# Static and Dynamic Frequency Scaling on Multicore CPUs

WENLEI BAO and CHANGWAN HONG, The Ohio State University
SUDHEER CHUNDURI, IBM Research India
SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory
LOUIS-NOËL POUCHET, Colorado State University
FABRICE RASTELLO, University Grenoble Alpes
P. SADAYAPPAN, The Ohio State University

Dynamic Voltage and Frequency Scaling (DVFS) typically adapts CPU power consumption by modifying a processor's operating frequency (and the associated voltage). Typical DVFS approaches include using default strategies such as running at the lowest or the highest frequency or reacting to the CPU's runtime load to reduce or increase frequency based on the CPU usage. In this article, we argue that a compile-time approach to CPU frequency selection is achievable for affine program regions and can significantly outperform runtime-based approaches. We first propose a lightweight runtime approach that can exploit the properties of the power profile specific to a processor, outperforming classical Linux governors such as `powersave` or `on-demand` for computational kernels. We then demonstrate that, for affine kernels in the application, a purely compile-time approach to CPU frequency and core count selection is achievable, providing significant additional benefits over the runtime approach. Our framework relies on a one-time profiling of the target CPU, along with a compile-time categorization of loop-based code segments in the application. These are combined to determine at compile-time the frequency and the number of cores to use to execute each affine region to optimize energy or energy-delay product. Extensive evaluation on 60 benchmarks and 5 multi-core CPUs show that our approach systematically outperforms the `powersave` Linux governor while also improving overall performance.

CCS Concepts:• **Software and its engineering** → **Power management**; **Compilers**; • **Hardware** → *Chip-level power issues;*

Additional Key Words and Phrases: Static Analysis, Voltage and Frequency Scaling, CPU Energy, Affine Programs

**ACM Reference Format:**
Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. Static and dynamic frequency scaling on multicore CPUs. ACM Trans. Archit. Code Optim. 13, 4, Article 51 (December 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/3011017

---

This work was supported in part by the U.S. National Science Foundation, award 1524127; by the U.S. Department of Energys (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award 63823 and DE-SC0014135. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830. We would like to thank Charles Lefurgy for his guidance in using the AMESTER tool and power monitoring on POWER8 nodes.
Authors' addresses: W. Bao, C. Hong, and P. Sadayappan, Department of Computer Science and Engineering, The Ohio State University, 395 Dreese Lab, 2015 Neil Avenue, Columbus, Ohio 43210; emails: bao.79@osu.edu, hong.589@osu.edu, saday@cse.ohio-state.edu; S. Krishnamoorthy, PO Box 999 MSIN J4-30, Richland, WA 99352; email: sriram@pnnl.gov; S. Chunduri, IBM Research, India (now at Argonne National Lab, 9700 S. Cass Avenue Lemont, IL 60439); email: sudheer@anl.gov; L.-N. Pouchet, Computer Science Department, Colorado State University, 1873 Campus Delivery, Fort Collins, CO 80523-1873; email: pouchet@colostate.edu; F. Rastello, Univ. Grenoble Alpes, Inria, CNRS, LIG, F-38000, Grenoble France; email: fabrice.rastello@inria.fr.

DOI: http://dx.doi.org/10.1145/3011017

## 1. INTRODUCTION

Energy efficiency is of increasing importance in a number of use cases ranging from battery-operated devices to data centers striving to lower energy costs. Dynamic Voltage and Frequency Scaling (DVFS) [Farkas et al. 2000] is a fundamental control mechanism in processors that enables a tradeoff between performance and energy.

Typical DVFS approaches fall into two categories: schemes that employ specific frequencies for default policies (e.g., `powersave` vs `performance` governors in Linux) or schemes that observe the CPU execution and dynamically react to CPU load changes (e.g., the `on-demand` governor). Although energy savings have been demonstrated with these approaches [Ge et al. 2007; Hsu and Kremer 2003; Li and Martinez 2006; Jimborean et al. 2014], we observe several limitations. First, as we show in this article, the frequency/voltage that optimizes CPU energy can vary significantly across processors. Even for the same compute-bound application, different processors have different energy-minimizing frequencies. Second, optimizing energy for a parallel application is highly dependent on its parallel scaling, which in turn depends on the operating frequency, an aspect mostly ignored in previous work. Third, dynamic schemes remain constrained to runtime inspection at specific time intervals, implying that short-running program phases (e.g., a few times longer than the sampling interval) will not see all the benefits of dynamic DVFS compared to using the best frequency for the phase from the start.

In this work, we propose to address these limitations using two complementary strategies. First, we present a simple lightweight runtime which throttles frequency based on periodic measurements of the energy efficiency of the application. This approach can exploit the specific properties of the CPU power profile and is applicable to arbitrary programs. Then, we develop a compile-time approach to select the best frequency but for program regions that can be analyzed using the polyhedral model [Feautrier 1992], which is typical of compute-intensive kernels/library calls such as BLAS operations [Netlib; pol] or image processing filters [OpenCV; pol]. Specifically, we develop static analysis to approximate the *operational intensity* (i.e., the ratio of operations executed per bytes of data transferred from the RAM) and the *parallel scaling* (i.e., the execution time speedup as a function of the number of cores) of a program region in order to *categorize* that region (e.g., compute-bound, memory-bound, etc.). The frequency and number of cores to use for each program category are chosen from the result of a one-time energy and energy-delay product profiling of the processor using microbenchmarks representative of different workload categories. Our extensive evaluation demonstrates significant energy and EDP savings over static (`powersave`) and dynamic (`on-demand`) schemes, thus validating our approach. We make the following contributions:

—We demonstrate that some limitations of purely load-based approaches to DVFS can be addressed using a lightweight runtime approach optimizing for CPU energy efficiency.
—We develop a compilation framework for the automatic selection of the frequency and number of cores to use for affine program regions by categorizing a region based on its approximated operational intensity and parallel scaling potential.
—We provide extensive evaluation of our approach using 60 benchmarks and 5 multicore CPUs, demonstrating significant energy savings and EDP improvements over the `powersave` Linux governor.

The article is organized as follows: Section 2 demonstrates the variability in optimizing for energy efficiency on CPUs. Section 3 presents our lightweight DVFS runtime approach. Section 4 presents our compile-time framework to compute program features

Table I. Processor Characteristics

| Intel CPU | Microarch. | Node | Cores | L1 | L2 | L3 | Freq. Range | Voltage Range |
|---|---|---|---|---|---|---|---|---|
| Core i7-2600K | SandyBridge | 32nm | 4 | 32K | 256K | 8192K | 1.6 ∼ 3.4 GHz | 0.97 ∼ 1.25 V |
| Xeon E3-1275 | IvyBridge | 22nm | 4 | 32K | 256K | 8192K | 1.6 ∼ 3.5 GHz | 0.97 ∼ 1.09 V |
| Xeon E5-2650 | IvyBridge | 22nm | 8 | 32K | 256K | 20480K | 1.2 ∼ 2.6 GHz | 0.97 ∼ 1.09 V |
| Core i7-4770K | Haswell | 22nm | 4 | 32K | 256K | 8192K | 1.2 ∼ 3.5 GHz | 0.76 ∼ 1.15 V |

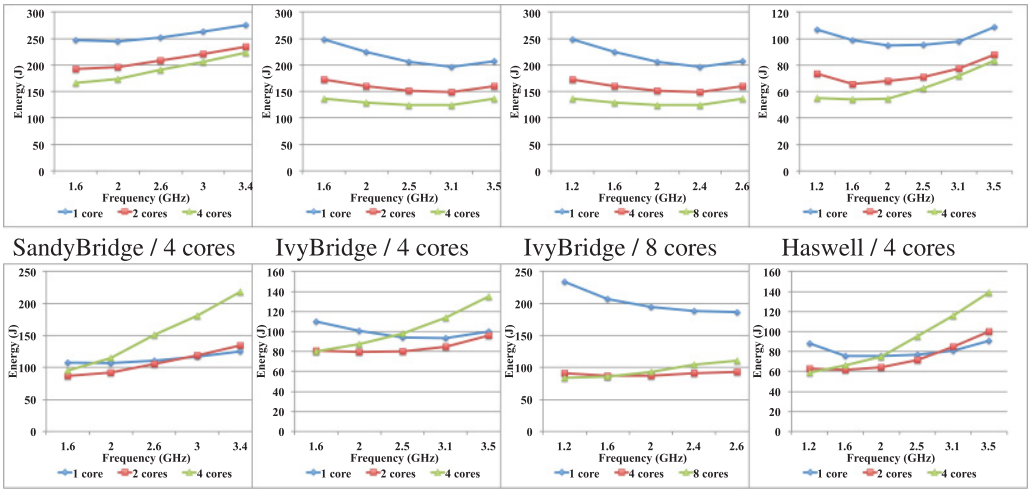


Fig. 1. Energy for DGEMM/MKL (top row) and Jacobi 2D (bottom row).

used to subsequently select the best frequency/core configuration for a program region, as detailed in Section 5. Sections 6 and 7 present extensive experimental results on 60 benchmarks and 5 multicore CPUs. Related work is presented in Section 8 before concluding.

## 2. MOTIVATION AND OVERVIEW

Previous research has shown that the increase in execution time when throttling down the frequency can be limited to a very small quantity by performing careful DVFS, leveraging the fact that, on bandwidth-bound applications, the processor frequently stalls waiting for data transfer. Therefore, processor frequency could be reduced without significant wall-clock time increase (if the latency of main memory operations is not affected by DVFS), resulting in energy savings [Ge et al. 2007; Hsu and Kremer 2003].

Moreover, when considering the CPU energy alone (in isolation from the rest of the system), it could be intuitive to think that the lower the frequency, the lower the energy consumption: Because power is often approximated to have a cubic relationship with frequency (assuming frequency and voltage have a linear relationship), using the minimal frequency (e.g., as with `powersave`) is expected to increase execution time linearly but decrease power in a nearly cubic way, thereby leading to minimal CPU energy. We now show that this is not always true.

*Energy profiles of CPUs*. Here, we discuss in detail the energy profiles of four off-the-shelf Intel x86 CPUs running on a Linux desktop. Table I outlines their key characteristics. We report in Figure 1 a series of plots obtained by actual power and time measurements using hardware counters available with Intel PCM [Intel b], on DGEMM/MKL [Intel a] for the top row. DGEMM/MKL is run on a large out-of-L3 problem size (reported to achieve near peak compute performance on these machines), and its execution

time scales nearly perfectly with the frequency and number of cores used: It captures a compute-bound scenario exploiting fully all the processor capabilities. The data plotted are the measured CPU energy for the computation where only one benchmark is running (no co-execution) and where Turbo-boost and other hardware DVFS mechanism have been turned off to obtain deterministic data. Each frequency was set prior to benchmarking using the `userspace` governor using the cpuFreq package, implicitly changing voltage along with frequency change as per the CPU specification.

We make several key observations from these data. First, *the optimal CPU energy is not achieved for the minimal or maximal frequency* in most cases. Taking the case where all cores are used, on SB-4 and HSW-4, minimal energy is achieved at 1.6GHz (the minimal frequency for our SB setup), but it is achieved near the (but not at) maximal frequency for the two Ivy Bridge processors. The reason relates to the voltages used and, in particular, to the ratio of voltage changes versus the ratio of frequency changes for each machine. Table I shows that, for the two Ivy Bridge CPUs, the voltage range (from 0.97V to 1.09V) is much smaller than for the other two CPUs. Overly simplified power equations ignore key effects such as the relation between leakage current and temperature and frequency and voltage relations. A more realistic power equation [Skadron et al. 2004] captures the Poole-Frenkel effect, which relates leakage to temperature, and careful derivation of the evolution of the power equation as a function of changes in V, f, and Temp demonstrates that the slope of increase of voltage versus frequency can influence the optimal frequency for energy efficiency. De Vogeleer et al. [2014] developed an analogous characterization on a mobile processor, modeling the energy variation between frequency steps as a function of voltage and temperature and obtaining a curve similar to our result for Haswell. They derived formulas to characterize the convex shape of the CPU energy efficiency for a fixed workload as a function of CMOS characteristics including voltage and frequency increase relations. Here, we observe across four different x86 Intel processors four different cases for the most energy-efficient frequencies for compute-bound codes. A runtime approach focusing only on workload properties (e.g., the lack of stall cycles) and not taking into account these processor-specific effects would fail to select the optimal frequency for CPU energy minimization.

Second, *the optimal CPU energy may be achieved at different frequencies depending on the number of cores used*. This is seen in particular on Haswell, where on one core 2GHz is the best frequency, but for two and four cores it is 1.6GHz. A similar situation is observed on Sandy Bridge, albeit the difference is very small. On the other hand, this is not observed for the Ivy Bridge cases.

To make the situation more complex, in practice, the most energy-efficient frequency is also affected by how the execution time evolves as a function of frequency. When the execution time decreases at a slower rate than the frequency increases (e.g., the expected acceleration is not achieved), this shifts the optimal frequency toward lower values than for the compute-bound cases like DGEMM/MKL. This is exemplified with a bandwidth-bound benchmark, as shown in the bottom row of Figure 1. J2D is a Jacobi 2D code from PolyBench/C 3.2 which is parallelized naively among rows of the image and uses out-of-L3 data, too. It represents a bandwidth-bound case which is more realistic (e.g., less exacerbated) than the STREAM [McCalpin 2007] benchmark.[1] We see a systematic shift of the most energy-efficient frequency toward the left; that is, lower frequencies. In addition, due to bandwidth saturation effects, the code does not have good weak scaling: Adding cores does not decrease the execution time linearly. It leads to higher energy consumption increase for the four- or eight-core cases than

[1]STREAM suffers from insufficient number of loads emitted at low frequencies on single-core: It did not always saturate the bandwidth in our experiments.

for single-core, when the frequency increases. *This motivates the need for an approach that also considers the application characteristics to determine its most energy efficient frequency*.

*Proposed approach*. Based on the preceding observations, we conclude that the best frequency to use to minimize CPU energy is per-processor, per-workload specific: One cannot be limited to looking at the CPU load (e.g., using `on-demand`) or using the minimal frequency (e.g., using `powersave`) to minimize CPU energy. We propose two approaches to address this problem: (i) A lightweight runtime that adapts the frequency based on the CPU energy changes dynamically during the application execution (see Section 3); and (ii) a compile-time approach for static selection of the ideal frequency for each affine computation kernel within a full application based on (a) a new static analysis of the program's operational intensity and its potential for weak scaling across cores (see Section 4) and (b) a characterization of the processor's power profile for a handful of extreme scenarios (e.g., compute-bound, bandwidth-bound, etc.) via micro-benchmarking of a processor, as described in Section 5.

## 3. ADAPTIVE RUNTIME TO OPTIMIZE ENERGY EFFICIENCY

We now describe our lightweight DVFS runtime approach. Its motivation is twofold. First, we want to design a runtime algorithm that is able to find automatically a good frequency to improve *energy efficiency* for any processor in light of the requirement for this frequency to be potentially radically different for different processors, as shown in the previous section. Second, we want an approach to energy savings that does not require analysis of the source code nor profiling of the workload [Hsu and Kremer 2003]; in other words, one that can work on arbitrary programs. This runtime serves as a baseline for our compile-time frequency/core selection method, the core contribution of our article, as described in Section 4. We will show how further energy savings can be achieved for specific computation kernels that can be analyzed with the polyhedral model in Section 6.

*Runtime algorithm*. The main idea of our runtime is to continuously inspect the changes in energy efficiency; that is, the energy consumption normalized by the number of instructions executed and the changes in operational intensity (OI) of the application. A decision to change frequency is made either because the OI has changed, indicating a change in the nature of the computation being performed, or because the energy efficiency has decreased compared to the last sample. Within a single phase (i.e., a time segment with similar OI), we exploit the energy convexity rule: There is only one frequency maximizing energy efficiency [De Vogeleer et al. 2014], as illustrated in the previous section. Consequently, our algorithm to find this frequency simply performs a gradient descent in the space of frequencies. The algorithm is shown on the right.

This algorithm has several parameters that can be tuned: The sampling interval $\Delta$t, the threshold for significant change in energy efficiency $\Delta$Eff, and the threshold for significant change in OI $\Delta$OI. We have implemented this approach using Intel Performance Counter Monitor (PCM) [Intel b] version 2.10, which provides counters to obtain the number of operations executed, the quantity of data transferred to/from RAM, and the average CPU power between two time points. We compute the OI and energy efficiency from these counters. The runtime program is a separate thread implemented via Unix signals, triggering the monitoring and frequency selection every $\Delta$t time. We measured the time overhead of the runtime approach to be well below 1% when using $\Delta t = 50$ms. We chose $\Delta Eff = \Delta OI = 5\%$ in our experiments, and the runtime aggregates energy and OI metrics for all CPU cores using Intel PCM counters. Precisely, the OI is measured respective to the RAM accesses and total number of CPU instructions executed, and the energy efficiency is the energy consumed by the socket

```
Input: Sampling time interval: Δt
       Energy efficiency threshold: ΔEff
       OI threshold: ΔOI
 1: Eff_last = OI_last = 0
 2: lastChange = increaseFreq
 3: for each time quanta Δt do
 4:   change = noChange
 5:   Eff = computeEnergyEfficiency()
 6:   OI = computeOI()
 7:   if Eff < Eff_last + ΔEff then
 8:     if OI < OI_last − ΔOI then
 9:       change = reduceFreq
10:     else if |OI − OI_last| < ΔOI then
11:       change = reverse(lastChange)
12:     else if OI > OI_last + ΔOI then
13:       change = increaseFreq
14:     end if
15:   else if Eff > Eff_last + ΔEff then
16:     change = lastChange
17:   end if
18:   if change != noChange then
19:     changeFrequency(change)
20:     lastChange = change
21:   end if
22: end for
```

in the time elapsed normalized by the number of total CPU instructions executed. Note that the possible frequencies the changeFrequency function can output are part of a pre-determined list of frequencies: We selected five (six for Haswell) frequencies between 1.6GHz and the maximal CPU frequency. Therefore, the algorithm always output a valid frequency.

*Experimental results*. Figure 2 shows the energy savings, compared to `powersave`, of the `on-demand` Linux frequency governor and of our runtime approach. We evaluate on 60 computation kernels as detailed in Section 6.1 to enable comparison with our compile-time approach developed in later sections using OpenMP parallelization and executing on all cores of the target machines. We detail the two most interesting machines for this experiment: an eight-core Ivy Bridge machine, where the lowest energy can be achieved for high frequencies, and a four-core Haswell, where the minimal frequency is not always minimizing energy, as shown in Section 2.

As expected, the `on-demand` governor can significantly improve energy consumption on Ivy Bridge compared to `powersave`, but it can also be highly detrimental, in particular for more bandwidth-bound benchmarks. For Haswell, `on-demand` is typically highly detrimental. In contrast, our proposed runtime approach is only rarely detrimental compared to `powersave` but, on average, significantly boosts energy savings for Ivy Bridge. It addresses the deficiencies of the two Linux governors by looking at energy efficiency instead of simply CPU workload and offers a viable one-size-fits-all algorithm despite the processor-specific characteristics of the energy-minimizing frequency. Note also that because `powersave` runs at the minimal frequency, our runtime approach can only equal or improve the overall execution time.

However, as we show in the next sections, further gains can be attained by addressing two inherent limitations of a runtime-based approach: (i) the "wasted" time to reach

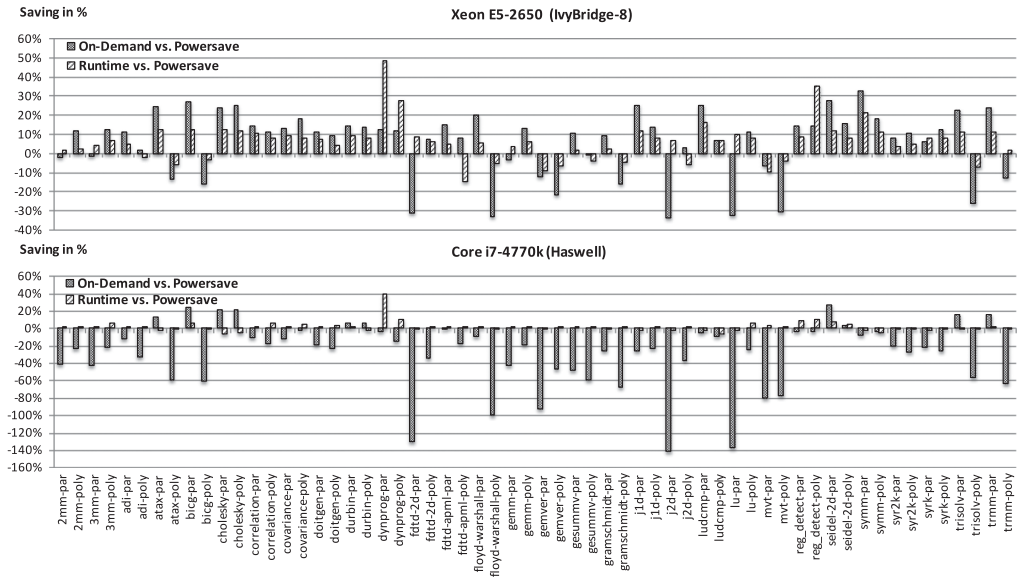


Fig. 2. Comparison of energy savings for `on-demand` Linux governor and our proposed runtime versus `powersave`.

the optimal frequency for a regular computation chunk (i.e., a phase in the program) and (ii) the inability to adapt the number of CPU cores allocated to the chunk based on parallel scaling. We show how these limitations can be efficiently addressed for affine kernels.

## 4. STATIC ANALYSES

A fundamental property of affine computations is to have only static control-flow and data-flow; that is, the code executed does not depend on the dataset value but only on the value of loop iterators and program constants. This regularity enables the design of key *static* analyses for these program regions, for instance, to characterize their operational intensity. We aim to substitute our runtime approach based on runtime OI inspection by a compile-time characterization of the program region using novel analyses we now develop.

### 4.1. Approximating Operational Intensity

The OI of a program, typically in FLOP per byte for scientific codes, is the ratio of operations executed per data moved to execute these operations. The OI may be computed during runtime using a data movement count from the transfers from RAM to the last-level cache or another level of cache. In this work, we are interested in categorizing a program region (e.g., a loop nest) as either memory-bound or compute-bound: Only a coarse approximation of the OI is needed. We also have the goal of developing a very fast analysis that can be applied on large source codes. Indeed, the result of applying polyhedral transformations to a program may lead to a code of thousands of lines from an input loop nest of a few lines, as is the case for numerous benchmarks we evaluate in Section 6.

We propose a *fast* static analysis that can categorize a program region at compile-time in less than 1 second for affine programs made of possibly thousands of lines of code. Prior work on cache miss modeling for affine programs, such as the Cache Miss

```
    for (i = 1; i < height - 1; ++i)
      for (j = 1; j < width - 1; ++j)
R:      Out[i][j] = 4*In[i][j]-In[i-1][j]
          -In[i+1][j]-In[i][j-1]-In[i][j+1];
```

Fig. 3. Jacobi 2D 5 point sweep.

Equations [Ghosh et al. 1999] or the work of Chatterjee et al. [2001], can provide exact counts of cache misses but at the expense of a prohibitively costly static analysis which may take hours to complete even for simple codes [Chatterjee et al. 2001]. These are not suitable for our objectives. In this work, we trade off accuracy for analysis speed, as an approximation of the OI is sufficient for our need of categorizing a program region. Our analysis takes an arbitrary C code as input, automatically extracts affine regions, and, for each, approximates the OI. It does not perform any transformation, and it works in a stand-alone fashion.

*Background on polyhedral program representation*. The polyhedral model is a flexible and expressive representation for imperfectly nested loops with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoPs) [Feautrier 1992; Girbal et al. 2006], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the enclosing loop iterators and variables that are invariant during the SCoP execution. Numerous scientific kernels exhibit those properties; they can be found in image processing filters, linear algebra computations, and the like [Girbal et al. 2006]. We now describe the key data structures and objects needed to represent and manipulate programs in this work. We illustrate the main ideas using the simple example in Figure 3, a single-sweep of a 2D stencil.

*Iteration domains*. For each textual statement in the program, the set of its runtime instances is captured with an integer set bounded by affine inequalities intersected with an affine integer lattice [Bastoul 2004] (i.e., the iteration domain of the statement). Each point in this set represents a unique dynamic instance of the statement, such that the coordinates of the point correspond to the value the surrounding loop iterators take when this instance is executed. For instance, for statement `R` in Figure 3, its iteration domain $\mathcal{D}_R$ is:

$$\mathcal{D}_R = \{(i, j) \in \mathbb{Z}^2 \mid 1 \leq i < \text{height} - 1 \wedge 1 \leq j < \text{width} - 1\}.$$

We denote $\vec{x}_R \in \mathcal{D}_R$ as a point in the iteration domain.

*Access functions*. These functions capture the location of the data accessed by the statement. In static control parts, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts that are affine expressions of surrounding loop counters and global parameters. For instance, the subscript function of the fifth read reference to `In` in R, `In[i][j+1]`, surrounded by two loops $i$ and $j$ is $F^{In}_{5,R}(i, j) = (i, j + 1)$ and its values when evaluated for all pairs $(i, j) \in \mathcal{D}_R$ capture the set of addresses of A accessed by this reference.

*Data space*. We first define the data space of an array $A$ for a program (i.e., the set of data accessed by all references to a specific array) during the entire program execution. The data space is simply the union of the sets of data elements accessed through the various access functions referencing this array for each value of the surrounding loop iterators where the reference is executed. The polyhedral program representation enables the use of the image of a polyhedron (e.g., the iteration domain) by an affine

function (e.g., the access function) to capture data spaces. The image of a polyhedron $\mathcal{D}$ by an affine function $F$ is defined as the set $\{\vec{y} \mid \forall \vec{x} \in \mathcal{D},\ F(\vec{x}) = \vec{y}\}$.

*Definition* 4.1 (*Data Space*). Given an array $A$, a collection of statements $\mathcal{S}$, and the associated set of memory references $F^A_{i,S}$ with $S \in \mathcal{S}$, the data space of $A$ is the set of unique data elements accessed during the execution of the statements. It is the union of the image of the iteration domains by the $i$ access functions in each statement:

$$DS^A = \bigcup_{S \in \mathcal{S}} Image\big(F^A_{i,S}, \mathcal{D}_S\big).$$

We remark that $DS^A$ is not necessarily a convex set but can still be manipulated with existing polyhedral libraries. For example, in Figure 3, $DS^{In} = \{(x, y) \mid 0 \leq x < height, 0 \leq y < width\}$ and $DS^{Out} = \{(x, y) \mid, 1 \leq x < height - 1, 1 \leq y < width - 1\}$.

*Data space of a loop iteration*. It is very convenient to be able to restrict the data space to a particular loop iteration, for instance, to compute the data space for one execution of the $j$ loop (i.e., one iteration of the $i$ loop). We will use such a mechanism to approximate cache misses later. A simple approach to achieve this is to compute a "slice" of interest of the iteration domain and use this sliced domain in the data space computation. A parametric slice [Pouchet et al. 2013] along a set of dimensions (i.e., loops) is defined as follows: Given a loop nest with a loop $l$ of depth $n$ surrounded by $k - 1$ loops, the parametric slice PS of loop $l$ is a subset of $\mathbb{Z}^n$ defined as:

$$PS_{l,\alpha} = \{(x_1, \ldots, x_n) \in \mathbb{Z}^n | x_1 = p_1, \ldots, x_{k-1} = p_{k-1}, x_k = p_k + \alpha\},$$

where $p_1, \ldots, p_n$ are parametric constants unrestricted on $\mathbb{Z}$, and $\alpha$ is an integer value.

For example, a slice of the first loop in Line 1 of Figure 3 for statement $S$ is: $PS_{l,0} = \{(i, j) \in \mathbb{Z}^2 | i = p_1\}$ This is a (parametric) set of 2D integer points with the first component of each point always having the same (unknown yet constant) value. When intersecting a slice with an iteration domain, one fixes certain loops to a unique "generic" value which is, by construction, in the set of possible values the loop iterators can take when the program execute.

We can now adapt the definition of a data space to the subset of data accessed by a loop iteration.

*Definition* 4.2 (*Data Space of a Loop Iteration*). Given an array $A$, a collection of statements $\mathcal{S}$ surrounded by a loop $l$ and their associated set of memory references $F^A_{i,S}$ with $S \in \mathcal{S}$, and $PS_{l,0}$ a PS for loop $l$, the data space of $A$ is the set of unique data elements accessed during one iteration of $l$:

$$DS^A_{PS_{l,0}} = \bigcup_{S \in \mathcal{S}} Image\big(F^A_{i,S}, \big(\mathcal{D}_S \cap PS_{l,0}\big)\big).$$

The data spaces of loop iterations we manipulate are a union of $\mathcal{Z} - polyhedra$ and operations such as intersection ($\cap$), union ($\cup$), and difference ($\setminus$); critically computing a counting function # (e.g., $\#DS^{In} = \text{height} * \text{width}$) can be done at compile-time using specialized libraries such as the Integer Set Library [Verdoolaege 2010] and Barvinok [Verdoolaege et al. 2007].

*4.1.1. Approximating Data Movement.* The Distinct Lines (DL) model was designed originally to estimate the number of distinct cache lines, or TLB entries, accessed in a loop nest [Ferrante et al. 1991; Sarkar 1997]. It essentially represents the footprint of a computation in terms of cache lines accessed, thereby taking into account spatial locality. DL formulas are typically used to analytically find values for the loop bounds (e.g.,

tile sizes) to ensure the number of distinct lines accessed is below the cache capacity, therefore ensuring that data reuse is implemented in cache.

When the DL of a loop nest is lower than the cache size, it is then a good approximation of the number of cache misses: Only cold misses will occur, one per line of cache accessed, because the reuse will be fully implemented without any data being evicted. Conflict misses are ignored here, and, in the following, we will assume they do not dominate the miss behavior (e.g., arrays have been properly padded [Hong et al. 2016]). On the other hand, when the DL is larger than the cache size, DL does not allow us to determine an estimate of the number of misses: It depends on the schedule of operations and, in particular, on the reuse distance between references to the same array.

In this work, we propose to approximate the data movement between two levels of memory by approximating the number of cache misses at a certain level. We achieve this by (i) formulating DL in the polyhedral framework using the concepts presented earlier; (ii) extending it to also capture the data reuse between consecutive loop iterations at any loop level; and (iii) designing an algorithm that approximates the number of misses when the DL exceeds the cache size.

*Computing DL using polyhedra*. The preceding data space definitions are the essential bricks to compute a polyhedral expression of the number of distinct lines: It already provides the set of distinct memory locations accessed by a (slice of the) program. The only missing part is to translate this into distinct cache lines. This is simply done by first representing the mapping from memory address to cache line of size $ls$ (line size) using an affine function of the same dimensionality as the array. For instance for a two-dimensional array and $ls = 8$ (e.g., `A[N][M]`), we create the function linemap$(i, j) = (i, j/8)$. Then, the set of distinct lines for an array $A$ is simply the image of $DL^A$ by the linemap function. For example, $DL^{In}$ using $ls = 8$ is: $DL^{In} = \{(a, b) \mid 0 \leq x < height, 0 \leq y < width, a = x, 8 * b \leq y < 8 * b + 8\}$.

In a manner analogous to data spaces, one can compute the DL of a particular loop iteration by using parametric slices of the domain. Such a case will be denoted $DL^A_{PS_{l,\alpha}}$ for array $A$ and loop $l$ with offset $\alpha$, which represents DL for one iteration of loop $l$. The DL of a full loop $l$ is noted as $DL^A_l$ and is computed by using a parametric slice of the loops surrounding $l$, considering only the array references in $l$'s body.

*Algorithm for miss estimation*. We are now equipped to build our procedure to estimate the misses. The idea is the following: We will recursively compute the DL of a loop (summing it for all arrays accessed by that loop) from the inner-most loops to the outer-most loops. For each loop $l$, its number of misses is estimated as either the product of its trip count and the number of misses of its loop body if the DL of this loop exceeds the cache size or as its DL if it is smaller than the cache size. For inner-most loops bodies, we set their number of misses to the DL of the loop body, regardless of its value. Some additional treatment is done to capture the data reuse between consecutive iterations of a loop body to adjust the number of misses. We optimistically assume that if data are reused between two iterations, then they correspond to the part of the data that was loaded last at the previous iteration (e.g., if this set is smaller than the cache size, it is not evicted from the cache by other data of the previous iteration). See Algorithm 1. For simplicity, we assume that the entire program is surrounded by a fake loop *root* having a single iteration, and we note that DL[] is a map, where DL[x] associates object x to an integer value.

Algorithm 1 uses the TripCount(l) function to compute the trip count of a loop. For affine programs, this is always computable thanks to the static control-flow property. This function is implemented by first forming the union of the iteration domains of all statements surrounded by the loop, then projecting this set onto the dimension corresponding to the loop of interest.

**ALGORITHM 1:** EstimateCacheMisses

**Input:** Number of array elements per cache line: $ls$
cache size, in lines: $C$
Polyhedral program: $P$
**Output:** Estimate of misses

```
1: for all loops l do
2:   Misses[l] = DLloop[l] = DLreuse[l] = Processed[l] = 0
3: end for
4: for all Arrays A do
5:   for all loops l do
6:     DLloop[l] += #DL_l^A
7:     DLreuse[l] += #(DL^A_{PS_{l,0}} ∩ DL^A_{PS_{l,1}})
8:   end for
9: end for
10: for all loops l in postfix AST order and Processed[l] == 0 do
11:   if DLloop[l] < C then
12:     Misses[l] = DLloop[l]
13:   else
14:     if l is inner-most loop then
15:       Misses[l] = DLloop[l]
16:     else
17:       Miss = 0
18:       for all loops ll immediately surrounded by l do
19:         Miss += Misses[ll]
20:         Processed[ll] = 1
21:       end for
22:       Misses[l] = Miss * TripCount(l)
23:       if DLreuse[l] < C then
24:         Misses[l] −= DLreuse[l] * TripCount(l)
25:       end if
26:     end if
27:   end if
28:   Processed[l] = 1
29: end for
30: return Misses[root];
```

A key remark about our algorithm is that it performs comparisons of expressions representing the number of points in polyhedra. While techniques exist to create such expressions as a function of the parameters of the program, this poses challenges when attempting to compare two different parameters: Given height and width without further information, is height > width? Or height < width? In general, this problem is not decidable. To avoid this issue, we require the algorithm to run on an instance of the polyhedral program where the parameter values are known numerical constants (e.g., 1024). In practice, we extract the polyhedral representation and compute data spaces using parametric representations but add context information about the exact parameter values before computing the size of the polyhedra (the # operation).

*4.1.2. Approximating FLOPs and Getting OI.* The last element needed to approximate the OI is the number of operations executed in the program so that we can divide it by the number of memory movements (obtained from the number of misses). This is straightforward in the polyhedral representation: We inspect the AST of each statement

body, collect the number of operations not part of an array subscript expression, and multiply it by the number of points in the iteration domain of the statement. Because this process requires explicit values for the parameters, we leverage the known values to obtain a numerical value for the FLOP count. The overall process to compute OI is outlined in Algorithm 2.

An interesting aspect of this compile-time approach is that it can be applied incrementally on each loop (nest) of the program to detect loop nests with significantly different behaviors. That is, this algorithm can be used to detect compute-bound and memory-bound application phases, thus allowing for individually fine-tuning the DVFS decision for each phase.

### 4.2. Parallelism Features

The parallelism features we extract are simple in comparison of the OI approximation, yet critical in terms of the quality of the approach. We have focused our implementation to cover the cases of automatic parallelization implemented by the PoCC polyhedral compiler [poc]; that is, if a code is parallel, then it has at least one OpenMP `for` pragma in the code and no other type of OpenMP pragmas. This is reminiscent of loop parallelizing compilers.

The first feature we extract is whether the program is sequential or parallel. This is simply done by checking if the program has any OpenMP pragma in it; if not, then it is sequential. The second feature attempts to capture poor scalability of the code (i.e., if the performance improvement does not scale with the number of cores). For this feature, we again rely on the assumption that there are only OpenMP for pragmas to model the parallelization. We analyze the AST to form an estimate of the regularity of the parallel loop trip count, studying the parallel workload properties as detailed in Algorithm 3.

**ALGORITHM 2:** Estimate Operational Intensity

**Input:** Number of array elements per cache line: $ls$
cache size, in lines: $C$
Polyhedral program: $P$
Parameter values: $\vec{n}$
**Output:** Estimate of OI
1: P' = attachContextInformation($\vec{n}$, P)
2: Misses = EstimateCacheMisses(ls, C, P')
3: Flops = countFlops(P')
4: **return** Flops/(Misses * ls)

Function stripMine performs a strip-mining of the loop so that the resulting outer loop has as many iterations as the maximal number of cores $c$ available on the target machine, enabling us to analyze its inner loop $l'.inner$ that is the outer-most serial loop in a thread. We then perform parametric slicing to reason on the different trip counts this loop can have (e.g., for load-imbalanced cases, this loop may iterate 0 times). Function count HasZeroIteration counts the number of values of $l'$ for which the trip count of $l'.inner$ can be equal to 0; if it can be for more than half of the available cores, then this loop is considered to have poor scaling.

Interestingly, this feature has only limited use in our tested benchmarks because the OI feature is actually a better discriminant for codes that do not scale due to bandwidth saturation, which is the common case in our test suite. Here, the poor scalability feature is limited to capturing compute-bound codes with high load imbalance, as

**ALGORITHM 3:** Check Poor Parallel Scaling

**Input:** Polyhedral program: $P$
Max. number of cores: $c$
**Output:** Compute parallel scaling

```
poorScaleWork = goodScaleWork = 0
for all parallel loops l in P do
  l' = stripMine(l, c)
  pds = parametricSlice(l'.inner)
  if countHasZeroIteration(pds,c) ≥ c/2 then
    poorScaleWork += countFlops(l')
  else
    goodScaleWork += countFlops(l')
  end if
end for
return poorScaleWork > goodScaleWork
```

arises typically in pipeline-parallel schedules with large startup and draining phases compared to the steady state.

## 5. PROCESSOR CHARACTERIZATION

### 5.1. One-Time Machine Profiling

As shown in Section 2, the frequency/number of cores configuration needs to be adapted as a function of the nature of the code executing. To find the best frequency/core configuration for a particular machine, we perform a profiling of four benchmarks on the target machine, running them on all frequency steps and core setups available and collecting the execution time and power using hardware counters. This enables us to build a profile of the energy and energy-delay product curves for each benchmark, which are the two metrics we optimize for. These benchmarks have been specifically built to exacerbate one of the features that is computed by our static analysis.

*Compute-bound code*. For this case, we use off-the-shelf DGEMM/MKL, in the same setup as shown in Section 2. We observed the near-perfect scaling of this code/problem size across the architectures tested, and, although being totally compute-bound by nature, it is an actual workload with heavy data traffic.

*Bandwidth-bound code*. Bandwidth-bound programs are frequent, especially if no data locality optimization has been performed on the original benchmark. For this case, we created a benchmark that implements both spatial and temporal locality via tiling with a low arithmetic intensity. It is inspired by an iterative stencil (e.g., Jacobi-2D). This benchmark is bandwidth-bound but with a relevant balance between computations and communications.

*Sequential code*. The sequential code we use as representative of sequential workloads is a SIMD-vectorizable benchmark, with balanced arithmetic intensity (i.e., there is non-negligible memory traffic). It was built from key code features of the durbin benchmark from PolyBench/C.

*Code with poor scalability*. The benchmark used here essentially consists of an OpenMP parallelization of a triangular loop with low trip count to ensure high load imbalance between threads. The computation performed in the loop body has balanced arithmetic intensity and is inspired from LU factorization.

### 5.2. Decision Tree for Frequency/Core Pairs

Finally, we conclude by displaying our process to assign at compile-time a frequency to an affine program region. A program is classified using its OI and its parallelism scaling features to determine to which of the four preceding categories it is closest to. In particular, we found that ordering the decision by prioritizing the cases where the dominant effect has the highest impact delivers the best results.

The decision process is identical whether we optimize for E or EDP. Only the frequency/core configuration selected changes between E and EDP using the best configuration found by profiling for each metric independently. The decision process is as follows:

1) If the code is sequential, choose config_sequential;
2) else, if the code is bandwidth-bound (i.e., its OI is below threshold, found during profiling), choose config_bwbound
3) else, if the code has poor scaling, choose config_poorscale
4) else, choose config_computebound.

## 6. EXPERIMENTAL RESULTS

### 6.1. Experimental Protocol

*Benchmarks considered*. We use the PolyBench/C 3.2 benchmark suite [pol], a popular suite of 30 numerical computations written using affine loops and static control flow. It spans computations in numerous domains such as image processing, linear algebra, and more. For each benchmark, we considered two versions of the code: par was generated by applying a simple auto-parallelization without any other code transformation. It amounts to inserting OpenMP parallel for pragmas around the outer-most parallel loops in the code and vectorization pragmas around the inner-most parallel loops. This can very substantially improve the performance of the original codes, which are sequential by default, and substitute for the auto-parallelization schemes of the C back-end compilers used. We used the PoCC source-to-source polyhedral compiler [poc] to generate the optimized C files. The second variant, poly, is the result of a complex automatic program transformation stage aimed at improving data locality via tiling, and it exposes coarse-grain (multicore) and fine-grain (SIMD) parallelism. It uses the Pluto algorithm [Bondhugula et al. 2008] combined with several other optimizers of PoCC implementing a model-driven prevectorization, unrolling, and parallelization. For these variants, the OI is typically significantly improved after transformation: Temporal data locality is improved via loop tiling whenever possible. These two versions form a total of 60 benchmarks that we evaluated on.

The static analysis was also implemented in the PoCC compiler in a fully automated way, as a new PolyFeat module in PoCC, to compute the features of the input source code to enable its categorization, as presented in the previous sections. Each of the 60 C source code was separately analyzed, with the analysis taking at most a second to complete.

Each benchmark was compiled using GNU compiler GCC-4.8.1 with `-O3 -fopenmp` optimization for the Intel CPUs, along with `-m64 -mcpu=power8 -mtune=power8` for the IBM POWER8.

*Collecting energy and time*. To assess the quality of our framework, we needed to compute the optimal frequency/core configuration for each binary individually by collecting its energy and execution time for each frequency/core configuration evaluated on the target machine. These data were collected using Intel PCM [Intel b] for the Intel chips, and IBM AMESTER [Floyd et al. 2010] for the POWER8. The instrumentation was inserted around each kernel of interest. To obtain stable and

sound energy measurement value, Turbo-boost was turned off on the Intel chips. The following process was repeated as needed to achieve an execution time of around 1 minute, and the average energy and time were obtained: (1) flush data caches, (2) start instruments, (3) execute kernels, (4) stop and collect energy and execution time

On each target machine, we used the different frequencies made available by the Linux OS running on these machines. Table I lists the frequency ranges. We used 5–6 frequency steps per machine, spaced evenly in the frequency range. We tested three core counts: one, half, and all available. So, a total of 78 configurations (15–18 per machine) was tested for each of the 60 binaries. We empirically found the best configuration (frequency/core) to optimize energy or energy-delay product for each benchmark when using a fixed configuration for the entire kernel execution. In the following, these are designated as the Best configuration for a given benchmark and optimization metric.

### 6.2. Summary of Results

Figure 4 summarizes our performance results. The five bar charts provide the energy savings, on a per-benchmark basis, of our approach (Ours) compared to using the `powersave` Linux governor and the savings that can be achieved by using the Best frequency/core configuration found empirically by testing all of them. Figure 5 summarizes the average energy savings E and energy-delay product EDP improvement across all 60 benchmarks, compared to `powersave` of the `on-demand` Linux governor, our implementation of the CPUMiser[2] runtime approach [Ge et al. 2007], Ours (static), and Best (static) empirically found.

*Energy savings over powersave*. The `powersave` Linux governor uses all available cores and sets the frequency to the minimal one. This principle uses the idea that a frequency increase has a cubic effect on power increase but a linear effect in execution time decrease, so using the minimal frequency should minimize CPU energy. But, as we demonstrated in Section 2, this may not lead to minimizing energy. Furthermore, this approach essentially ignores the remaining system power consumption. *In this work, we show how CPU energy savings can be achieved by increasing the frequency*, which in turn typically leads to also reducing the kernel execution time compared to `powersave`. As a consequence, the system energy consumption is further reduced with our approach compared to `powersave`; we, however, do not report it and focus solely on CPU energy reports.

For SandyBridge, we observe that minimizing frequency is a viable way to minimize energy, and for numerous benchmarks our approach selects the minimal frequency (therefore showing 0% savings over `powersave`), which was also found to be the best scenario by empirical evaluation of all frequency/core pairs (the Best bars). Still, a significant improvement is achieved using our approach for benchmarks which are either sequential or have poor scalability, such as dynprog-par, where we select a reduced number of cores.

The benefits of our approach over `powersave` are highlighted with architectures where the energy-minimizing frequency is not the minimal one, such as on Ivy Bridge and Haswell. An average savings of 13% is achieved for the eight-core Ivy Bridge, coming from selecting higher frequencies to balance completion time and power consumed following the processor-specific characteristics. For almost all kernels, savings above 10% are achieved with our static approach. We also show savings comparable to the best possible ones for the vast majority of cases, thus demonstrating the viability of our compile-time categorization of codes. For cases with lower savings than Best, the frequency/core selected was typically one step above the best one (e.g., we used 1.6GHz

[2]Our IBM POWER8 setup does not allow for runtime DVFS, so CPUMiser results are omitted.

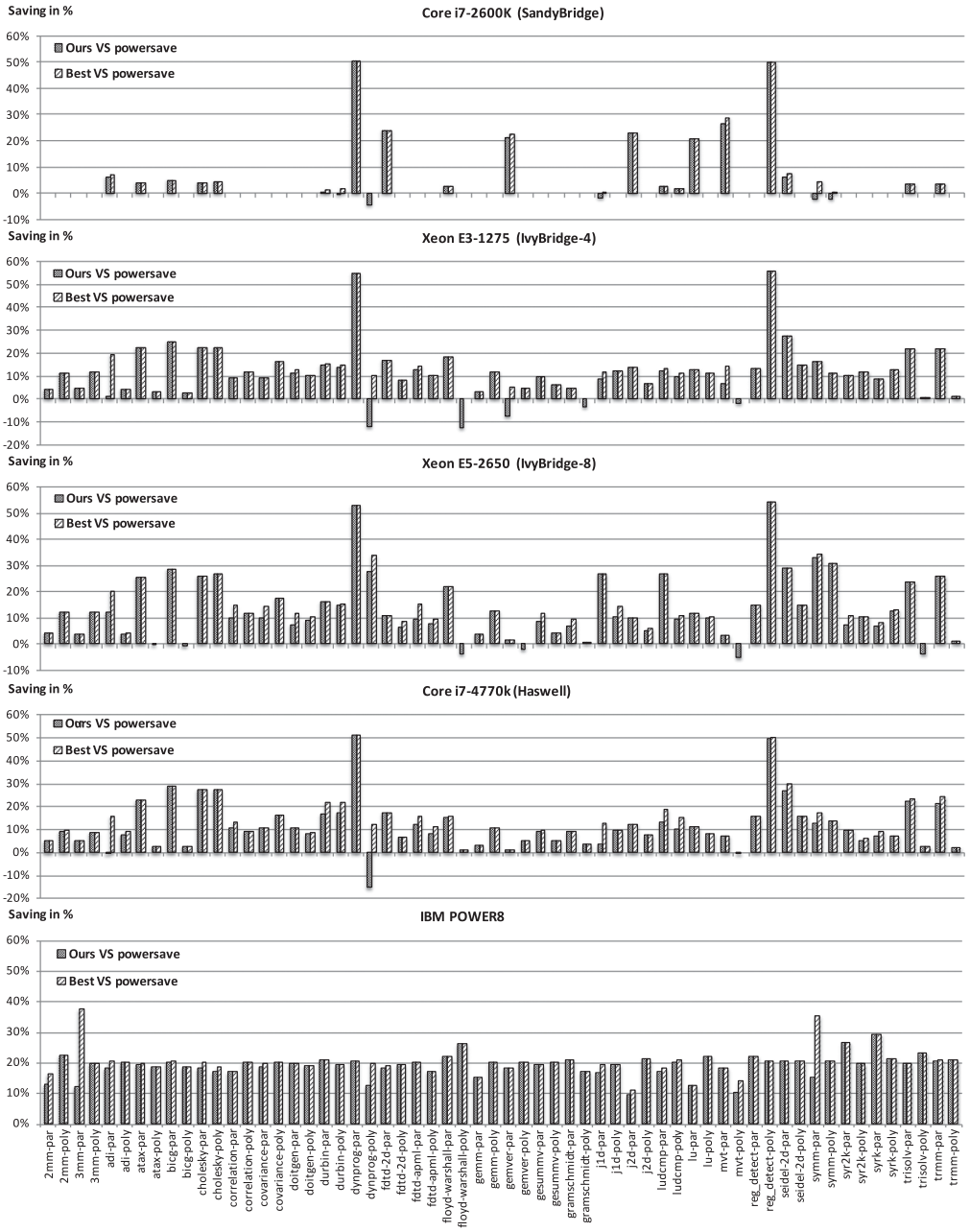


Fig. 4. CPU Energy gains with static approaches versus `powersave`.

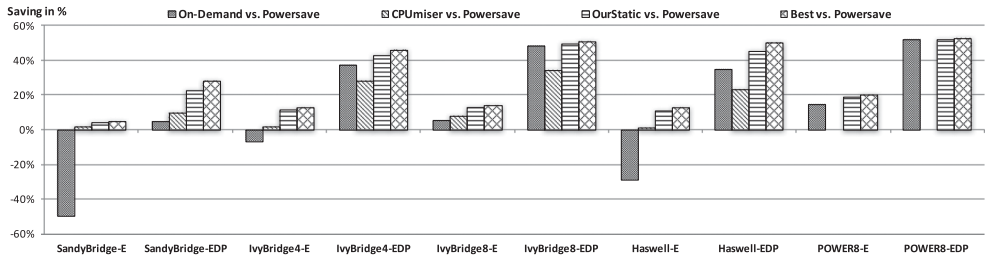


Fig. 5. Summary of experimental results: Energy (E) and energy-delay-product (EDP) gains versus `powersave` of two dynamic approaches (`on-demand` and CPUMiser) and two static approaches (Ours and Best).

instead of the ideal 1.2GHz) and suggests that further improvements may be achieved using more categories for the applications.

*Energy and EDP improvements*. Figure 5 summarizes the results for all architectures. For Ivy Bridge and P8, the energy difference of `on-demand` over `powersave` is characteristic of these machines, where actually maximal frequency leads to minimizing energy for compute-bound codes. This is exacerbated by the fact that the poly version of the benchmarks has been transformed for improved data locality and, for many, has become essentially compute-bound. Our static analysis seamlessly captured these changes, with, for instance, gemm and seidel-2d moving from being categorized as memory-bound in their par version to compute-bound in their poly version as the result of transformations for locality, tiling, and SIMD vectorization.

CPUMiser by Ge et al. [2007] is a runtime approach to adapt frequency/voltage based on the measured cycle per instructions (CPI) achieved by the workload. We have reimplemented it using exactly the same runtime harness as in Section 3, also using Intel PCM to capture CPU behavior. CPUMiser is geared to finding a frequency that does not reduce the execution time beyond a threshold. In Figure 5, we configured CPUMiser to use a 100ms sampling interval and 10% maximum performance loss. More comprehensive results are shown in Section 6.3. CPUMiser is able to work with arbitrary binaries/programs, contrary to our static approach, which is restricted to affine program regions. But CPUMiser is unable to adapt the number of cores to use, thus leading to increased energy usage compared to our technique for codes with poor scaling. In addition, CPUMiser does not consider energy efficiency as the optimization objective; it only attempts to reduce frequency while keeping CPI increase under a certain threshold. As we showed in Section 2, the frequency that minimizes CPU energy may be below the maximal frequency even for compute-bound codes which would see a significant increase in CPI by using a lower-than-maximal frequency. This is one of the reasons that our approach outperforms CPUMiser, and this is particularly visible for Haswell.

Finally, when optimizing for EDP, we observe consistent conclusions between the different schemes as when optimizing for energy, with our approach consistently outperforming competing schemes and being close to the best achievable in our framework.

### 6.3. Comparison with Runtime DVFS

Figure 6 presents a comparative study of the energy savings of various DVFS schemes. We report the results of CPUMiser with different performance degradation parameters: 10%, 50%, and 100% respectively. We also tuned the Linux `on-demand` (Od) and `conservative` (Co) governors. They are both dynamic schemes based on CPU usage and can be parameterized for different CPU loads. The `conservative` governor adapts the frequency gracefully, while `on-demand` goes to maximal frequency when the load

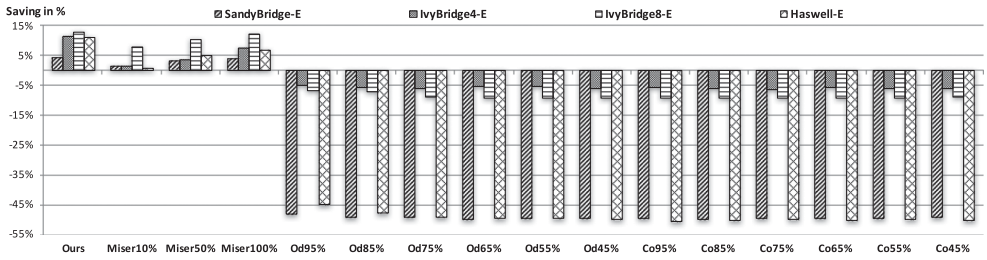


Fig. 6. Energy savings versus runtime DVFS schemes.

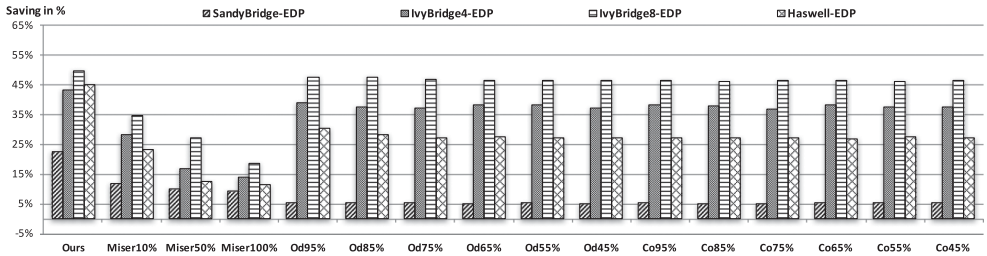


Fig. 7. EDP savings versus runtime DVFS schemes.

threshold is reached. We vary the CPU load threshold from 45% to 95% by a step of 10% for both governors. We also report Ours, the performance of our static approach as presented earlier, for comparison. Figure 7 summarizes the energy-delay product savings over the `powersave` governor for the same DVFS schemes.

Given the nature of the workloads evaluated, both `conservative` and `on-demand` use the maximal CPU frequency most of the time. Consequently, the CPU energy compared to `powersave` increases on average because, for most benchmarks, maximal frequency is not the way to minimize CPU energy. This is especially true for SandyBridge and Haswell. In contrast, for Ivy Bridge, where running at maximal frequency is often good for energy optimization, the energy loss of these governors compared to `powersave`is very small. This analysis holds true for EDP results in Figure 7: Significant EDP savings over powersave are achieved by these governors, especially for Ivy Bridge. Note in all cases that our static approach Ours outperforms these governors, whether it was set to optimize CPU energy or to optimize energy-delay product.

CPUMiser results show that by allowing for large performance degradation (e.g., 2×; i.e., 100%), it is possible to reach good levels of CPU energy savings, as shown in Figure 6, although in turn it leads to limited EDP savings, as shown in Figure 7. Our approach significantly outperforms CPUMiser in terms of EDP savings. In fact, we also evaluated CPUMiser with 1% as the performance threshold and observed that our static approach achieves EDP savings 2× or more higher than CPUMiser for SandyBridge and Haswell.

### 6.4. Summary of Intel ICC and GNU GCC experiments

We conducted an extensive characterization of the energy and energy-delay product of the same benchmarks as in the previous section, for a total of 60 benchmarks. Each was compiled using GNU GCC-4.8.1 with -O3 optimization and also Intel ICC-2013-update5 compiler with -fast optimization, for a total of 120 different binaries evaluated.

*Summary of results*. Table II reports the summary of experiments when optimizing for CPU energy minimization. For each 60 benchmarks, the average energy savings (E)

Table II. Energy Efficiency Summary

| Microarchitecture | Compiler | E save Best vs. powersave | E save Perf. vs. powersave | E save Ours vs. powersave |
|---|---|---|---|---|
| Haswell | GCC | 12.95% | −28.84% | **11.27%** |
| | ICC | 11.84% | −35.33% | **10.43%** |
| IvyBridge-4 | GCC | 12.83% | −6.48% | **11.35%** |
| | ICC | 11.69% | −9.11% | **10.71%** |
| IvyBridge-8 | GCC | 14.38% | 5.78% | **13.11%** |
| | ICC | 13.39% | 5.30% | **12.15%** |
| SandyBridge | GCC | 4.59% | −49.71% | **4.17%** |
| | ICC | 3.75% | −51.11% | **3.06%** |

Table III. Energy-Delay Product Efficiency Summary

| Microarchitecture | Compiler | EDP save Best vs. powersave | EDP save Perf. vs. powersave | EDP save Ours vs. powersave | Time loss Ours vs. |
|---|---|---|---|---|---|
| Haswell | GCC | 50.52% | 35.24% | **45.16%** | **11.64%** |
| | ICC | 47.57% | 31.32% | **43.26%** | **11.08%** |
| IvyBridge-4 | GCC | 46.10% | 37.14% | **43.14%** | **6.87%** |
| | ICC | 44.70% | 35.46% | **42.00%** | **3.94%** |
| IvyBridge-8 | GCC | 51.17% | 48.44% | **49.50%** | **0.17%** |
| | ICC | 51.22% | 48.49% | **50.03%** | **1.92%** |
| SandyBridge | GCC | 28.41% | 4.87% | **22.61%** | **8.05%** |
| | ICC | 25.46% | 5.67% | **20.77%** | **9.04%** |

compared to `powersave` is reported for three approaches. Best is the best frequency/core configuration found by exhaustive search of all frequency/core configurations individually for each binary tested and each CPU tested. Perf. is the `performance` Linux governor, and Ours is our static approach.

In Table III, we compare the same three approaches. Note that the Best approach here is the result of an empirical search for the best frequency/core configuration that maximizes EDP. We additionally report the average wall-clock execution time increase using our predicted configuration versus using performance.

We observe that, on average, our approach vastly outperforms performance and powersave in terms of both energy savings and EDP improvements: We are, in fact, very close to the optimal situation for energy, with our approach being within 102% or less (i.e., less than 2% worse) of the truly minimal energy that can be achieved using the best configuration found individually for each binary tested and each processor. This result is even further exacerbated for EDP, where our EDP savings can be up to 50% better than using the performance governor. Our predictor approach achieves always within less than 10% of the optimal EDP. The penalty in execution times is also very good: The codes are slowed on average compared to their maximal speed by at most 11.6%; for the Ivy Bridge machines, where performance is the best configuration for energy efficiency for compute-bound benchmarks, our execution time increase is significantly smaller than our EDP improvement over performance. In fact, such execution time degradation can be so small that, even if the system power is large compared to the CPU power, our DVFS approach still leads to significant overall energy savings.

*Detailed results on Haswell*. Table IV details similar results on a per-benchmark basis focusing exclusively on the Haswell architecture and the GCC compiler. We selected this architecture because it is the most recent one in our testbed. On Haswell, as shown in Section 2, the optimal frequency for energy even for compute-bound codes is not the maximal (nor minimal) frequency.

Table IV. Results for Haswell/GCC: F1.6/4 Is a CPU Frequency of 1.6GHz Using Four Cores

| Benchmarks | Ver. | Freq/core: Best for energy | Freq/core: OurStatic for energy | %E savings OurStatic vs. powersave | %E savings Best vs. powersave | Freq/core: Best for EDP | Freq/core: OurStatic for EDP | %EDP savings OurStatic vs. powersave | %EDP saving Best vs. powersave | %Time increase OurStatic vs. performance |
|---|---|---|---|---|---|---|---|---|---|---|
| 2mm | par | F1.6/4 | F1.6/4 | 4.9 | 4.9 | F3.1/4 | F2.5/4 | 41.6 | 43.7 | 28 |
| | poly | F1.6/4 | F2.0/4 | 9.1 | 9.5 | F3.5/4 | F3.5/4 | 57.0 | 57.0 | 0 |
| 3mm | par | F1.6/4 | F1.6/4 | 5.1 | 5.1 | F3.1/4 | F2.5/4 | 41.7 | 43.7 | 28 |
| | poly | F2.0/4 | F2.0/4 | 8.6 | 8.6 | F3.5/4 | F3.5/4 | 57.4 | 57.4 | 0 |
| adi | par | F2.0/2 | F1.2/2 | −0.5 | 15.6 | F3.1/4 | F2.0/4 | 47.2 | 56.7 | 57 |
| | poly | F2.0/4 | F1.6/4 | 7.5 | 9.1 | F2.5/4 | F2.5/4 | 43.8 | 43.8 | 18 |
| atax | par | F2.5/1 | F2.5/1 | 22.9 | 22.9 | F3.5/1 | F3.5/1 | 71.1 | 71.1 | 0 |
| | poly | F1.6/4 | F1.6/4 | 2.5 | 2.5 | F2.0/4 | F2.5/4 | 27.1 | 31.0 | 12 |
| bicg | par | F2.5/1 | F2.5/1 | 28.7 | 28.7 | F3.5/1 | F3.5/1 | 74.2 | 74.2 | 0 |
| | poly | F1.6/4 | F1.6/4 | 2.3 | 2.3 | F2.0/4 | F2.5/4 | 26.2 | 29.2 | 11 |
| cholesky | par | F2.5/1 | F2.5/1 | 27.0 | 27.0 | F3.5/1 | F3.5/1 | 72.6 | 72.6 | 0 |
| | poly | F2.5/1 | F2.5/1 | 27.1 | 27.1 | F3.5/1 | F3.5/1 | 73.6 | 73.6 | 0 |
| correlation | par | F2.0/4 | F1.6/4 | 10.6 | 13.3 | F3.1/4 | F2.5/4 | 52.3 | 57.5 | 30 |
| | poly | F2.0/4 | F2.0/4 | 9.1 | 9.1 | F3.5/4 | F3.5/4 | 58.4 | 58.4 | 0 |
| covariance | par | F1.6/4 | F1.6/4 | 10.8 | 10.8 | F3.5/4 | F2.5/4 | 50.5 | 56.1 | 32 |
| | poly | F2.0/4 | F2.0/4 | 16.2 | 16.2 | F3.5/4 | F3.5/4 | 64.3 | 64.3 | 0 |
| doitgen | par | F1.6/4 | F1.6/4 | 10.4 | 10.4 | F3.5/4 | F2.5/4 | 53.0 | 59.0 | 40 |
| | poly | F1.6/4 | F2.0/4 | 7.8 | 8.8 | F3.5/4 | F3.5/4 | 56.2 | 56.2 | 0 |
| durbin | par | F2.0/1 | F2.5/1 | 16.7 | 21.6 | F3.5/2 | F3.5/1 | 62.3 | 62.7 | 1 |
| | poly | F2.0/2 | F2.5/1 | 17.1 | 21.8 | F3.5/1 | F3.5/1 | 62.8 | 62.8 | 1 |
| dynprog | par | F2.5/1 | F2.5/1 | 50.9 | 50.9 | F3.5/1 | F3.5/1 | 82.5 | 82.5 | 0 |
| | poly | F2.0/4 | F1.2/2 | −15.1 | 12.2 | F3.5/4 | F2.0/4 | 46.8 | 58.0 | 65 |
| fdtd-2d | par | F1.2/2 | F1.2/2 | 17.1 | 17.1 | F1.2/2 | F2.0/4 | −24.7 | 12.9 | 0 |
| | poly | F1.6/4 | F1.6/4 | 6.6 | 6.6 | F3.1/4 | F2.5/4 | 45.8 | 47.8 | 25 |
| fdtd-apml | par | F2.0/4 | F1.6/4 | 12.0 | 15.5 | F3.5/4 | F2.5/4 | 56.4 | 65.5 | 40 |
| | poly | F2.0/4 | F1.6/4 | 8.2 | 10.8 | F3.1/4 | F2.5/4 | 52.3 | 58.2 | 35 |
| floyd-warshall | par | F2.0/1 | F2.5/1 | 15.3 | 15.7 | F3.1/1 | F3.5/1 | 50.2 | 55.6 | 0 |
| | poly | F1.6/4 | F1.6/4 | 0.9 | 0.9 | F1.6/4 | F2.5/4 | −13.6 | 13.8 | 2 |
| gemm | par | F1.6/4 | F1.6/4 | 3.1 | 3.1 | F3.1/4 | F2.5/4 | 41.2 | 43.1 | 27 |
| | poly | F2.0/4 | F2.0/4 | 10.3 | 10.3 | F3.5/4 | F3.5/4 | 58.4 | 58.4 | 0 |
| gemver | par | F1.6/4 | F1.6/4 | 1.0 | 1.0 | F1.6/4 | F2.5/4 | −4.2 | 16.2 | 3 |
| | poly | F1.6/4 | F1.6/4 | 5.0 | 5.0 | F2.5/4 | F2.5/4 | 36.1 | 36.1 | 17 |
| gesummv | par | F2.0/4 | F1.6/4 | 9.0 | 9.6 | F2.0/4 | F2.5/4 | 44.5 | 45.1 | 2 |
| | poly | F1.6/4 | F1.6/4 | 4.8 | 4.8 | F2.0/4 | F2.5/4 | 33.9 | 35.3 | 8 |
| gramschmidt | par | F1.6/4 | F1.6/4 | 8.9 | 8.9 | F3.1/4 | F2.5/4 | 47.5 | 50.5 | 30 |
| | poly | F1.6/4 | F1.6/4 | 3.7 | 3.7 | F2.0/4 | F2.5/4 | 17.8 | 25.2 | 8 |
| jacobi-1d | par | F1.6/2 | F2.5/1 | 3.6 | 12.5 | F2.5/1 | F3.5/1 | 28.7 | 41.1 | 0 |
| | poly | F1.6/4 | F1.6/4 | 9.5 | 9.5 | F3.5/4 | F2.5/4 | 52.5 | 58.4 | 40 |
| jacobi-2d | par | F1.2/2 | F1.2/2 | 12.0 | 12.0 | F1.6/2 | F2.0/4 | −29.8 | 8.0 | 0 |
| | poly | F1.6/4 | F1.6/4 | 7.6 | 7.6 | F3.1/4 | F2.5/4 | 42.8 | 43.7 | 24 |
| ludcmp | par | F2.0/1 | F2.5/1 | 13.2 | 18.8 | F3.1/1 | F3.5/1 | 50.3 | 54.4 | 2 |
| | poly | F2.0/1 | F2.5/1 | 10.1 | 15.1 | F3.1/1 | F3.5/1 | 47.0 | 49.4 | 0 |
| lu | par | F1.2/2 | F1.2/2 | 10.9 | 10.9 | F1.6/2 | F2.0/4 | −27.7 | 6.3 | 1 |
| | poly | F2.0/4 | F2.0/4 | 8.0 | 8.0 | F3.5/4 | F3.5/4 | 56.4 | 56.4 | 0 |
| mvt | par | F1.6/4 | F1.6/4 | 7.1 | 7.1 | F2.0/4 | F2.5/4 | 9.2 | 28.2 | 4 |
| | poly | F1.2/4 | F1.6/4 | −0.7 | 0.0 | F2.0/4 | F2.5/4 | 10.6 | 19.2 | 8 |

(Continued)

Table IV. Continued

| Benchmarks | Ver. | Freq/core: Best for energy | Freq/core: OurStatic for energy | %E savings OurStatic vs. powersave | %E savings Best vs. powersave | Freq/core: Best for EDP | Freq/core: OurStatic for EDP | %EDP savings OurStatic vs. powersave | %EDP saving Best vs. powersave | %Time increase OurStatic vs. performance |
|---|---|---|---|---|---|---|---|---|---|---|
| reg_detect | par | F2.0/4 | F2.0/4 | 15.4 | 15.4 | F3.5/4 | F3.5/4 | 64.3 | 64.3 | 0 |
| | poly | F2.0/1 | F2.5/1 | 49.6 | 49.9 | F3.5/1 | F3.5/1 | 82.3 | 82.3 | 0 |
| seidel-2d | par | F3.1/1 | F2.5/1 | 26.5 | 29.9 | F3.5/1 | F3.5/1 | 75.6 | 75.6 | 0 |
| | poly | F2.0/4 | F2.0/4 | 15.5 | 15.5 | F3.5/4 | F3.5/4 | 66.6 | 66.6 | 0 |
| symm | par | F2.0/1 | F2.5/1 | 12.4 | 17.0 | F3.1/1 | F3.5/1 | 49.9 | 56.9 | 0 |
| | poly | F2.5/1 | F2.5/1 | 13.8 | 13.8 | F3.5/1 | F3.5/1 | 57.5 | 57.5 | 0 |
| syr2k | par | F1.6/4 | F1.6/4 | 9.8 | 9.8 | F3.5/4 | F2.5/4 | 52.7 | 58.6 | 39 |
| | poly | F1.6/4 | F2.0/4 | 5.1 | 5.8 | F3.5/4 | F3.5/4 | 55.8 | 55.8 | 0 |
| syrk | par | F2.0/4 | F1.6/4 | 7.2 | 9.1 | F3.5/4 | F2.5/4 | 51.8 | 57.7 | 39 |
| | poly | F2.0/4 | F2.0/4 | 6.8 | 6.8 | F3.5/4 | F3.5/4 | 56.4 | 56.4 | 0 |
| trisolv | par | F2.0/1 | F2.5/1 | 22.3 | 23.1 | F3.5/1 | F3.5/1 | 72.0 | 72.0 | 0 |
| | poly | F1.6/4 | F1.6/4 | 2.3 | 2.3 | F2.0/4 | F2.5/4 | 26.7 | 28.8 | 15 |
| trmm | par | F2.0/1 | F2.5/1 | 21.1 | 23.9 | F3.5/1 | F3.5/1 | 71.1 | 71.0 | 0 |
| | poly | F1.6/4 | F1.6/4 | 2.0 | 2.0 | F2.0/4 | F2.5/4 | 22.9 | 27.3 | 11 |
| Average | | | | 11.3 | 12.9 | | | 45.2 | 50.52 | 11.6 |

We report the best frequency/core configuration for each binary found by empirically evaluating all frequency/core cases in the design space—frequencies of 1.2GHz, 1.6GHz, 2.0GHz, 2.5GHz, 3.1GHz, and 3.5GHz—using 1, 2, or 4 cores. We report the one predicted by our approach, OurStatic, and compare the energy savings compared to `powersave` for both cases. A similar study is shown when optimizing for energy-delay product. Averages (in % also) are reported on the last line. Finally, the last column shows the increase in execution time by using OurStatic compared to using `performance`. It is obvious that, in many cases, optimizing CPU energy can be done without being detrimental to the best possible execution time or the increase remains moderate.

For Haswell, the empirical one-time characterization of the machine by microbenchmarking as described in Section 5.1 gave the following four configurations to use when optimizing for energy: (1) 1.6GHz/4 cores for bandwidth-bound codes, (2) 2.0GHz/4 cores for compute-bound codes, (3) 1.2GHz/2 cores for codes with poor parallel scaling, and (4) 2.5GHz/1 core for sequential codes. The configurations found when optimizing for EDP were (1) 2.5GHz/4 cores, (2) 3.5GHz/4 cores, (3) 2.0GHz/4 cores, and (4) 3.5GHz/1 core. The category to which a binary was classified by the decision tree discussed in Section 5.2 can be inferred in Table IV from the frequency/core pair reported in OurStatic.

In the majority of cases, our static approach based on code classification allows us to use the best static frequency/core configuration for the benchmark, whether optimizing for energy or for EDP. There are, however, sporadic but clear cases of "failure," such as dynprog-poly or ADI-par, which correspond to codes with poor parallel scaling. This suggests our coarse approximation of parallel scaling can be improved, for example by more accurately computing the workload of the slowest thread and the expected parallel speedup.

### 6.5. Summary of Features

We conclude our experimental evaluation with Table V, which reports how many benchmarks belong to each category. This classification is done per binary and is independent

Table V. Summary of Features

| Benchmarks | seq/par | bw-bound | poor scale | comp-bound |
|---|---|---|---|---|
| polybench-parrallel | 12/18 | 27 | 4 | 1 |
| polybench-poly | 5/25 | 15 | 1 | 10 |

Table VI. Benchmark Features

| Benchmarks | version | seq | bw-bound | poor scale | comp-bound |
|---|---|---|---|---|---|
| correlation | par | | | ✓ | |
| | poly | | | | ✓ |
| gemm | par | | ✓ | | |
| | poly | | | | ✓ |
| jacobi-2d | par | | ✓ | ✓ | |
| | poly | | ✓ | | |
| seidel-2d | par | ✓ | ✓ | | |
| | poly | | | | ✓ |

of the compiler used. A benchmark may have more than one characteristic feature, which is why we use a decision tree to prioritize them, as described in Section 5.

We see that, for the simple parallelized versions, many codes are bandwidth-bound, and this decreases when using program transformations to improve data locality. Our static analysis captures how a program is implemented (i.e., its result may change as a function of, for example, which loop transformations have been applied). This is confirmed by Table VI, which zooms in on a few benchmarks, displaying how their features change depending on the structure of the code implementing the algorithm in the original benchmark (par versus poly, see Section 6.1 for details).

In particular, as tiling improves the operational intensity, we see gemm moves from being memory-bound (no tiling) to compute-bound (after tiling). On the other hand, for Jacobi-2D, tiling by itself did not improve the OI enough to make it go beyond the memory-bound cutoff point. This is expected because the best frequency/core configuration for this optimized kernel is indeed the best frequency for the bandwidth-bound codes: The program still suffers from bandwidth contention, even after the tiling we applied.

## 7. PHASE ANALYSIS

Here, we discuss the occurrences of phases in the programs we evaluated via a separate set of experiments conducted using the runtime system described in Section 3. The objective is to show that, thanks to the stability of most affine kernels we evaluated (i.e., they contain only one phase), a purely static approach where we select a single frequency/core configuration for the entire kernel duration can achieve near-optimal results. We then discuss cases where phases occur and point to future work on using Algorithm 2 to detect those phases analytically to enable the selection of different frequency/core configurations for different phases of the program.

### 7.1. Phase Characterization

We plot in Figure 8 the output of our adaptive runtime on several kernels. To better visualize phases, if any, we used a very low threshold for frequency change: As soon as there is a 1% energy-efficiency difference between two time quanta (set to 50ms), the runtime is allowed to increase/decrease frequency. Benchmarks were set to using a very large problem size for better illustration. There is one point per time quanta, and we report both the current frequency (in green) and the "instantaneous" power for the quanta (in purple). We conducted this study on all 60 benchmarks on Haswell, using four cores, and we isolated the most representative cases.

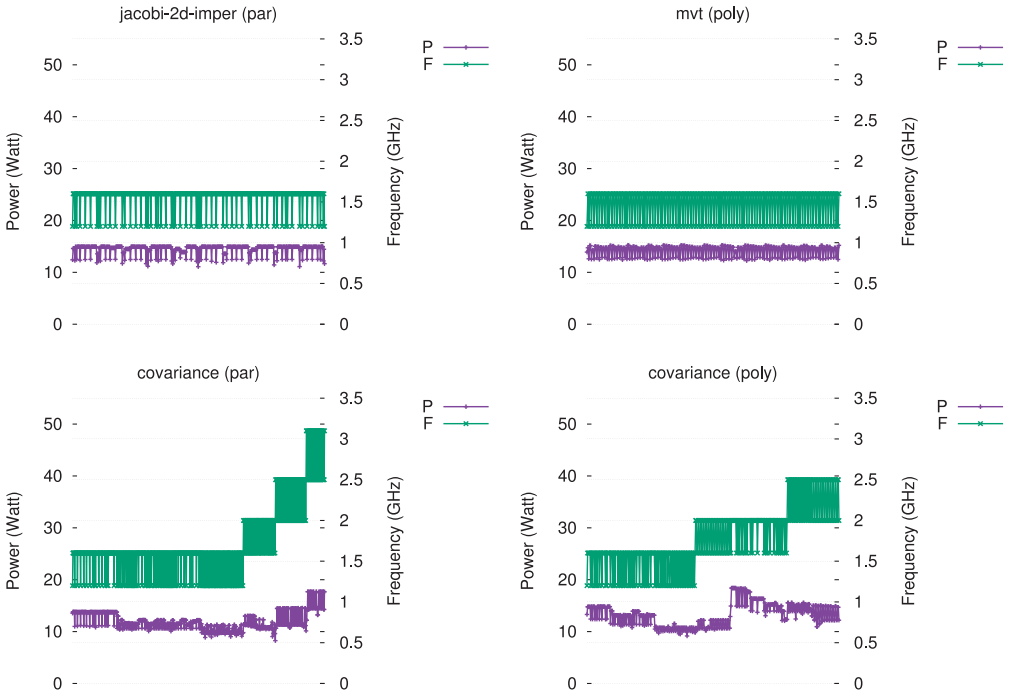


Fig. 8. Different types of phases on Haswell/4 cores.

The top charts show typical power trace examples for single-phase kernels: The power consumption is mostly stable, and the frequency typically oscillates between two of the frequency points (e.g., 1.2GHz or 1.6GHz), indicating that the optimal frequency may be between these points. We conducted this characterization for all 60 benchmarks, and, for 46 of them, there is a single phase. These includes gemm and trmm, for example. The bottom charts illustrate cases of multiple phases in the program. A total of 14 benchmarks show two or more phases (up to five), visualized as a stable change for a part of the program execution of the frequency used. The effect of program transformations to improve data locality can be visualized here: Phases are not identical between both plots, in particular since data locality was changed by the polyhedral code transformation, and, therefore, the operational intensity was modified.

### 7.2. Possible Improvements

As illustrated in Figure 8, there can be cases with stable phases in the computation, necessitating ideally different frequency/core configurations for ideal energy savings. Algorithm 2 can be applied on any subpart of the program; in particular, it can be applied on different loop nests to obtain the estimated operational intensity for different program regions. If this intensity differs, then the classification of the region using the decision tree from Section 5.2 may also change, leading to using different frequencies within the same program to be better suited to exploit its phases. As future work, we will investigate how to embed such mechanisms in our static analysis to detect program phases at compile-time.

## 8. RELATED WORK

DVFS technology, which originated in the real-time and embedded system community [Farkas et al. 2000], is widely applied nowadays since energy use and power

consumption have become the primary optimization metrics for the entire spectrum of computing, from handheld devices [Farkas et al. 2000] to desktops [Lorch and Smith 2001] to clusters [Yuki and Rajopadhye 2014].

CPU MISER by Ge et al. [2007] is a runtime approach to adapt the frequency/voltage based on the measured cycle per instructions achieved by phases of the running workload. This approach works on arbitrary binaries; however, it is geared to finding the smallest frequency that does not degrade performance.

Hsu and Kremer [2003] developed a compiler-assisted technique to change the operating voltage for specific regions of the program while attempting to control the execution time penalty, but they require an actual profiling of each application to be run, whereas we rely solely on computing compile-time characteristics for new applications. Saputra et al. studied the impact of loop transformations on static and dynamic voltage strategies to reduce power [Saputra et al. 2002]. Jimborean et al. proposed a Decoupled Access/Execute (DAE) model using access phase generation as a speculative prefetch to make the execute tasks effectively compute-bound, so that the DVFS could adjust frequency accordingly [Jimborean et al. 2014]. Our compile-time approach to DVFS could be employed on top of both approaches to further improve energy savings.

Yuki and Rajopadhye [2014] paid particular attention to the race-to-sleep scenario, especially for current supercomputers. Using simplified power equations for the CPU, and looking at the ratio between CPU power versus the power of the rest of the system, they analytically determined that massive DVFS may not be overall profitable in a race-to-sleep scenario where the entire system goes to sleep when the computation ends. While our results do not contradict theirs, when evaluating using actual programs, we observe that the execution time penalty may be minimal, thus leading to overall energy savings as demonstrated in other works [Ge et al. 2007]. In addition, the race-to-sleep scenario is not always applicable: Very often, the operating system (and the entire node) keeps running when a code/kernel is completed, so there is still large potential benefits in exploiting CPU DVFS for program segments.

Power modeling methodologies have been also studied [Diop et al. 2014; Austin and Wright 2014] to analytically study the evolution of processor power as a function of voltage, frequency, and temperature. Skadron et al. [2004] in particular studied the role of temperature on leakage, leading to a more realistic power equation. Recently, De Vogeleer et al. [2014] used measurements in a control environment on a mobile CPU to confirm a realistic power/energy equation for CPU power. They showed the existence of an energy/frequency convexity rule; that is, the existence of a unique optimum frequency for energy efficiency for a fixed workload. Interestingly, our data in Section 2 illustrate exactly their point, showing different optimal frequencies across four Intel x86/64 desktop processors. Note, however, that this principle does not necessarily hold in the case of co-execution of different workloads on the same CPU core(s).

Power and performance adaption based on DVFS is also applied for thread-level parallelism. Li and Martinez [2006] proposed a low-overhead heuristics DVFS algorithm to obtain optimal power savings. Similar studies of DVFS to improve energy efficiency also have been applied for GPUs [Mei et al. 2013; Ge et al. 2013; You and Chung 2012]. Mei et al. [2013] performed a measurement study to explore the efficiency of GPU DVFS on system energy consumption. Ge et al. [2013] proposed a GPU DVFS study of performance and energy efficiency on the Nvidia K20c Kepler GPU and compared it with CPU DVFS. Our proposed compile-time characterization of OI could help improve these works to make better DVFS decisions and could also be combined with other energy-saving approaches on different topics [Tavarageri and Sadayappan 2013; Bao 2014; Bao et al. 2014] that reduce the cache size to save energy without affecting performance.

## 9. CONCLUSION

DVFS is a well-known technique to adapt power consumption based on application demands and hardware state by modifying the frequency (and the associated voltage) at which a processor operates. Previous works have typically focused on two aspects: how to reduce frequency without much of a wall-clock time penalty to reduce the power and energy used by a computation and how to increase the frequency on demand to improve completion time and optimize the overall energy-delay product.

In this work, we demonstrated inherent limitations to existing simple DVFS approaches due to processor-specific and application-specific effects that must be considered. We demonstrated that for a class of computations—namely, affine programs—we can develop a *compile-time* categorization of programs by approximating their OI and parallel scaling. This allows us to automatically select at compile-time the best frequency and number of cores to use to optimize CPU energy or the energy-delay product without the need for any application profiling or runtime monitoring. Our evaluation on 60 benchmarks and five multicore CPUs validated our approach, obtaining significant CPU energy savings and EDP improvements over the `powersave` Linux governor.

## REFERENCES


Brian Austin and Nicholas J. Wright. 2014. Measurement and interpretation of micro-benchmark and application energy use on the cray XC30. In *Proceedings of E2SC*. 51–59.

Wenlei Bao. 2014. *Power-Aware WCET Analysis*. Master's thesis. Ohio State University.

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic verification of iteration space transformations on affine programs. In *Proceedings of POPL*. ACM, 539–554.

Wenlei Bao, Sanket Tavarageri, Fusun Ozguner, and P. Sadayappan. 2014. PWCET: Power-aware worst case execution time analysis. In *Proceedings of ICPPW*. IEEE, 439–447.

Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of PACT*. 7–16.

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. PLUTO: A practical and fully automatic polyhedral program optimization system. In *Proceedings of PLDI*.

Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of PLDI*. ACM, 286–297.

Karel De Vogeleer, Gerard Memmi, Pierre Jouvelot, and Fabien Coelho. 2014. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *Parallel Processing and Applied Mathematics*. Vol. 8384. Springer Berlin, 793–803. DOI:http://dx.doi.org/10.1007/978-3-642-55224-3_74

Tahir Diop, Natalie Enright Jerger, and Jason Anderson. 2014. Power modeling for heterogeneous processors. In *Proceedings of GPGPU*.

Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M. Anderson. 2000. Quantifying the energy consumption of a pocket computer and a Java virtual machine. *ACM SIGMETRICS Performance Evaluation Review* 28, 1 (2000), 252–263.

P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash. 1991. On estimating and enhancing cache effectiveness. *LCPC* 589 (1991), 328–343.

M. Floyd, B. Brock, M. Ware, K. Rajamani, A. Drake, C. Lefurgy, and L. Pesantez. 2010. Harnessing the adaptive energy management features of the power7 chip. *HOT Chips 2010* (2010).

Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W. Cameron. 2007. CPU miser: A performance-directed, run-time system for power-aware clusters. In *Proceedings of ICPP*. 18–25.

Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. 2013. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *Proceedings of ICPP*. 826–833.

Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 4 (1999), 703–746.

Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* 34, 3 (2006).

Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2016. Effective padding of multidimensional arrays to avoid cache conflict misses. In *Proceedings of PLDI*. ACM, 129–144.

Chung-Hsing Hsu and Ulrich Kremer. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of PLDI*. ACM, 38–48.

Intel. Intel Math Kernel Library (Intel MKL). https://software.intel.com/en-us/intel-mkl.

Intel. Intel Performance Counter Monitor. www.intel.com/software/pcm.

Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the code. Don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of CGO*. ACM, 262.

Jian Li and Jose F. Martinez. 2006. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proceedings of HPCA*. 77–87.

Jacob R. Lorch and Alan Jay Smith. 2001. Improving dynamic voltage scaling algorithms with PACE. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 29. ACM, 50–61.

John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report. Retrieved from http://www.cs.virginia.edu/stream/.

Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. 2013. A measurement study of GPU DVFS on energy conservation. In *Proceedings of Workshop on Power-Aware Computing and Systems*. 10.

Netlib. Netlib BLAS. Retrieved from http://www.netlib.org/blas/index.html.

OpenCV. OpenCV: Open Source Computer Vision Library. Retrieved from http://opencv.org.

PoCC, the Polyhedral Compiler Collection, version 1.3. Retrieved from http://pocc.sourceforge.net.

PolyBench/C 3.2. Retrieved from http://polybench.sourceforge.net.

Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of FPGA*.

H. Saputra, M. Kandemir, and others. 2002. Energy-conscious compilation based on voltage scaling. In *Proceedings of LCTES*.

Vivek Sarkar. 1997. Automatic selection of high order transformations in the IBM XL Fortran compilers. *IBM Journal of Research & Development* 41, 3 (May 1997).

Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noel Pouchet. 2014. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, Tiziana Margaria and Bernhard Steffen (Eds.). Lecture Notes in Computer Science, Vol. 8803. Springer Berlin Heidelberg, 493–508. DOI:http://dx.doi.org/10.1007/978-3-662-45231-8_41

Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. 2004. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization* 1, 1 (March 2004), 94–125. DOI:http://dx.doi.org/10.1145/980152.980157

Sanket Tavarageri and P. Sadayappan. 2013. A compiler analysis to determine useful cache size for energy efficiency. In *Proceedings of IPDPSW*. IEEE, 923–930.

Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*. Springer, 299–302.

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2009. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification*. Springer, 599–613.

S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48, 1 (June 2007), 37–66.

Daecheol You and K.-S. Chung. 2012. Dynamic voltage and frequency scaling framework for low-power embedded GPUs. *Electronics Letters* 48, 21 (2012), 1333–1334.

Tomofumi Yuki and Sanjay Rajopadhye. 2014. Folklore confirmed: Compiling for speed = compiling for energy. In *Proceedings of LCPC*. 169–184.

Received June 2016; revised September 2016; accepted October 2016