] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 233–248, 2012.

[95] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *ICSE'15*, May 2015.

[96] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *ICSE'15*, pages 767–777, 2015.