Vincent: Green Hot Methods in the JVM

Kenan Liu Khaled Mahmoud Joonhwan Yoo Yu David Liu

SUNY Binghamton, USA

{kliu20, kmahmou1, jyoo45, davidl} @binghamton.edu

Abstract

In this paper, we show the energy efficiency of Java applications can be improved by applying Dynamic Voltage and Frequency Scaling (DVFS) inside the Java Virtual Machine (JVM). We augment the JVM to record the energy consumption of *hot methods* as the underlying CPU is run at different clock frequencies; after all the frequency possibilities for a method have been explored, the execution of the method in an optimized run is set to the CPU frequency that leads to the most energy-efficient execution for that method. We introduce a new sampling methodology to overcome the dual challenges in our design: both the underlying measurement mechanism for energy profiling and the DVFS for energy optimization are overhead-prone. We extend JikesRVM with our approach and benchmark it over the DaCapo suite on a server-class Linux machine. Experiments show we are able to use 14.9% less energy than built-in power management in Linux, and improve energy efficiency by 21.1% w.r.t. the metric of Energy-Delay Product (EDP).

Keywords: energy efficiency, JVM, just-in-time compilation

1. Introduction

The carbon footprint of data centers has recently received significant scrutiny [42]. After mobile workloads, server-class workloads once again place energy-efficient computing in the spotlight. This design goal is addressed at many layers of the computing stack. Among them, a less explored approach is to study the energy impact of *managed runtimes*, a middle layer between high-level applications and low-level systems. Relative to lower-layer techniques on hardware design (e.g., [18]) and OS design (e.g., [61]), a runtime approach has the benefit of guiding energy optimization with runtime-specific information. Relative to higher-layer techniques e.g., energy-aware programming languages [56, 11, 50, 27, 20, 12, 35, 26, 41, 62, 16], a runtime approach can work with programs written in existing languages, arguably easier for adoption. In a nutshell, the runtime — strategically positioned 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., [2]), just-in-time (JIT) compilation strategies (e.g., [57]), and garbage collector (GC) designs (e.g., [30]) to inform energy optimization.

1.1. Our Approach: JVM-Level Method-Grained DVFS

We introduce a novel energy optimization at the level of the JVM. It relies on two basic 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.

Our key idea is *method-grained energy optimization*: it demarcates the boundary of DVFS [28, 14] adjustment with the boundary of methods. Our premise 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` [13] may carve out the boundary of a CPU-intensive computation, and the method `PSStream.write` in a file processing benchmark `fop` [13] may demarcate an I/O-intensive computation. It is well known that energy optimization based on DVFS can be effectively performed based on program *phased behaviors* [53, 54, 33], i.e., an application may go through phases of different levels of CPU intensity. For example, running an I/O-intensive program fragment at a lower CPU frequency can often save energy without hampering performance (see § 2.2 for details).

Operationally, our approach relies on profiling to assign desirable CPU frequencies to *hot methods*, the methods identified by the JIT for their frequent execution. This design decision is rooted in the fact that hot methods are frequently executed, and any improvement to their energy behavior may have an amplified effect. A fundamental challenge in design is that the gain resulted from DVFS is often eclipsed by the time/energy overhead introduced by DVFS itself. We address this challenge with two solutions. First, we come up with an automated energy profiling process to identify the most energy-consuming hot methods, so that the optimizer can focus more on how "energy hotspot" code regions respond to DVFS. Second, we introduce a form of *counter-based sampling* to DVFS instrumentation, so that the overhead introduced by DVFS is negligible given a reasonable range of sampling rates.

In contrast, the state-of-the-art approach for DVFS-based energy management relies on dynamically monitoring system states, e.g., the rate of cache or TLB misses. A classic example of this approach is the `ONDEMAND` governor, the default power governor in many Linux versions. This governor continuously predicts the level of CPU activities, and adjusts the CPU frequency to meet the demand. This approach is oblivious to the logical structure of the running application, and is 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 there is a phase change. In philosophy, our approach is more aligned with a small body of work that relies on compilers or runtimes to guide DVFS [51, 29, 60, 25, 59]. The relationship between these approaches and ours will be discussed in § 7.

1.2. Contributions

We introduce Vincent [1], the incarnation of JVM-level method-grained DVFS as an extension to JikesRVM [4, 3]. This paper makes the following contributions:

- the design of a profile-directed energy optimizer, an end-to-end solution that can automatically identify the most energy-consuming hot methods, determine the judicious frequency settings for executing hot methods, and apply DVFS for optimization;
- the specification of method-grained energy optimization at the level of JVM, including the low-overhead sampling algorithm for energy profiling and optimization;
- the implementation and evaluation of method-grained DVFS, which demonstrates its effectiveness relative to existing power governors.

Vincent is an open-source project. Its source code and all raw experimental data can be found online [2]. A preliminary version of this work appeared at the European Conference on Object-Oriented Programming (ECOOP 2022). In this journal version, we have expanded our experiments to include a design space exploration on parameter settings (§ 5.2), and a discussion on design choices (§ 6).

[1] "I have tried to express the terrible passions of humanity by means of red and green." — Letter from Vincent van Gogh to Theo van Gogh, Arles, 8 September 1888

[2] `https://bitbucket.org/vincent-paper/vinccent`

2. Background

Vincent lies at the intersection of two active yet largely independent research directions, energy-efficient computing and managed language runtimes, which we briefly review now.

2.1. Energy Optimization and Metrics

In physics, *energy* (in the unit of joules) is the multiplication of *power* (in the unit of watts) and *time* (in the unit of seconds). Not to lose generality, energy optimization techniques fall into 3 categories: (1) reducing power only; (2) reducing time only; (3) balancing the trade-off between power and time. The first route is an established area of research in hardware design, such as low-power VLSI design [18]. The second route is also mundane: any compiler or runtime optimization that can reduce the execution time of a program can be broadly viewed as an energy optimization. As these first two routes should be more properly named *power* optimization and *performance* optimization respectively, most existing *energy* optimization techniques *de facto* refer to the third route above, which Vincent also belongs to.

The obvious metric for evaluating energy efficiency is the energy consumption itself. In practice however, as most energy optimization techniques are a balancing act between power and time, the effect of these techniques on power and time should not be ignored. This is particularly true for time, as maintaining performance is an implicit and universal goal. As a result, a prevalent metric for evaluating energy efficiency is the *Energy-Delay Product* (EDP), the multiplication of energy and time. A lower EDP is aligned with our intuition that the energy consumption is reduced while the application remains performant.

2.2. DVFS

DVFS [28, 14] is a classic CPU hardware feature that enables the trade-off exploration between power and time. Except for specialized embedded CPUs, DVFS is supported in nearly all commodity CPUs available today. With DVFS, the operational frequency of a CPU can be dynamically adjusted, such as from 2Ghz to 1Ghz. Strictly speaking, DVFS is a *power* optimization design: the power consumption of a CPU has a near cubic relationship with its operational frequency; as a result, when the operational frequency is reduced (or *scaled down*), the power reduction can be dramatic. What makes DVFS a challenging *energy* optimization solution is that, when the CPU frequency is lowered, the execution time of a program typically becomes longer. Recall our earlier discussion that energy consumption is the multiplication of power and time, so the energy consumption effect of DVFS is complex. With EDP as a metric placing more emphasis on time (i.e., not energy consumption alone), the EDP effect of DVFS is even less obvious.

Empirically, downscaling is most effective when the program execution is less dependent on the CPU clock speed. The well known example is the I/O-intensive workload: the program may be waiting for an I/O to complete, and a wait will cause CPU pipeline stalls no matter what frequency is used.

Informally, DVFS is also known as *throttling*. This widely used informal term has an undertone to emphasize the effect of downscaling. Note that DVFS as an approach subsumes both *downscaling* and *upscaling*. The latter refers to the scenario when the operational frequency of the CPU is increased. Upscaling increases power, but may serve as a performance optimization (i.e., reducing execution time).

DVFS, when implemented, takes the form of a system call, where a special system file is written. Each DVFS call generally takes tens of microseconds to complete in modern CPUs [32].

2.3. OS Governors

DVFS provides the hardware capability on adjusting CPU frequencies, but in itself, no algorithm is defined on *when* scaling should happen, and *what* frequency the CPU should be scaled to. The latter is provided through OS-level algorithms called *governors*. The implementation of governors is platform-dependent: the algorithm used by the OS depends on what hardware features are available for power management (beyond DVFS itself).

For generality reasons, Linux provides a set of *generic governors* that do not require additional hardware support [6]. The `ONDEMAND` governor adjusts the underlying CPU frequency based on monitoring the status reported by the performance counters, and a higher CPU frequency is applied when a higher workload is encountered, and *vice versa*. Relative to the middle-of-the-road `ONDEMAND` governor, the `PERFORMANCE` governor on one side of the spectrum is a *time-biased* DVFS regulation algorithm; it lays emphasis on preserving execution time by setting the CPU frequency to be as high as possible. On the other side of the spectrum, the `POWERSAVE` governor is a *power-biased* DVFS regulator, laying more emphasis on reducing power consumption by setting the CPU frequency to be as low as possible. To facilitate customized energy optimization, Linux also comes with a `USERSPACE` governor, deferring all decisions of *when* and *what* decisions of DVFS to the layers of the software stack above the OS.

With additional hardware support for power management, the OS governor can delegate some regulation tasks to the hardware. One example is the Intel P-State [32, 31] support, where the CPU can be set to different power state levels. Instead of operating at a per-core level, the P-State power management operates at the level of a CPU package shared by all cores. When a particular P-State is set, the hardware is able to balance off the individual CPU frequencies of different cores to achieve a particular power budget. More recently, the question of *when* power state transitioning should happen can also be managed by the hardware itself, a feature called hardware-managed P-states (HWP).

On Intel architectures with P-State support, Linux power management can operate in either the *passive* mode or the *active* mode for power management [5]. For architectures without HWP, Linux defaults its behavior to the passive mode, where the Linux generic governors — `ONDEMAND`, `PERFORMANCE`, `POWERSAVE`, and `USERSPACE` — remain in use, except that setting the highest/lowest CPU frequencies in the generic governors are now supported as setting the highest/lowest power states. On Intel architectures with HWP support, Linux defaults its behavior to an *active* mode of P-state use, essentially deferring all its "govenoring" ability to the HWP hardware itself. In the active mode, there is no longer a `USERSPACE` governor; in other words, application-specific or user-specific DVFS is *not* allowed.

2.4. Energy Measurement and RAPL

A relatively independent design and evaluation question is how the energy consumption can be measured. For example, a traditional approach is to rely on the external power/current meters. With the progress of energy-aware computing, newer architectures come with hardware interfaces that can directly query the energy consumption of a computer system "live.", i.e., during the execution of its hosted application.

The most widely known hardware feature is Intel's Running Average Power Limit (RAPL) [21], available on all Sandybridge or newer Intel CPUs since 2011 and AMD's RAPL-compatible CPUs. RAPL can dynamically report the hardware energy consumption and incrementally store it in Machine-Specific Registers (MSRs). The reported energy consumption includes (i) CPU core energy consumption; (ii) CPU uncore energy consumption, i.e., those of on-chip caches, bus controllers, etc; (iii) DRAM energy consumption. RAPL has other features, such as capping the power consumption of a CPU, beyond the scope of this paper.

When implemented, each RAPL reading can be obtained through a number of reads to MSR registers, taking tens of microseconds in modern CPUs. To determine the energy consumption of an execution, a user may take one RAPL reading at the beginning of the execution and the other at the end, and compute the difference of the two.

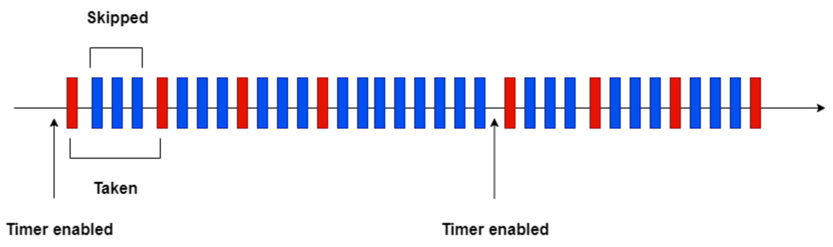

Figure 1: Counter-based Sampling

2.5. JVM Design and JIT

We briefly summarize key aspects of JVM design relevant to this paper. Vincent is built on top of JikesRVM, a representative research-oriented JVM. Research on JikesRVM contributed significantly in JVM design such as on JIT compilation and garbage collection.

JIT compilation allows selected bytecode to be dynamically compiled. One key component of JIT design is to determine which code fragments are most worthy for dynamic compilation. From JikesRVM to HotSpot, a common approach to this task is *hot method selection*: the JVM runtime observes the most frequently encountered methods and select them as the candidate for JIT. Conceptually, the JVM can achieve this task by keeping record of how frequent the beginning (commonly called the *prologue*) and the end (the *epilogue*) of each method are encountered. Realistic JVMs are more sophisticated implementations of this view, for reasons of both improving precision and reducing overhead.

2.6. Counter-Based Sampling

Precisely accounting for the number of times each method is called is expensive. Practical implementations are mostly *sampling*-based: the JVM only counts the prologue/epilogue encounters at time intervals.

In JikesRVM for example, a timer thread runs so that a sample is taken at fixed time intervals. JikesRVM further enhances this model by introducing *counter-based* sampling [7], allowing multiple samples 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. As shown in Fig. 1, 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 and the number of samples to skip between two

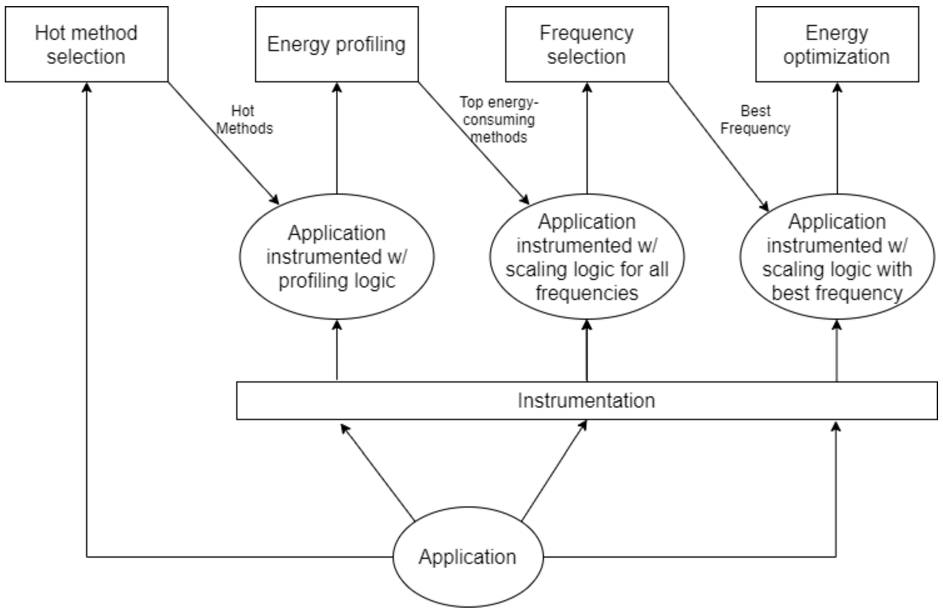
Hot method selection
Energy profiling
Frequency selection
Energy optimization
Hot Methods
Top energy-consuming methods
Best Frequency
Application instrumented w/ profiling logic
Application instrumented w/ scaling logic for all frequencies
Application instrumented w/ scaling logic with best frequency
Instrumentation
Application

Figure 2: Vincent Design and Workflow (The top 4 boxes refer to the 4 passes of Vincent workflow, subsequently from left to right. Each circle represents the application under optimization, in different forms of instrumentation. Each arrow refers to a data dependency/flow.)

samples. Vincent will adopt JikesRVM's counter-based sampling for *energy* profiling and optimization.

3. Vincent Design

In this section, we describe the design of Vincent, with a high-level description in § 3.1, followed by an algorithm specification in § 3.2.

3.1. System Overview

A Conceptual Overview. The system components and the workflow of Vincent are shown in Fig. 2. On the high level, Vincent is a profile-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 and execution time of the application when the execution of this method is scaled to each CPU frequency, which we call a *configuration*. For each top energy-consuming method, VINCENT ranks the efficiency of its different configurations according to energy metrics, and selects the most efficient one.

- **Energy Optimization**: VINCENT runs the application when the execution of each top energy-consuming method is scaled to the CPU frequency determined in the Frequency Selection phase.

The core design elements are the algorithms for energy profiling (the second pass) and method-based scaling (the third/fourth passes), which we will detail in § 3.2. Conceptually, one may view each pass as a separate run of the application, in the same spirit as a profile-guided optimizer. Therefore, the "energy profiling" pass and the "frequency selection" pass are two separate runs, which we informally call the *profiling run* and the *scaling run*, respectively.

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.*

A High-Level Implementation Overview. From the implementation perspective, VINCENT builds on top of JikesRVM, and we resort to existing support in JikesRVM for the first pass, Hot Method Selection. JikesRVM's built-in process—from how to sample methods to what heuristics are introduced to determine hotness—is not altered. Conceptually, hot method selection can be a separate run of the application itself, outputing a list of methods that JikesRVM deems "hot." In our implementation, the hot method selection and profiling is combined in one run: i.e., whenever a hot method is identified during the execution of an application, the energy profiling component of VINCENT will start profiling its energy consumption. In this regard, the VINCENT development interfaces with existing JikesRVM logic through a common data structure where hot methods are kept: whenever such a data structure is updated by JikesRVM, VINCENT under the profiling run will start profiling for the newly added entry. We also follow a similar implementation for the scaling run.

In addition, VINCENT does not alter the dynamic compilation process of JikesRVM, except that the additional logic for profiling (or scaling) is inserted through instrumentation at the beginning of the dynamic compilation process. Take the profiling run for instance. Whenever a hot method is

Algorithm 1 Thread Bookkeeping and Timer Thread Loop

```
1: typedef T {
2:     vtimer: int // timer
3:     skipCount: int // # calls to skip
4:     sampleCount: int // # samples to collect
5:     edata: EDATA // energy profiling data
6:     gov: GOVERNOR // saved governor
7:     freq: FREQ // saved CPU frequency
8: }
9: const EPOCH // time unit
10: const SKIPNUM // skipped samples between
11: const SAMPLENUM // samples per interval
```

```
1: ts: T[THREADNUM] // running threads
2: procedure TIMER
3:     while TRUE do
4:         SLEEP(EPOCH)
5:         for each $t \in ts$ do
6:             t.vtimer++
7:         end for
8:     end while
9: end procedure
```

identified, we dynamically instrument that method with the VINCENT profiling logic in the profiling run, which will be subsequently compiled by JIT dynamic compilation.

Combining the hot method selection and profiling in one run has one additional benefit: our profiling is fundamentally adaptive to the dynamic hot method selection process. For long-running applications with phased behaviors, what methods are deemed hot may dynamically change. Recall that VINCENT shares the same common data structure where the list of hot method are maintained by JikesRVM. In the scenario of dynamic hot method change, JikesRVM will dynamically place the newly identified hot method(s) in the common data structure. Upon the addition of each hot method entry in the data structure, dynamic compilation for that method will be triggered, in this case, with the profiling logic of VINCENT. As a result, the energy consumption of that method will be hereafter profiled by VINCENT.

3.2. VINCENT *Specification*

We now specify the algorithm implemented by VINCENT. We first describe the top-level thread bookkeeping (§ 3.2.1), and then the profiling algorithm (§ 3.2.2) and the scaling algorithm (§ 3.2.3).

3.2.1. 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

Algorithm 2 Profiling Algorithm

```
typedef LOG {
  mn: MNAME // method name
  edata: EDATA // data
}
typedef CVAL enum { TAKE, SKIP, LAST }
typedef EDATA float
const PN // profiling timer factor
l: LOG[LOGNUM]

procedure prologueProfile()
    $t \leftarrow$ currentThread()
    if Counter($t$, PN) == TAKE then
        $t.edata \leftarrow$ readEnergy()
    end if
    if Counter($t$, PN) == LAST then
        $t.edata \leftarrow \perp$
    end if
end procedure

procedure epilogueProfile()
    $t \leftarrow$ currentThread()
    if Counter($t$, PN) == TAKE or LAST then
        $e \leftarrow$ readEnergy()
        if $t.edata \neq \perp$ then
            $l \xleftarrow{+}$ LOG(THISM, diff($e$, $t.edata$))
        else
            $t.edata \leftarrow e$
        end if
    end if
    if Counter($t$, PN) == LAST then
        $t.edata \leftarrow \perp$
    end if
end procedure

function Counter($t$: T, $factor$: int): CVAL
    if $t.vtimer >= factor$ then
        $t.skipCount \leftarrow t.skipCount - 1$
        if $t.skipCount == 0$ then
            $t.skipCount \leftarrow$ SKIPNUM
            $t.sampleCount \leftarrow t.sampleCount - 1$
            if $t.sampleCount == 0$ then
                $t.vtimer \leftarrow 0$
                $t.sampleCount \leftarrow$ SAMPLENUM
                return LAST
            end if
            return TAKE
        end if
    end if
    return SKIP
end function
```

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 LIME 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.

In the rest of this section, we specify our algorithm design for energy profiling and DVFS-based energy optimization. Both passes are unified by one fundamental hurdle: if naive instrumentation is used, the overhead for obtaining raw energy samples (in energy profiling) and the overhead for performing DVFS (in energy optimization) are too high. We now detail our solution in § 3.2.2, i.e., how we overcome the overhead challenge of obtaining raw energy samples in energy profiling through a sampling-based approach. Note that in § 3.2.3, the same sampling-based solution is also used for DVFS-based energy optimization to overcome the challenge posed by the overhead for performing DVFS.

3.2.2. 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 § 4) 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. 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 instrument the begin (i.e., prologue) and the end (i.e., epilogue) 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 RAPL 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. Observe 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 and Counter-Based Sampling with VINCENT. To address these challenges, the solution adopted by VINCENT consists of two ideas: *delimited sampling* and *counter-based sampling.* Overall, the former is an overhead-reducing approximation that combines the strawman solutions above, and the latter is a precision-increasing optimization over the general sampling-based approach.

Delimited Sampling 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 § 4. Sampling happens within the function of COUNTER, which we will describe shortly.

The key observation 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 or epilogue (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. This can be concretely observed in the specification of EPILOGUEPROFILE in Algorithm 2. At Line 23, the energy difference between a prior energy sample and the current energy sample is computed. Now that the method has reached its epilogue, the “current energy sample” intuitively keeps the accumulated energy value until the method reaches its end. The intriguing question however is when the “prior energy sample” is collected. Delimited sampling introduces an approximation: it is collected during the last time in the sampling trace when a method is called (i.e., a prologue is executed) or a method is returned (i.e., an epilogue is executed). They can be seen at Line 12 and Line 21 respectively in Algorithm 2. In other words, the refutation-based algorithm says that any prior encountered prologue or epilogue “*delimits*” where the method could start: any energy consumption before the last method is called or returned *must not* belong to the current method we encounter in the epilogue.

On a more technical level, treating the prior encounter of an epilogue as a "limit" of the method start (as well as the prior encounter of a prologue) is also friendly for accounting for the energy consumption of a recursive/nested method. For some applications, the hot method happens to be a recursive call. When a sample is ready to be taken, it is possible that the activation record of the recursive call is popping. Without Line 21, the sampling algorithm would only take the next energy sample when a prologue is executed (i.e., a push), and hence would miss a round of sampling in this pop-only phase of recursive execution. With Line 21, the energy consumption between 2 pops can be recorded and attributed to the recursive method.

Finally, note that the energy accounting specified here is conceptually "flat": in the presence of a call chain where both the caller method and the callee method are hot, the callee's energy consumption is *not* accounted as a part of the caller's energy consumption. This is implied in the delimited approach itself: when the epilogue of the caller method is encountered, the epilogue of the callee method is already encountered. As a result, only the energy consumption after the callee method is completed is attributed to the caller method. Indeed, due to sampling, our implementation is an approximation of this conceptually flat view.

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 (see § 2), allowing multiple (but still bounded) pairs of energy readings to be collected within a time interval. In general, counter-based sampling is a precision-improving strategy known to strike a balance for accounting both long methods and short methods. 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 energy consumption.

In Algorithm 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

Algorithm 3 Scaling Algorithm

```
1: enum GOVERNOR {USERSPACE, ONDEMAND, ...}
2: const SN // scaling timer factor

3: procedure PROLOGUESCALE($f$ : FREQ)
4:     $t \leftarrow$ CURRENTTHREAD()
5:     if COUNTER($t$, SN) == TAKE then
6:         $t.gov \leftarrow$ GETGOVERNOR()
7:         if $t.gov$ == USERSPACE then
8:             $t.freq \leftarrow$ 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:    $t \leftarrow$ CURRENTTHREAD()
20:    if COUNTER($t$, SN) == TAKE then
21:        if $t.gov \neq \perp$ 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
```

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 profiling, this means to reset the *edata* field.

Finally, observe that the COUNTER function only accesses data that records the state of the *current* thread. This can be observed that every access in this function is prefixed with variable *t*. In other words, it is not possible for two application threads to access the same fields in a race condition.

3.2.3. Scaling Instrumentation

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

The scaling instrumentation is used in two passes of VINCENT. In the Frequency Selection pass (§ 3.1), the program is run for `FN` $\times$ `MN` number of times, where `FN` is the number of CPU frequencies supported by the hardware and `MN` is the number of top energy-consuming hot methods. Each run scales a particular top energy-consuming method to a particular CPU frequency when the former is encountered, according to the scaling instrumentation algorithm. The `FN`$\times$`MN` runs will allow VINCENT to iterate over all configurations. For each top energy-consuming method, VINCENT identifies the best CPU frequency among `FN` runs, i.e., the one that leads to the most energy efficiency. In the Energy Optimization pass (§ 3.1), only the optimal CPU frequency is scaled to when the aforementioned top energy-consuming method is encountered, again, according to the scaling instrumentation algorithm specified here. This process will be concretized in Section 4, where the `FN` $\times$ `MN` runs will be represented as a heatmap.

4. Implementation and Experimental Settings

4.1. 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. Hyperthreading is enabled. In total, we have 20 physical cores and 40 virtual cores. The machine runs Debian 9.11 (stretch), 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 paper, we use `F1` to refer to 2.2Ghz, `F2` for 2.1Ghz, `F3` for 2.0Ghz, . . . , `F11` for 1.2Ghz. In other words, the constant `FN` = 11 in our experiments. The CPU frequencies are switched through the `scaling_setspeed` file, under the directory of `/sys/devices/system/cpu/cpu*/cpufreq` for CPU cores.

Intel E5-2630 v4 is an instance of the Intel Broadwell architecture. It supports P-states but does not have HWP support. The P-states operate in the `passive` mode (see § 2), and the Linux governors of `ONDEMAND`, `PERFORMANCE`, `POWERSAVE`, `USERSPACE` remain available. The governors are switched through

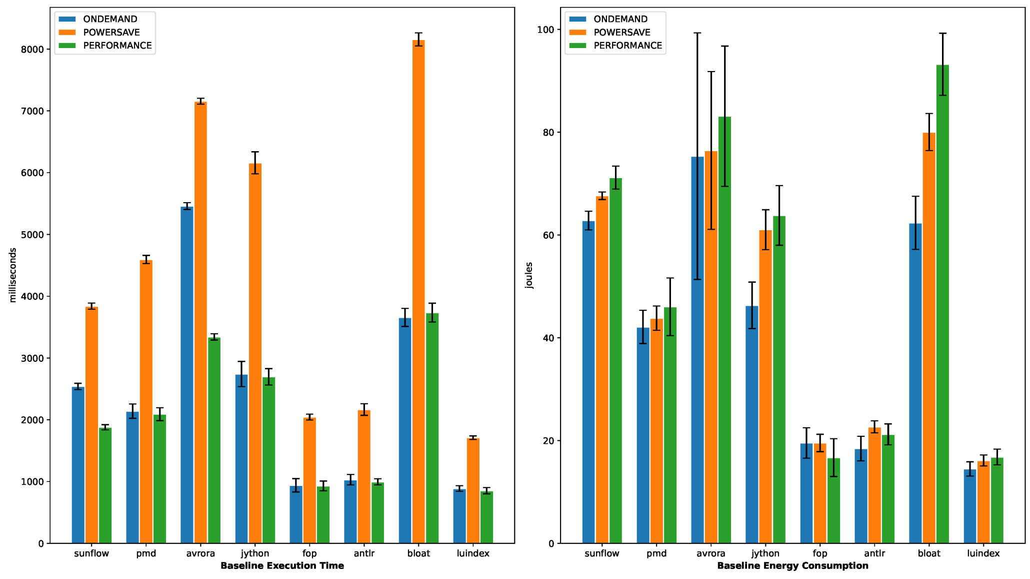

Figure 3: Benchmark Statistics under Different Governors as Evaluation Baselines

setting the `scaling_governor` file under the same directory as above. Recall that the `active` mode does not support `USERSPACE` govenor, so it cannot be used for VINCENT. To avoid feature intervention, Turbo boost is turned off. None of the experiments described in this paper (including both for VINCENT and for baselines) alters other system settings related to power management.

We rely a Java-based tool jRAPL [38] to obtain raw RAPL 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) [8] of JikesRVM.

4.2. Hot Method Selection

We rely on the JIT component of JikesRVM for hot method selection. We do not alter JikesRVM's hot method selection logic. The interaction between the JikesRVM logic and VINCENT is primarily through the data structure where hot methods are placed: whereas JikesRVM places hot methods into the structure, the profiling/scaling logic of VINCENT reads from it. 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. For each experiment, we consider the top-5 energy-consuming methods for optimization. In other words, `MN` = 5.

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 desired frequency. Fortunately, the top energy-consuming methods identified by VINCENT's energy profiler (as seen in § 5) 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.

4.3. 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 instrumentation: as the method signature is carried with the IR, VINCENT stores the method information when instrumentation is added. Other than this instrumentation, we preserve the original JikesRVM logic for dynamic compilation.

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 [7].

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

4.4. Benchmarking and Experimental Setup

We evaluate VINCENT with benchmarks in the Dacapo suite [13], arguably the most widely used benchmark suite for multithreaded 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.

4.5. Baselines

To evaluate the effectiveness of VINCENT, we choose 3 baselines. They are the three application execution scenarios where DVFS is guided by the `ONDEMAND`, `POWERSAVE`, and `PERFORMANCE` OS governors respectively (see § 2). They are representative scenarios of running Java applications on commodity software/hardware stack today. As variants of DVFS approaches guided by dynamic monitoring, they set a contrast with the core idea of VINCENT's approach, *method*-based DVFS.

The baseline execution time and energy consumption of each benchmark while running with the 3 Linux governors can be found in Fig. 3. In addition to serving as experimental baselines, this figure may also help gain intuition on the characteristics of DVFS guided by the 3 governors. For example, the `PERFORMANCE` governor often leads to the shortest execution time, as shown in the left sub-figure; it however generally increases the energy consumption, as shown in As shown in the right sub-figure. Overall, the `ONDEMAND` governor strikes a good balance between maximizing energy savings while delivering competitive performance. As a result, we will conduct a more detailed comparative analysis between VINCENT and the `ONDEMAND` baseline in the following section.

Unless otherwise noted, all experiment results throughout the paper (including both baseline runs and VINCENT runs) are collected by running each benchmark 20 times in a hot run, and reporting the average of the last 15 runs.

5. Vincent Evaluation

In this section, we evaluate the effectiveness of VINCENT. We aim at answering the following questions: (**Q1**) Do method-frequency configurations exist that can lead to energy savings and favorable EDPs, compared with existing Linux power governors? (**Q2**) How does the choice of sampling settings impact 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.

5.1. Method-Grained Energy Optimization

5.1.1. Energy Profiling

The VINCENT lifecycle starts with energy profiling. Fig. 4 shows the top-5 energy-consuming methods for selected benchmarks. Thanks to sampling, sunflow

Rank	Method Name	Percentage(%)
1	`org.sunflow.core.light.TriangleMeshLight.getRadiance`	9.36
2	`org.sunflow.core.primitive.TriangleMesh.init`	4.60
3	`org.sunflow.math.Matrix4.transformP`	2.19
4	`org.sunflow.core.shader.MirrorShader.getRadiance`	0.45
5	`org.sunflow.core.accel.KDTree.BuildTask.<init>`	0.005

pmd

Rank	Method Name	Percentage(%)
1	`org.jaxen.expr.DefaultAllNodeStep.matches`	15.52
2	`org.jaxen.expr.iter.IterableChildAxis.supportsNamedAccess`	8.21
3	`org.jaxen.QualifiedName.hashCode`	7.01
4	`net.sourceforge.pmd.jaxen.DocumentNavigator.getAttributeName`	4.78
5	`org.jaxen.util.SingleObjectIterator.hasNext`	4.18

antlr

Rank	Method Name	Percentage(%)
1	`antlr.CodeGenerator._println`	5.56
2	`antlr.SimpleTokenManager.getTokenSymbol`	5.23
3	`antlr.LLkAnalyzer.look`	3.92
4	`antlr.CSharpCharFormatter.escapeChar`	2.61
5	`antlr.Grammar.getSymbol`	2.61

Figure 4: Top Energy-Consuming Methods According to VINCENT Energy Profiling (The first column is the rank; the second column is the name of the method; the third column is its normalized energy consumption relative to the overall energy consumption of the benchmark.)

the reported percentage of energy consumption for each listed method is likely to be lower than its actual normalized energy consumption, but what matters here is the relative standing of the methods: we are able to identify the most-energy consuming methods so that the methods that DVFS should be applied upon are identified.

Very short methods rarely appear in the top energy-consuming methods. One example is `pmd`'s top-consuming method, `DefaultAllNodeStep.matches`, which only contains a simple boolean return as its method body. As we shall see soon, these methods are indeed unfriendly for DVFS (see § 4). That being

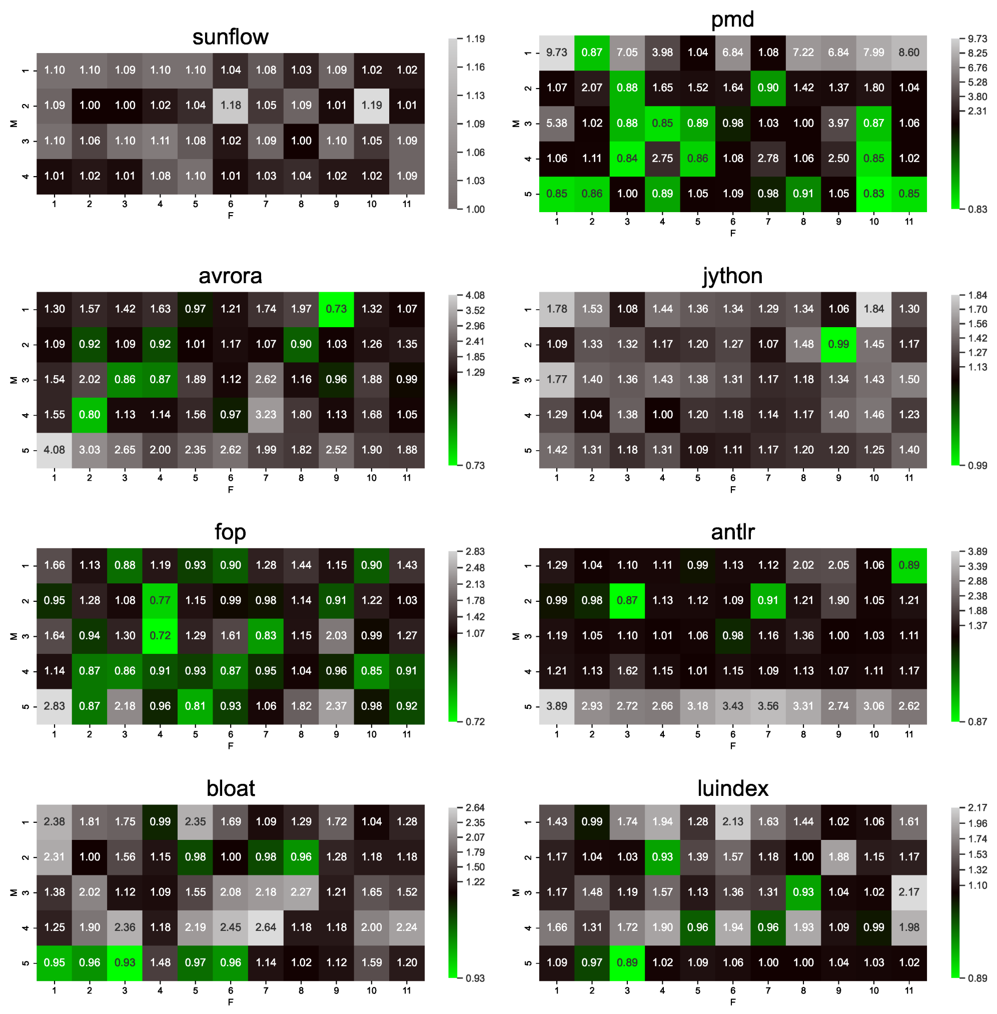

Figure 5: VINCENT Energy Consumption Normalized Against the `ONDEMAND` Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has energy consumption v, normalized against that of the `ONDEMAND` run. If $v < 1$, the VINCENT incurs less energy than the `ONDEMAND` run.)

said, the vast majority of methods identified by VINCENT's profiling phase are methods of reasonable length (in terms of execution time) where the DVFS time overhead is relatively small to the execution time of the method itself.

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.

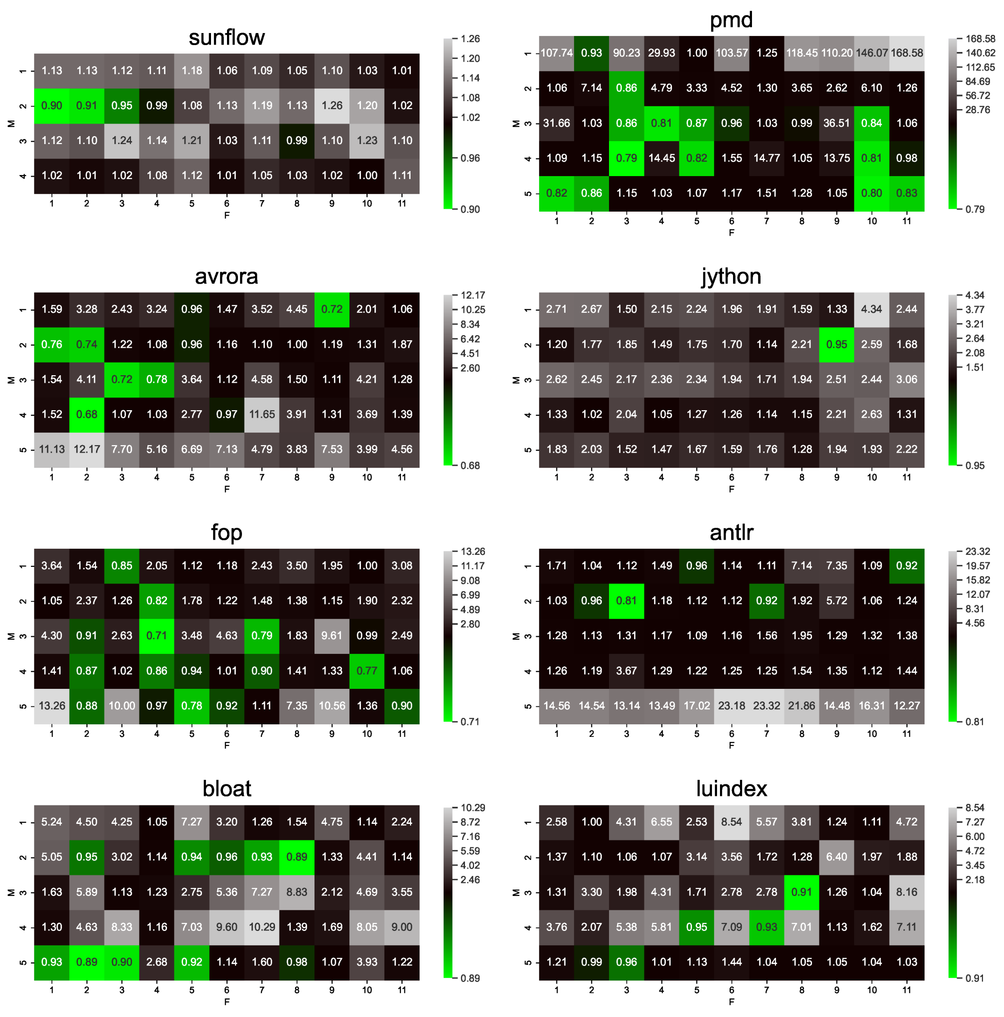

Figure 6: VINCENT EDP Normalized Against the `ONDEMAND` Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has EDP v, normalized against that of the `ONDEMAND` run. If $v < 1$, the VINCENT is more energy-efficient than the `ONDEMAND` run w.r.t. EDP.)

5.1.2. 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 energy consumption results of VINCENT in Fig. 5 when a single hot method is scaled to a particular CPU frequency. 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

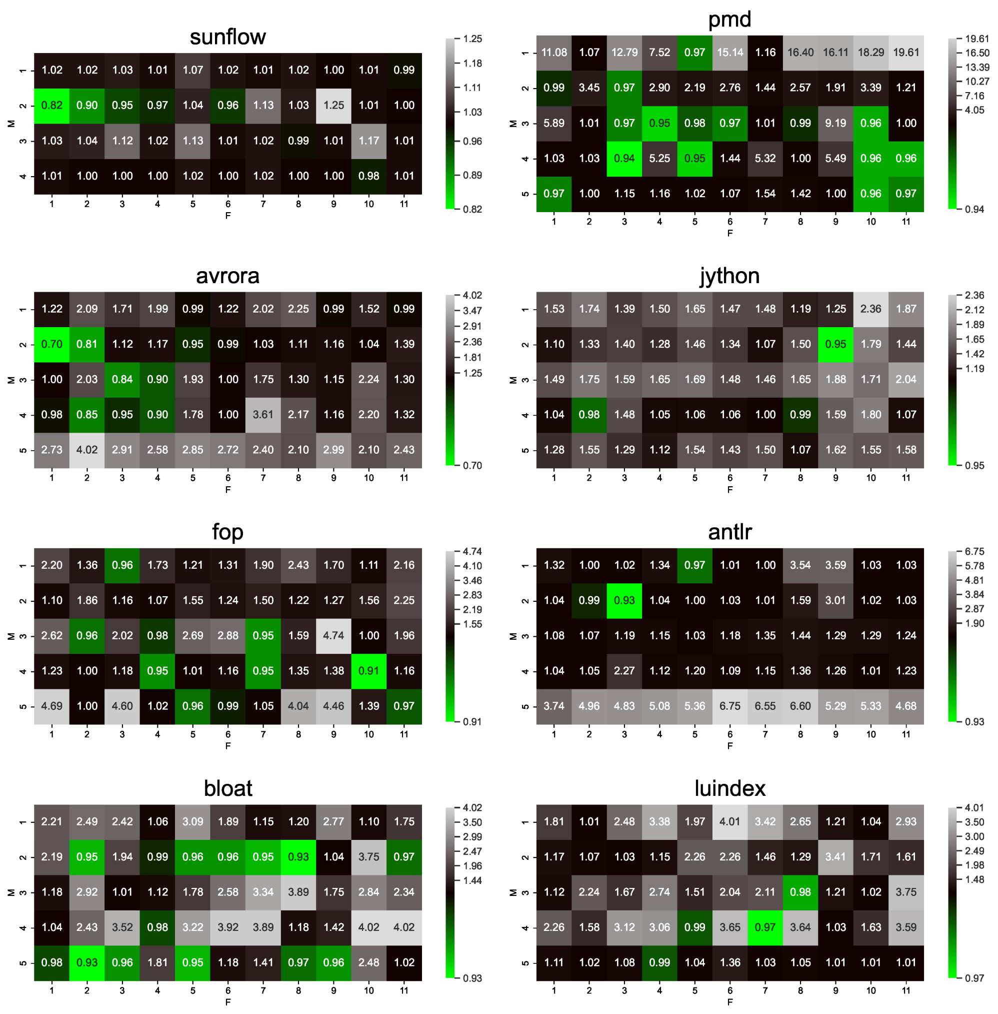

Figure 7: VINCENT Execution Time Normalized Against the ONDEMAND Baseline (For a cell of method m and frequency f with a value of v, it says that the VINCENT run with method m running at frequency f has execution time v, normalized against that of the ONDEMAND run. If $v < 1$, the VINCENT runs faster than the ONDEMAND run.)

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. 4.

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.

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.

5.1.3. The Impact on EDP

Fig. 6 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 reduced EDP.

Overall, we find VINCENT an effective solution to reducing EDP as well as energy consumption. Occasionally, it is even more effective for the former than the latter: when we correlate Fig. 5 and Fig. 6, the best configuration for a benchmark often exhibits a lower normalized value in Fig. 6 than in Fig. 5. As energy optimization is a known trade-off between maximizing energy savings and minimizing performance loss, an EDP-friendly solution is of practical importance.

5.1.4. The Impact on Execution Time

In Fig. 7, we show the impact of VINCENT on execution time. Observe that every benchmark consists of at least one configuration that may speed up the benchmark relative to its `ONDEMAND` run. At the first glance, the fact that VINCENT may serve as a *performance* optimizer may come as a surprise, but this is indeed natural for two reasons.

First, even though DVFS is better known for its effect on energy savings with *downscaling*, the opposite is also true: it can speed up the program execution with *upscaling*. What this figure shows is that VINCENT may select a performance-sensitive method and execute it on a higher CPU frequency than an `ONDEMAND` governor baseline would, potentially speeding up the program.

Second, note that `ONDEMAND` governor is a "middle-of-the-road" governor (see § 2) in terms of how aggressive/conservative it scales up CPU frequencies in the presence of workload increase. As we shall see in § 5.3, the `PERFORMANCE` governor is a more challenging baseline to overcome in terms of viewing VINCENT as a performance optimization.

5.2. 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 results shown in Fig. 8, and EDP results shown in Fig. 9.

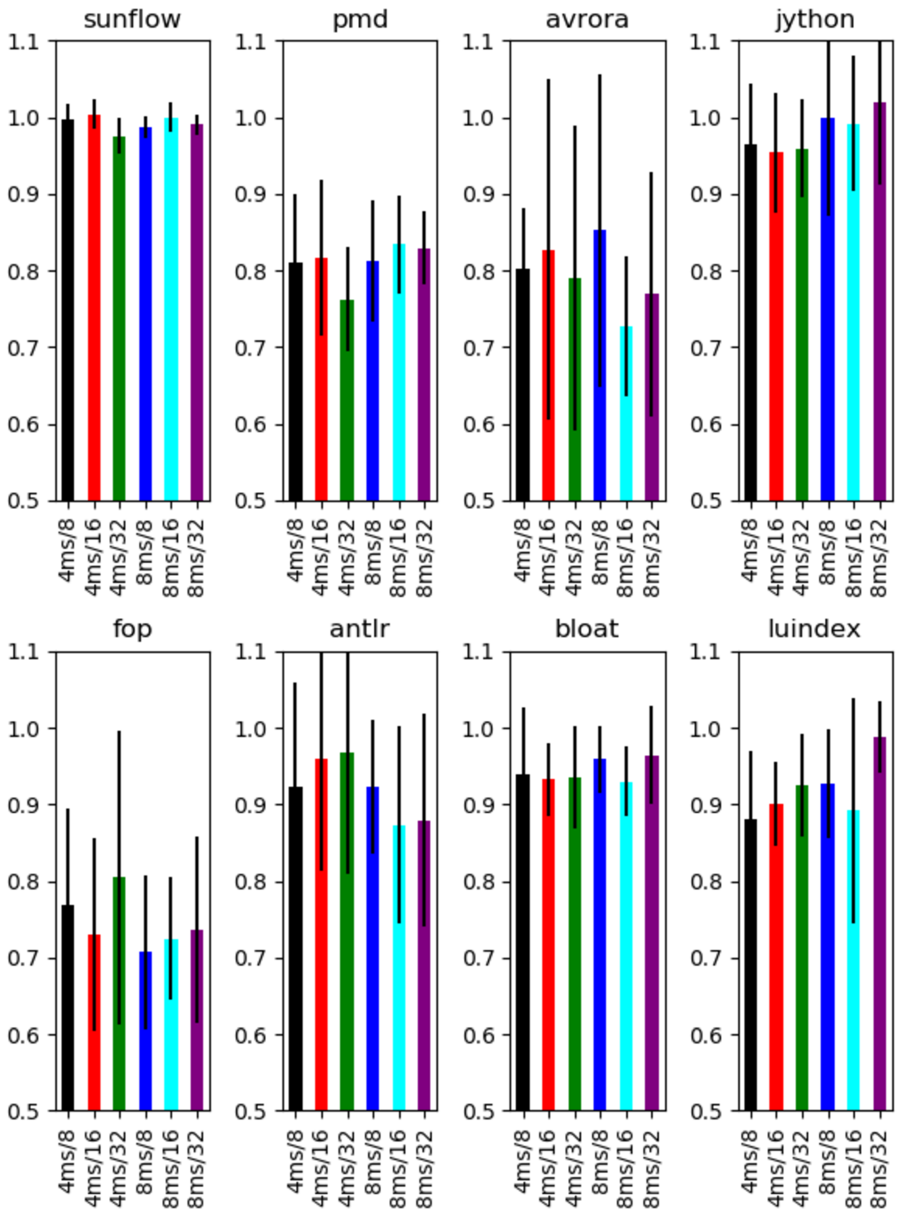

Figure 8: Vincent Least Energy Consumption Under Different Sampling Settings Against the `ONDEMAND` Baseline (Under each bar, "Xms/Y" means `EPOCH` $\times$ `SN` $= X$ and `SAMPLENUM` $= Y$. Each bar shows the least energy consumption among all combinations of methods and CPU frequencies for that benchmark, i.e., among all cells in a heatmap such as produced in Fig. 5. Each result is the average of multiple runs according to the discussion in § 4 under that setting. The thin bar on each bar shows standard deviation. For all bars, shorter is better.)

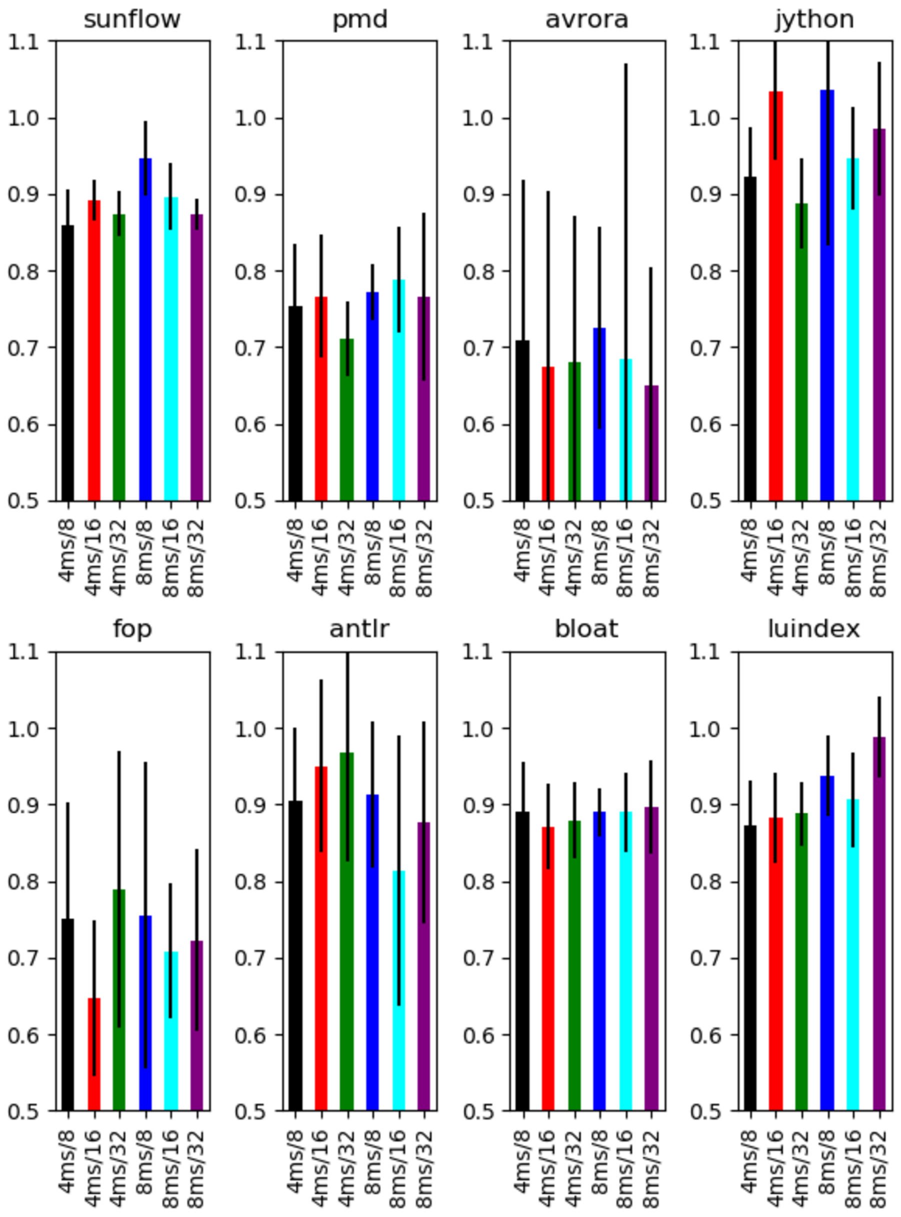

Figure 9: VINCENT Least EDP Under Different Sampling Settings Against the `ONDEMAND` Baseline (All legends are identical to those in Fig. 8. For all bars, shorter is better.)

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 remains to be the benchmark itself. For example, `pmd`, `avrora`, and `fop` can lead to the most energy savings, and these

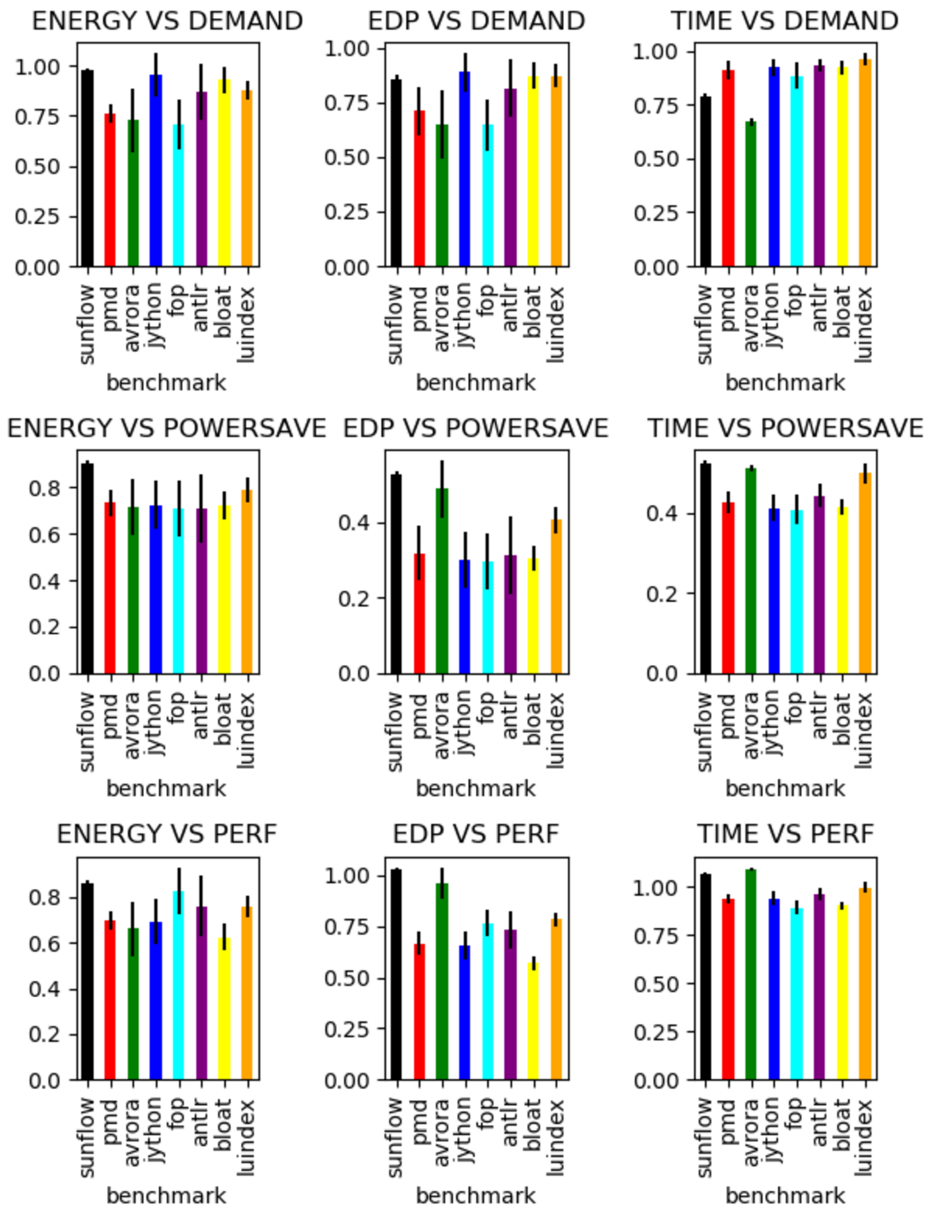

Figure 10: VINCENT Best Results against Different Governor Baselines (The first row shows results normalized against the `ONDEMAND` governor. The second row shows results normalized against the `POWERSAVE` governor. The third row shows results normalized against the `PERFORMANCE` governor. For all bars, shorter is better.)

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` — 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, `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 DVFS overhead.

5.3. 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. 10, 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 energy-efficient than the `PERFORMANCE` governor. Note that in the last row of Fig. 10, 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 surprising fact is that the VINCENT runs for some benchmarks can in fact lead to a small but noticeable reduction in the execution time than their counterpart `PERFORMANCE` runs. When the `PERFORMANCE` governor is used to regulate DVFS on Intel architectures where P-States are available, the highest power state is used. Note however the highest power state is not tantamount to the highest CPU frequency [32, 1]. Recall that (§ 2) P-States are managed at the level of the CPU package, not at the level of individual cores. How the supply voltage and the CPU frequencies of individual cores are assigned given a power state subjects to a variety of design constraints, such as area power and thermal considerations. The DVFS of VINCENT however is targeted at the core level: when a method is determined to run with the highest CPU frequency, the CPU core hosting the thread in which the method runs is set at the highest CPU frequency. This interesting phenomenon may indicate a potential for performance optimization, but there are caveats. First, the average performance improvement is small: only a subset of benchmarks can benefit, while there is degradation in others (Fig. 10). Second, as P-State maintenance is a platform-dependent black-box hardware feature, the phenomenon may be restricted to specific architectures (Broadwell in our case), and may no longer presents itself in other architectures.

5.4. The Impact during the Warm-Up Phase

The data we have shown so far result from the last 15 runs in a 20-run execution for each benchmark (see § 4), i.e., the post-warmup runs. This evaluation choice is in sync with the general focus of energy optimization on long-running applications, where energy consumption matters the most. In those server-class settings, a `sunflow` application will continuously process images (instead of a fixed number necessitated by the benchmark), and an `xalan` application will continuously process XML documents (instead of a fixed number of documents).

For completeness, we now describe the result of the first 5 runs in a 20-run execution, with the per-benchmark results shown in Fig. 11. Overall, VINCENT remains an effective optimizer relative to the 3 baselines. Nearly all benchmarks retain the similar trend as post-warmup runs in Fig. 10.

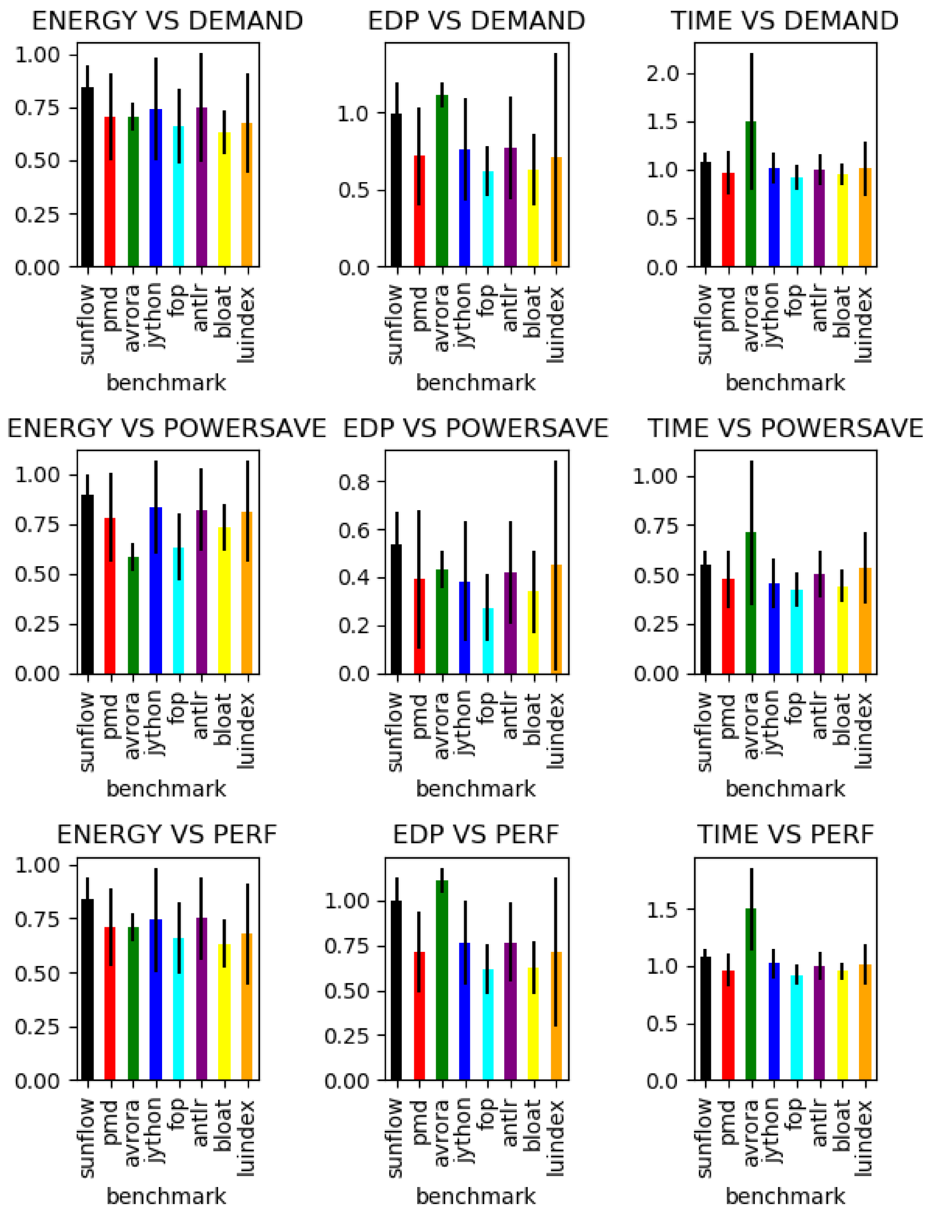

Figure 11: VINCENT Best Results against Different Governor Baselines for the First 5-Runs (All legends are otherwise identical to Fig. 10)

Relative to the latter however, the results exhibit a larger deviation. As the majority of hot methods are identified in the earlier runs, the combined 5-run results shown here demonstrate that VINCENT has already started to play an effective role in the optimization. Note however, the hot method selection process in JVMs is incremental: some hot methods may be identified during the first run, whereas others may be deferred to the later runs. As a result, the effectiveness of VINCENT relative to the 3 baselines is only incrementally more pronounced, leading to larger deviation across the 5 runs.

5.5. 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. 6), `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 — DVFS is implemented as blocking I/O writes — an extensive increase in execution time ensues, bad news for energy efficiency. The multi-method result is a 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.

5.6. Overhead

Finally, we quantify the overhead introduced by DVFS in VINCENT. Similar to all DVFS-based approaches, this is a scenario where DVFS calls themselves may come with a small cost of energy consumption, but in return, the overall system can (often) save energy. The purpose of this discussion is to illuminate the optimization space: if an application under VINCENT saves X% energy compared against a baseline but the DVFS calls for that run come with a Y% energy overhead, then VINCENT could *theoretically* save X+Y% of energy if the DVFS calls were further optimized at the hardware/system level.

We compute the energy overhead as $\texttt{EUNIT} \times \texttt{DVFSNUM}_b$, where `EUNIT` is the energy overhead of each DVFS call, and $\texttt{DVFSNUM}_b$ is the number of DVFS calls in the optimized run for benchmark b. To measure `EUNIT`, we created a simple multi-threaded program where the number of threads is the number of CPU cores (`CNUM`), and each thread is in a continuous loop performing `LNUM`

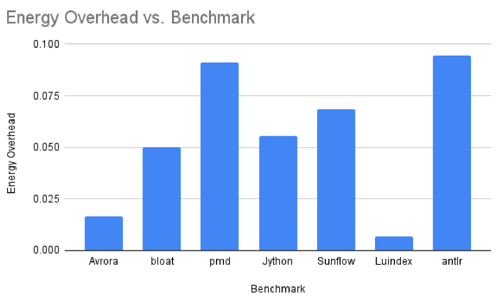

Figure 12: Normalized Energy Overhead

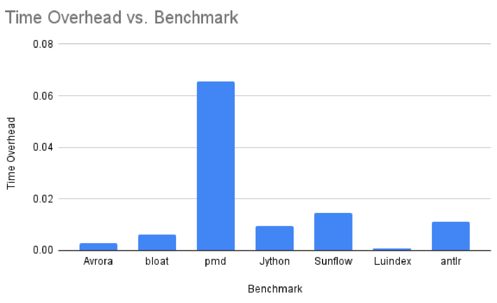

Figure 13: Normalized Time Overhead

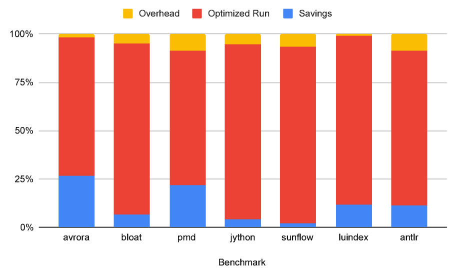

Figure 14: Energy Overhead and Savings (normalized against the energy consumption of the `ONDEMAND` run)

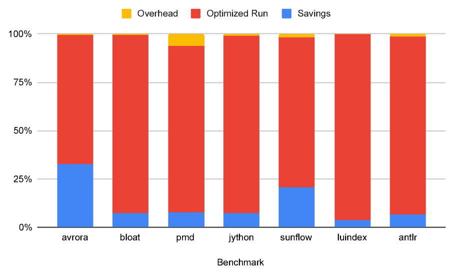

Figure 15: Time Overhead and Savings (normalized against the energy consumption of the `ONDEMAND` run)

number of DVFS calls. We compute `EUNIT` as $\frac{\texttt{ESUM}}{\texttt{LNUM} \times \texttt{CNUM}}$, where `ESUM` is the overall energy consumption of the program. According to our experiment, $\texttt{EUNIT} = 0.0006489615 05664$J. In a similar fashion, we also compute the time overhead of each DVFS call, $\texttt{TUNIT} = 2.53056888702\ \mu$s. Observe that both the `EUNIT` calculation and the `TUNIT` calculation have taken the effect of multi-threading into account.

The overall normalized energy overhead is shown in Fig. 12 and the overall normalized time overhead is shown in Fig. 13. Note that both results are normalized, i.e., in percentage. Overall, the average energy overhead is around 5.47% and the average time overhead is around 1.58%. To risk stating the obvious, we wish to reiterate that VINCENT does save energy for most benchmarks; the result here simply means that VINCENT uses a small amount of energy to save more energy for most benchmarks. To illustrate this point, we plot the overhead of VINCENT together with the savings of VINCENT (relative to the `ONDEMAND` run). Fig. 14 shows the energy consumption, and Fig. 15 shows the execution time.

5.7. An Experimental Summary

Fig 16 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.

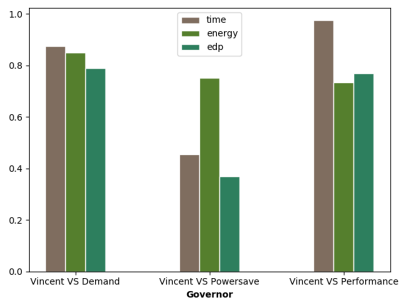

Figure 16: A Summary of Results with Different Governor Baselines (In each group, the energy/EDP/time data are normalized with their corresponding data under a built-in governor based on dynamic monitoring. For all bars, being shorter means VINCENT is more effective than the built-in governor.)

6. Lessons Learned

The development and evaluation of VINCENT is a rewarding experience. The final design is simple in our opinion, which may be good news for adoption. In retrospect however, we spent a large amount of time developing features that did not make it to the final design. We now document some of these efforts, hopefully useful to future designers.

The Power of Timer-Based Sampling. The first strawman approach in § 3.2.2 was implemented in the early stage of our development. Both the RAPL-based profiling algorithm and the DVFS-based scaling algorithm performed magnitudes slower than the baseline runs. We further implemented some optimizations of this strawman approach. As one example, we added a counter to each pair of prologue and epilogue (for a hot method), decremented it each time a prologue is encountered. A RAPL reading (during the profiling phase) or a DVFS call (during the scaling phase) is only performed when the counter can be divided by a constant C. In essence, it only performs $\frac{1}{C}$ of the RAPL/DVFS calls. This implementation led to divergent results that vary greatly across benchmarks. We think the root problem of that approach is its blindness to the inherent difference among methods, in terms of how often each is called, and how long each execution takes. For a method that is encountered once every U microseconds, this design would imply that RAPL/DVFS call is performed once per $U \times C$ microseconds. If $C = 10$, the overhead due to RAPL/DVFS may be too large for a method where $U = 10$, while RAPL/DVFS may be performed too infrequently for a method where $U = 10000$. In other words, there is no universal C that can fit the need of all methods.

As a next step, we further considered a variant where the value U is profiled for each method, so that the C value may also be method-specific. This optimization did not lead to acceptable results either, because the value of U indeed took on *phased behaviors*: during the execution of an application, it may change during each phase.

VINCENT currently adopts timer-based sampling. Here, the number of samples taken is managed by the sampling rate and the counter value we choose. The overhead is no longer correlated to method-specific characteristics (such as U above). Given the influence of this general approach in JIT design, it may be self-evident — in hindsight — that we should have explored it in the first place.

The Occam's Razor. The main reason behind our initial hesitation with the timer-based sampling approach is its *approximate* nature. Given that energy values in RAPL registers are accumulative, we need to correlate two samples to compute the difference. This is a unique problem unseen in the sampling-based design of e.g., JIT.

We spent a large amount of time implementing features to correlate samples. As one example, we attempted to record the name of methods when a sample is taken. To further account for recursion, we instrumented a counter in the prologue/epilogue, so that each taken sample can also record the level of recursion. Finally, a difference is only taken when the method names and recursion levels match. These implementations led to poor experimental results, for two reasons. First, the requirement of matching method names and/or recursion levels significantly reduces the number of energy readings we could collect: given the sampling nature of the approach, many temporally adjacent samples do not have matching method names or recursion levels. Second, and more importantly, the overhead of bookkeeping — querying/recording method names and/or tracking recursion levels — quickly adds up. As energy is the multiplication of power and time, any modification that significantly slows down the execution leads to poor energy efficiency. When the metric of EDP is considered, the time overhead becomes even less desirable.

In retrospect, we converged on the refutation-based delimited approach (see § 3.2) through "elimination": we removed a number of features that we thought would improve precision, one by one, and the effectiveness of our approach continued to improve. A similar path of exploration happened when we looked into the potential of multi-method optimization.

Overall, we believe it remains an interesting future direction to further improve the precision in our sampling approach, *if such improvement does not introduce overhead.*

7. Related Work

Compiler-Directed or Runtime-Directed DVFS. The underlying philosophy of our work — *programs* matter for DVFS-based energy optimization — is shared among a number of compiler-directed energy optimization approaches. Saputra et al. [51] describes a DVFS-based approach at the level of compiler optimization. Their algorithm first observes the potential speed-up of loop transformation (e.g., tiling and loop fusion) over the unoptimized program, and then scales the CPU voltage and frequency down over the optimized program to a desirable level that matches the original execution time of the unoptimized program, through integer linear programming. Hsu and Kremer [29] defines a compiler-directed DVFS algorithm where a desirable CPU frequency is selected for running a code region; the selection is based on solving a minimization problem where the need for limited performance loss is encoded as constraints. Xie et al. [60] defines an analytical model — built in the compilation process — where energy minimization is reduced to a mixed-integer linear programming problem. Overall, the previous work focused on building *analytical models* in the presence of DVFS. This general direction, building analytical models to identify slacks in programs, can be traced back to a classic analysis for energy-efficient OS scheduling [58].

A small body of work further extends analytical models to virtual machines and dynamic compilation. In Haldar et al. [25], methods are instrumented with DVFS calls, and the frequency of choice when a method executes is based on the comparison among the projected energy consumption of the method at different frequencies. To make this decision, it was necessary for their analytical algorithm to introduce heuristics (that may no longer hold for state-of-the-art application workloads), such as the projected future execution time is the same as the execution time so far, and the execution time increases linearly with the CPU frequency slowdown. Wu et al. [59] proposed a dynamic compilation framework for C programs, where important code regions such as loops are manually identified and instrumented, and the CPU frequency for DVFS is selected based on an analytical model. Relative to Haldar et al., their model addressed the non-linear effect of DVFS on execution time: through analyzing the memory-related instructions in the code region, their algorithm projects smaller performance loss for memory-intensive code regions when the CPU frequency is scaled down.

As both Haldar et al. and Wu et al. are runtime-level efforts, a more in-depth comparison is warranted. First, VINCENT does not rely on an analytical model to estimate or extrapolate the execution time or energy effect of DVFS, and does not need to instantiate the often unknown parameters in the analytical model through heuristics. Second, VINCENT identifies the most energy-consuming methods in an automated process. In contrast, the code region for DVFS in Wu et al. is manually identified, Third, both existing efforts centrally relied on instrumenting method boundaries for DVFS calls. Acceptable performance may be achievable at the era of these developments — e.g., Haldar et al. was evaluated against the Java Grande benchmark suite [55] and Wu et al. against SPEC 95 and SPEC2K — but modern Java applications are significantly more complex than e.g., `heapsort` in Java Grande. In § 3.2.2, we described the high overhead of that approach for Dacapo benchmarks.

In the context of related work, VINCENT can be understood as a revisit to a historically significant research direction — compiler/runtime-based DVFS — which has unfortunately been overtaken by black-box approaches e.g., DVFS based on dynamic performance counters. VINCENT defines an end-to-end approach that is *simple* (no analytical model), *automated* (no manual efforts in code region identification), and *scalable* in overhead (no instrumentation for DVFS). It is our hope that VINCENT is a new beginning to re-study this largely overlooked direction in the presence of modern applications in managed runtimes.

Energy-Aware Languages. Another direction of energy optimization at the boundary of programming abstractions is energy-aware programming languages [56, 11, 50, 27, 40, 20, 12, 35, 26, 41, 62, 16]. For example, Energy Types [20] introduces DVFS at the boundary of methods based on phase information declared by programmers or inferred by the compiler. Green [11] and LAB [35] select alternative algorithm-specific parameters based on energy and QoS need. Ent [15] 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.

Runtime-Level Energy Efficiency. Chen et al. [19] relies on garbage collection tuning to save memory system energy consumption in JVMs. Cao et al. [17] improves the energy efficiency of JVM by assigning JVM services to small cores on asymmetric hardware. `DEP+BURST` [2] 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. [30] 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 [48, 39] and Aequitus [49] are energy-efficient solutions built on top of Cilk. They perform DVFS based on the dependencies between thief threads and victim threads in work stealing runtimes.

Empirical studies often illuminate the energy consumption (and performance) of managed language runtimes. An early study by Vijaykrishnan et al. [57] focuses on the energy consumption impact on the memory hierarchy (cache and main memory) by JIT-enabled Java applications. Esmaeilzadeh et al. [22] studies energy efficiency with a focus on diverse configurations of workload and hardware. Sartor and Eeckhout [52] 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. [46] 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 [36], previous studies [45] 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 were also a subject of studies [24, 47]. Kambadur et al. [34] takes a cross-layer approach to surveying the energy management solutions, studying the interface and interaction of different hardware/OS/compiler configurations.

Energy Profiling. Energy profiling is more commonly conducted at the system level (e.g., [44, 23]), rather than at the boundary of programming abstractions such as methods. Jalen [43] is an energy profiler relying on program-level bytecode instrumentation and static compilation. To rein in on overhead, Jalen also relies on sampling. For each method of interest identified by the user, Jalen produces its energy consumption, one run for each method. Instead of relying on RAPL to query energy consumption, Jalen is endowed with an analytical power model that takes into account of a wide range of system states, such as CPU frequencies and network status. Thanks to the sophisticated power modeling, Jalen can profile the energy consumption of networks, which VINCENT does not consider. Chappie [10] also supports method-grained energy profiling. Chappie runs as a separate thread to the monitored application, and continuously samples the method at the top of the call stack at fixed time intervals. Like VINCENT, Chappie does not require users to identify the method to be profiled. VINCENT is fundamentally a JVM-centric approach. It takes advantage of the JVM support such as dynamic compilation and instrumentation to enable delimited sampling. In one profile run, it produces the energy consumption values of all hot methods. Broadly, the relationship between existing energy profilers and VINCENT is complementary. On the one hand, energy profiling is an intermediate step for energy optimization in VINCENT, which Jalen or Chappie does not support. It demonstrates that profiling and optimization can be unified in one framework, in a coherent JVM-based approach. On the other hand, VINCENT can indeed further borrow ideas from existing energy profilers (such as network energy profiling in Jalen, or multi-application support in Eflect [9]) to obtain comprehensive energy profiles in the complex systems.

8. Threats to Validity

While we believe leveraging hot methods in the JVM for DVFS-guided energy optimization is a generalizable idea, VINCENT as an experimental system is implemented and evaluated within specific software/hardware environments. The validity of our experimental data is restricted to these environments.

First, VINCENT is an extension to the JikesRVM, so the validity of our results can only be safely confirmed in that JVM. We are hopeful that the ideas behind VINCENT can translate to alternative JVMs, for several reasons. (1) VINCENT does not rely on unique JikesRVM features; hot method selection, dynamic instrumentation and compilation, and counter-based sampling are available in many JVMs; (2) To the best of our knowledge, alternative JVMs widely in use today do not perform DVFS-specific optimizations, so the likelihood of feature intervention is small if the idea behind VINCENT is adopted on them. (3) JikesRVM has incubated other influential JVM ideas (e.g., JIT, garbage collection), whose effectiveness has been confirmed in alternative JVMs.

Second, VINCENT relies on CPU architectures where DVFS is enabled. Fortunately, DVFS is a standard feature whose support is the rule not the exception in commodity CPUs, including the vast majority of chips from Intel, AMD, ARM, and others. RAPL is used for VINCENT energy measurement, a hardware feature also widely available in Intel after 2011, and more recently, AMD CPUs.

Third, the experimental results are limited to the benchmark suite we used, Dacapo. Dacapo is commonly used for Java evaluating the performance of JVMs and Java applications. The benchmarks we used are multi-threaded, and they have diverse workload characteristics (CPU-bound vs. I/O-bound) that matter to energy optimization.

As for the OS governor support, note that the `ONDEMAND`, `PERFORMANCE` and `POWERSAVE` governors are used for the purpose of evaluation. The only OS requirement for Vincent is that the OS can expose the capability of DVFS regulation to the application. This is the `USERSPACE` governor in Linux. Such support is also available in other OS such as Windows [37].

9. 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 profile-directed optimization guided by DVFS. Our experiments show Vincent can reduce the energy consumption and improve the energy efficiency of Java applications. Vincent is a novel instance among a small number of energy optimization approaches that take advantage of the information available to the managed runtime. It requires no modification to the underlying OS/hardware, and requires no programmer effort.

Acknowledgments

This project is supported by the US NSF award CNS-1910532.

References

[1] Kristen Accardi. Balancing power and performance in the linux kernel, `https://events.static.linuxfound.org/sites/events/files/slides/LinuxConEurope_2015.pdf`. In *The 2015 Linux Conference*, 2015.

[2] 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. https://doi.org/10.1109/TC.2016.2609903 `doi:10.1109/TC.2016.2609903`.

[3] 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. https://doi.org/10.1145/320385.320418 `doi:10.1145/320385.320418`.

[4] 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. https://doi.org/10.1147/sj.391.0211 `doi:10.1147/sj.391.0211`.

[5] The Linux Kernel Archives. Intel p-state driver, `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`.

[6] The Linux Kernel Archives. Linux cpufreq governors, `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.

[7] 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.

[8] 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, Languages, and Applications*, OOPSLA '00, page 47–65, New York, NY, USA, 2000. Association for Computing Machinery. https://doi.org/10.1145/353171.353175 `doi:10.1145/353171.353175`.

[9] Timur Babakol, Anthony Canino, and Yu David Liu. Eflect: Porting energy-aware applications to shared environments. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 823–834, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3510003.3510145 `doi:10.1145/3510003.3510145`.

[10] 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.

[11] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pages 198–209, 2010.

[12] Thomas Bartenstein and Yu David Liu. Green streams for data-intensive software. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, May 2013.

[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. https://doi.org/10.1145/1167473.1167488 `doi:10.1145/1167473.1167488`.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1840845.1840883`, https://doi.org/10.1145/1840845.1840883 `doi:10.1145/1840845.1840883`.

[22] 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. https://doi.org/10.1145/1950365.1950402 `doi:10.1145/1950365.1950402`.

[23] 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.

[24] 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.

[25] Vivek Haldar, Christian W. Probst, Vasanth Venkatachalam, and Michael Franz. Virtual-machine driven dynamic voltage scaling. Technical report, In Technical Report No.03-21, SICS, 2003.

[26] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, 2015.

[27] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*, 2011.

[28] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, 1994.

[29] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI'03*, pages 38–48, 2003.

[30] 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.

[31] Intel. Energy analysis user guide, available at `https://www.intel.com/content/www/us/en/develop/documentation/energy-analysis-user-guide/`.

[32] Intel. Intel 64 and ia-32 architectures software developer's manual: Volume 3.

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

[34] 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. https://doi.org/10.1145/2660193.2660196 `doi:10.1145/2660193.2660196`.

[35] 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, 2013.

[36] 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. https://doi.org/10.1145/337449.337465 `doi:10.1145/337449.337465`.

[37] Bin Lin, Arindam Mallik, Peter Dinda, Gokhan Memik, and Robert Dick. User- and process-driven dynamic voltage and frequency scaling. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 11–22, 2009. https://doi.org/10.1109/ISPASS.2009.4919634 `doi:10.1109/ISPASS.2009.4919634`.

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

[39] Yu David Liu. Green thieves in work stealing. In *Proceedings of ASPLOS'12 (Provactive Ideas session)*, 2012.

[40] Yu David Liu. Variant-frequency semantics for green futures. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'12)*, 2012.

[41] 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.

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

[43] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. *Automated Software Engg.*, 22(3):291–332, sep 2015. https://doi.org/10.1007/s10515-014-0171-1 `doi:10.1007/s10515-014-0171-1`.

[44] 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.

[45] 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.

[46] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA '14*, 2014.

[47] 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.

[48] 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.

[49] Haris Ribic and Yu David Liu. AEQUITAS: coordinated energy management across parallel applications. In *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*, pages 4:1–4:12, 2016.

[50] 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*, 2011.

[51] H. Saputra, M. Kandemir, N. vijaykrishan, M Irwin, J. Hu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *In Proc. ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems*, pages 2–11. ACM Press, 2002.

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

[53] 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.

[54] 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.

[55] L.A. Smith, J.M. Bull, and J. Obdrizalek. A parallel java grande benchmark suite. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 6–6, 2001. https://doi.org/10.1145/582034.582042 `doi:10.1145/582034.582042`.

[56] 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, 2007.

[57] 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.

[58] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[59] Qiang Wu, V.J. Reddi, Youfeng Wu, Jin Lee, D. Connors, D. Brooks, M. Martonosi, and D.W. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *38th Annual IEEE/ACM International Sympo-*

sium on Microarchitecture (MICRO'05), pages 12 pp.–282, 2005. https://doi.org/10.1109/MICRO.2005.7 `doi:10.1109/MICRO.2005.7`.

[60] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI'03*, pages 49–62, 2003.

[61] 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.

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