Aequitas: Coordinated Energy Management Across Parallel Applications

Haris Ribic
SUNY Binghamton
Binghamton NY 13902, USA
hrubic1@binghamton.edu

Yu David Liu
SUNY Binghamton
Binghamton NY 13902, USA
davidL@binghamton.edu

ABSTRACT
A growing number of energy optimization solutions operate at the application runtime level. Despite delivering promising results, these application-scoped optimizations are fundamentally greedy: They assume to have an exclusive access to power management and often perform poorly when multiple power-managing applications co-exist, or different threads of the same application share power management hardware. In this paper, we introduce Aequitas, a first step to address this critical yet largely overlooked problem. The insight behind Aequitas is that co-existing applications may view power-managing hardware as a shared resource and coordinate power management decisions. As a concrete instance of this philosophy, we evaluated our ideas on top of a state-of-the-art energy-efficient work-stealing runtime. Experiments show that without Aequitas, multiple co-existing power-managing application runtimes suffer up to 32% performance loss and negate all power savings. With Aequitas, the beneficial energy-performance tradeoff reported in the single-application setting (12.9% energy savings and 2.5% performance loss) can be retained, but in a much more challenging setting where multiple power-managing runtimes co-exist on parallel architectures and multiple CPU cores share the same power domain.

CCS Concepts
• Software and its engineering → Power management;
• Computing methodologies → Parallel programming languages;

Keywords
Energy Management; Parallelism; Work Stealing; DVFS

1. INTRODUCTION
As cloud servers, scientific computing clusters, and even mobile multi-cores increasingly dominate today’s computing world, energy efficiency of parallel systems is becoming critical to the future of computing. Energy efficiency has become a first-class design goal with direct consequences on operational cost, reliability, usability, maintainability, and environmental sustainability. Two philosophies largely guide existing solutions: those that view the applications as a black box and those that view them as a white box. Black-box solutions make power/energy management decisions based on hardware performance counters and operating system states during application execution (e.g., [19]). In contrast, white-box solutions make optimization decisions based on the inspection of the application runtime such as control-flow and data-flow dependencies between application threads [20, 44, 16, 41, 3, 2, 36, 8, 21, 48, 47, 42, 45]. At a time when pure black-box solutions become more established and the options for their further optimizations dwindle, white-box solutions are gaining attention, especially for their benefits of using application-specific knowledge for optimization, and their ability for fine-grained energy management.

Promising as they are, wide adoption of white-box optimizations face a fundamental hurdle as they are greedy in nature when viewed from the whole-system perspective. These optimizations are often designed as if the application of concern was the only one executing on the underlying system and could make exclusive power management decisions. However, this is often too idealistic for real-world deployments. Consider a white-box solution that uses application runtime knowledge to guide Dynamic Voltage and Frequency Scaling (DVFS) [15, 7] – a strategy commonly adopted by existing energy-aware programming languages [36, 3, 8] and compilers [20, 44, 16] – and is deployed in the two scenarios presented in Figure 1:
• **CO-EXISTING APPLICATIONS:** Multiple applications often share the same underlying system. Assume applications A and B co-exist as in Figure 1(a). Consider a white-box approach guiding A and its thread Xₐ to set the frequency to 2GHz whereas it is guiding B and its thread Xₐ to set the frequency to 1GHz. The two are deployed on the same CPU core. Which frequency should be set?

• **CO-MANAGED POWER DOMAINS:** CPU cores often share the same power management hardware as in Figure 1(b). Consider a white-box approach guiding A and its thread Xₐ to set the frequency to 1GHz and Yₐ to set the frequency to 2GHz. The two are deployed on two cores within the same power domain. Which frequency should be set?

This paper presents **Aequitas**, an effective strategy to coordinate energy management across multiple multi-threaded applications over parallel architectures. The key insight is that the challenges faced by CO-EXISTING APPLICATIONS and CO-MANAGED POWER DOMAINS are a symptom of contention over shared power-management hardware — either by different applications or by different threads within the same application. The core of **Aequitas** is a coordinated round-robin algorithm that allows different applications and different threads within each application to access the underlying hardware power management within their share of time.

To evaluate our system, we selected **Hermes** [34] as a baseline. **Hermes** is an energy-efficient multi-threaded work-stealing language runtime with a sophisticated algorithm to make semantics-aware DVFS decisions. Originally, **Hermes** reported 12.9% energy savings and 2.5% performance loss in a single-application optimization setting where each thread is deployed to a distinct power domain. However, in the presence of CO-EXISTING APPLICATIONS and CO-MANAGED POWER DOMAINS, all reported energy management benefits of **Hermes** are erased with a significant performance loss of 32% in some cases (see Section 2). In contrast, **Aequitas** is capable of retaining nearly all of the benefits reported by **Hermes**, but in the presence of much more challenging scenarios where multiple applications coexist and their threads are deployed on cores that share power domains i.e., scenarios presented in Figure 1.

During our design space exploration, we also considered alternative contention resolution policies for shared power management such as a first-come-first-serve style or a policy to average the CPU frequencies requested by contending parties. Comparatively, **Aequitas** achieves the best energy-performance tradeoff. Our results are stable across a variety of (homogeneous and heterogeneous) combinations of applications and different configurations of the underlying systems. Furthermore, **Aequitas** design is independent of the specifics of the **Hermes** algorithm.

To the best of our knowledge, **Aequitas** is the first system of its kind to gracefully extend application-runtime-level energy optimization to a more holistic setting. Concretely, our work makes the following contributions:

• A practical system to coordinate multiple applications guided by greedy energy optimizations on shared power domains, with a key insight that transforms an energy management problem into a contention management problem.

• A novel design with time-sliced power management, where each application and each thread within an application receives a share of time to regulate power. The time-slicing strategy for power management is further decoupled from that of OS-level scheduling.

• A thorough design space exploration covering alternative contention resolution policies beyond time slicing, and an “under the hood” study to analyze the reasons behind **Aequitas’** effectiveness.

<table>
<thead>
<tr>
<th>Application</th>
<th>Power Domain</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1; 2; 3; 4; 5; 6; 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: An illustration of worker-to-module mapping in **Hermes**.

## 2. BACKGROUND

This section provides an overview of the baseline **Hermes** system, together with a review of our targeted parallel architectures.

**Hermes Overview.**

**Hermes** [34] is an energy-efficient work-stealing runtime built on top of Intel Cilk Plus, which traces its roots to MIT Cilk [5, 6, 14]. The design of Aequitas is independent of both the algorithm details of classic work stealing and those of **Hermes**, but a review may provide a concrete picture of the challenging context to which **Aequitas** is aimed to address.

Work stealing is a classic fine-grained task management strategy over parallel architectures, known for its strength in load balancing and scalability. Under work-stealing runtimes, a program consists of tasks and a pre-assigned number of threads (called workers) competing to process these tasks. In practice, the number of workers is often equal to the number of available CPU cores. Each worker maintains a queue-like data structure called a **double-ended queue** or **deque** containing tasks to be processed. When a worker finishes processing a task, it picks up a next one from its deque and continues the execution of that task. When its deque is empty, a worker steals a task from the deque of another worker. In this case, the stealing worker is called a **thief** whereas the worker whose task is stolen is called a **victim**. The selection of victims follows the principles observed by load balancing and may vary in different implementations of work stealing.

**Hermes** is a white-box runtime-level energy optimization and takes advantage of the inherent features within the work-stealing runtimes to guide DVFS. A worker (thief) who steals a task from another worker (victim) may form a data dependency relationship. Slowing down the thief may not significantly affect the overall performance of the application, but may reduce energy consumption. Therefore,
threads in a work-stealing environment – thieves and victims – have varying impacts on the overall program running time.

Hermes coordinates DVFS frequencies associated with each thread, so that in the scenario above, a workpath-sensitive algorithm will determine the state of each worker in a thief-victim relationship and set DVFS frequencies accordingly. A second, workload-sensitive, algorithm samples the deque size and selects appropriate DVFS frequency based on it. Hermes unifies of the two algorithms into one and a combination of the two provides an energy optimization that is aware of the states and semantics of the work-stealing program runtime.

In the original Hermes, each worker has an exclusive access to power management, and as shown in Figure 2, one worker is mapped to each power domain module. Consequently, Hermes uses only 50% of available CPU cores. The reported data showed an average of 11-12% energy savings with an average of 3-4% performance loss. The latest version of their system reports a 12-13% energy savings with an average of 2-3% performance loss through some optimizations. This optimized version of Hermes is the baseline of our experiments.

We performed Hermes “stress test” by extending it to support Co-Existing Applications and Co-Managed Power Domains as presented in Figure 3. Labels on the X-Axis describe the representative benchmarks (see Section 4) and the number of benchmarks executed in parallel. For example, label KNN indicates that an instance of benchmark KNN is executed, whereas 4 KNN indicates that four KNN instances are executed in parallel. Labels on the Y-Axis are the normalized time of Hermes-based execution w.r.t. Intel Cilk Plus, deployed on a system with an 8-core CPU where every two cores share the same power domain. Each application instance consists of 8 threads, each mapped to a CPU core.

As shown in Figure 3, a single application with Co-Managed Power Domains yields 5-8% slowdown w.r.t. Intel Cilk Plus. When Co-Existing Applications are allowed, performance further degrades to a 32% slowdown for the case where 4 instances of KNN co-execute. Performance degradation at this level is unacceptable for most applications.

Nonetheless, we believe Hermes is an ideal client for Aequitas for a number of reasons. First, work-stealing is a promising approach among runtime systems for constructing parallel applications. It attracted significant interest in recent years including a growing number of commercial implementations [18, 17, 26, 25, 9, 30, 24, 29]. Second, the key feature of work stealing – its natural support for dynamic load balancing – happens to be a classic energy management strategy as well [32]. In that sense, work-stealing systems are “pro-green” from onset. Third, Hermes improves energy efficiency of work-stealing runtimes through fine-grained DVFS-based energy management. Retaining the energy efficiency of Hermes sets a “high bar” for evaluation of Aequitas.

### Parallel Architectures with Co-Managed Power Domains.

Modern CPUs generally support basic power management operations, such as DVFS and power state adjustment. The CPU cores that can only be adjusted at the same time belong to the same power domain (or frequency domain). Ideally, every CPU core would be equipped with its own power management module, but in the real world, most commercial CPUs do not have a per-core-per-power-domain design due to costs and other design constraints.

The hardware reality often poses challenges in designing fine-grained algorithms to regulate power management. For instance, Hermes could only take advantage of half of the cores available in the AMD Piledriver/Bulldozer architectures, because every two cores share the same power domain. With Aequitas, the variations in the topology of power domains will no longer hamper fine-grained white-box energy management.

## 3. AEQUITAS DESIGN

In this section, we describe the core algorithm adopted by Aequitas, together with alternative design ideas.

### High-Level Design.

We transformed the dual challenges described in Section 1 into a contention management problem – that is – how can we manage and resolve access contentions to power management, provide equal opportunity access to power management, and avoid excessive DVFS that leads to performance degradation? Aequitas answers these questions with a number of interconnected ideas:

![Figure 4: An Illustration of Aequitas with Round-Robin Time-Slicing Algorithm for Power Management.](image)
1. Power management modules are accessed through coordinated time slicing, where multiple applications take turns dominating the power management to address the Co-Existing Applications problem.

2. Threads deployed within an application on the same power domain also employ coordinated time slicing and take turns dominating the power management to address the Co-Managed Power Domains problem.

3. Domination only implies exclusive access to all power management “knobs” and not the CPU itself. Only dominating threads can adjust CPU frequencies in a DVFS-based power management strategy but every thread – including the non-dominating ones – still execute based on the OS decision and allocation.

4. The size of the dominating time slice is independent of the time-sharing OS scheduler. Aequitas fundamentally decouples the overhead of power management – such as CPU frequency switch cost – from that of OS-level context switch.

The fourth Aequitas design idea is set in contrast with a naive approach that suggests piggybacking the context switch of power management – such as preserving and recovering CPU frequencies for each application – over that of the OS. The unfortunate consequence of such approach is that every time a thread switches in, an action to the power management module must be performed. Power management, such as DVFS, is known to have a switching cost in the tens and hundreds of microseconds. The naive approach would not only incur excessive overheads, but also hamper the independence between OS scheduler design and energy management design. As a result, Aequitas enjoys coordinated and equalitarian power management, where each application and each thread within an application receives a share of time to regulate power.

Equalitarian Coordination with Two-Tiered Domination.

Aequitas is endowed with a “resource” contention mediator to coordinate DVFS-based power management among multiple competing energy-efficient applications and threads within those applications. Aequitas resolves power management contention using a round-robin time-slicing approach. Unlike OS schedulers, the goal of time slicing in Aequitas is not to schedule CPU use but resolve resource of contention and regulate power management. We call this flavor of contention resolution the domination of power management or domination for short. Domination is supported at two levels:

- Co-existing applications take time-sliced turns to manage power. When a time slice is held by an application, we also informally call the application the app dominator, and the time slice it holds the app domination slice.
- Threads (of the same application) deployed on the same power domain take turns to manage power of the co-managed power domain. When a time slice is held by a thread, we informally call the thread the pd dominator, and the time slice it holds the pd domination slice.

Two design aspects are critical to understanding Aequitas and its advantages:

- **Double Domination of Power Management**: Power management “knobs” can only be accessed – i.e., DVFS can only be issued – from a dominating thread, and if the host application of that thread is the app dominator.

- **No Domination ≠ No Progress**: A non-dominating multi-threaded application can still execute but cannot make power adjustments.

In Aequitas, each dominating time slice is assigned an equal portion and in a round-robin order; however, the app domination slices and the pd domination slices are of different magnitudes. Figure 4 is a visual illustration of Aequitas with 2 applications, each with 4 threads/workers deployed on 4 CPU cores pairwise sharing 2 power domains. Darkened color in the application rectangles illustrates application-level domination. While applications execute, Aequitas switches between workers within an application based on the pd domination slice. Figure 4(A) shows half of the workers (rectangles with darkened color) within Application A dominate. The other half of workers still execute, but their DVFS requests are no-ops. In Figure 4(B), the dominating worker group is switched. In Figure 4(C) and (D), Application B becomes the app dominator and a similar switching process with its workers ensues.

In Aequitas, the pd domination slice is smaller than the app domination slice. If the opposite were true, half of an application’s workers may dominate during the entire app domination slice. In the worst case, whenever an application becomes dominant again, the same half of the workers would dominate again, leading to inequality on the worker level.

Two advantages exist with the time-sliced round-robin design. First, it promotes equalitarian power management. Through app domination, each application receives an “equal share” of time where it can set frequency. Within an application, through pd domination, each thread deployed to the same power domain also receives an “equal share” of time to set frequency. It should be straightforward to extend this design to support a “proportionally fair” system where the time slice sizes are proportional to application/thread priorities or system resource needs. Second, the round-robin design significantly reduces the number of contending DVFS actions, and their ensuing overhead.

Phase-based Domination Abdication.

Applications are long known to display phased behaviors [38, 39, 40, 37]. This is especially true for parallel applications, where parallel phases and serial phases often alternate. One design consideration of Aequitas is how the round-robin domination should behave in the presence of prolonged inactivity of a dominating worker. If the dominating worker in a dominating application happens to be inactive, it would be a waste of opportunity for it to stay in the dominator role and prevent other workers from performing DVFS. Motivated by this insight, our algorithm is phase-aware: if the dominating worker is in an inactive phase, it abdicates its domination, and Aequitas passes on the domination to the module co-worker of the same app.

Algorithm Details.
Algorithm 0.1 Main Scheduler
1: w : Worker
2: procedure scheduler(w)
3: SET AFFINITY(w.cpu)
4: while !work done do
5: if w.scale == w.G.pdDmntr
6: then
7: SCHEDULE(w)
8: else
9: ENDAFFINITY(w.cpu)
10: END procedure

Algorithm 0.2 Schedule Work
1: w : Worker
2: procedure schedule(w)
3: procedure workerApps(w)
4: loop
5: if w.scale == w.G.pdDmntr
6: then
7: Scale Up
8: else
9: Scale Down
10: end if
11: end loop
12: end procedure

Algorithm 0.3 Scale Up
1: w : Worker
2: procedure scaleUp(w)
3: if w.G.coreGlobal[w.cpu].remove(w.self)
4: and w.G.coreGlobal[w.cpu].size == 0
5: then
6: G.coreGlobal[w.cpu].remove(w.self)
7: w.scale == w.G.pdDmntr
8: w = w + 1
9: end if
10: end procedure

Algorithm 0.4 Scale Down
1: w : Worker
2: procedure scaleDown(w)
3: if w.G.coreGlobal[w.cpu].remove(w.self)
4: and w.G.coreGlobal[w.cpu].size == 0
5: then
6: G.coreGlobal[w.cpu].remove(w.self)
7: w.scale == w.G.pdDmntr
8: w = w - 1
9: end if
10: end procedure

Worker Structure
Structure Worker w
self // application worker id
scale // scale order
cpu // cpu utility
child // app scheduler
G // global state
end structure

Algorithm 0.5 Make Worker
1: w : Worker
2: procedure makeWorker(w, g)
3: SET AFFINITY(w.cpu)
4: while !work done do
5: SCHEDULE(w)
6: end procedure
7: END procedure

Algorithm 0.6 Schedule Apps
1: w : Worker
2: procedure scheduleApps(w)
3: BIND(w.G.appDmntr)
4: BIND(w.G.pdDmntr)
5: BIND(w.G.liveApp)
6: BIND(w.G.coreGlobal)
7: w.cpu = w.cpu[w.self]
8: w.G.coreGlobal[w.cpu].add(w.self)
9: w.scale == w.G.pdDmntr
10: w = w + 1
11: end if
12: end procedure

Algorithm 0.8 Schedule Apps
1: w : Worker
2: procedure scheduleApps(w)
3: BIND(w.G.appDmntr)
4: BIND(w.G.pdDmntr)
5: BIND(w.G.coreGlobal)
6: if w.G.coreGlobal[w.cpu].size == 0
7: then
8: G.coreGlobal[w.cpu].remove(w.self)
9: w.scale == w.G.pdDmntr
10: w = w + 1
11: end if
12: end procedure

Alternative Contention Resolution.
When two threads from Co-Existing Applications or Co-Managed Power Domains desire different CPU frequencies, alternative contention resolution possibilities exist to resolve the conflict. During our design exploration that led to round-robin time-slicing approach adopted by Aequitas, we also considered alternatives. For example, let us consider that there are two contending workers. Other possible contention resolution policies are (for all policies, frequency is only changed if it differs from current frequency):

- First Comer Dominates (FCD): CPU frequency is set by the worker who is first scheduled to execute a task in the shared power domain, and the worker scheduled later does not perform DVFS until first worker finishes;
- Last Comer Dominates (LCD): opposite of FCD;
- Higher Frequency Dominates (HFD): CPU frequency is set to the highest level desired by the two workers scheduled to the power domain;
- Lower Frequency Dominates (LFD): opposite of HFD;
- Average Frequency Dominates (AFD): CPU frequency is set to average of all requests.

We have implemented these alternative policies, whose benchmarking results will be discussed in Section 5.5. None is as effective as the round-robin time-slicing approach adopted by Aequitas. More importantly, these policies are application-blind, which sacrifices equalitarian coordination in energy management. Nonetheless, we believe these results complement Aequitas as a part of the complete design space exploration.

4. IMPLEMENTATION AND EXPERIMENTAL SETUP

Aequitas was implemented on top of Intel Cilk Plus (build 2586). In this section, we present our experimental setup(s).

We selected benchmarks from the Problem-Based Benchmark Suite (PBBS) [4]. PBBS is widely used for benchmarking parallel applications and contains a subset of benchmarks that support work stealing. Figure 6 presents a 100ms snapshot overlay of worker 1 from each benchmark which shows that, comparatively, individual benchmarks want to make a different energy management decisions. The details of our selected benchmarks are defined in Table 1.
To validate the effectiveness of our approach, we constructed our experiments on two platforms as shown in Table 2. In both systems the default Completely Fair Scheduler (CFS) in used in Linux. CFS is known to have no fixed time slices for scheduling, even though the ballpark is in the magnitude of tens of milliseconds. For both Piledriver and Bulldozer architecture, every two cores share a frequency domain.

Energy consumption is measured through current meters over power supply lines to the CPU module, including the cores and the uncores (such as on-chip caches and interconnects). For CPU-intensive benchmarks – to which work-stealing applications/benchmarks belong to – these hardware components are the ones whose energy consumption is most likely to vary. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress [10] with 100 samples per second. The supply voltage is stable at 12V, and we compute energy consumption as the sum of current samples multiplied by 12 × 0.01.

We executed each benchmark using Aequitas runtime. Since Aequitas manages DVFS at the application runtime level, the Linux governor is set to userspace. All energy and performance results reported in Section 5 are normalized to a control, defined as executing the same benchmark (in the same configuration) with the unmodified Intel Cilk Plus (build 2856) runtime executing against default Linux ondemand governor. For each benchmark, we ran 20 trials (disregarding the first 2 trials) and calculated the average of the trials.

**HERMES Comparison.**

Our experimental evaluation also involved some quantitative comparisons against HERMES. HERMES runtime does not define behaviors when the number of workers exceeds the number of power domains. Aequitas is capable of handling Co-MANAGED POWER DOMAINS and our experiments use all CPU cores, even if some share the same power domain. To construct a fair comparison, we updated HERMES source code to support the same number of workers as CPU cores e.g., increasing the number of workers from 4 to 8 on a 8-core machine where every two cores share a power domain. Initially, we did not explicitly bind the additional 4 workers to any core. The experimental results were poor: the co-execution of two instances of Ray incurred a 167% slow down w.r.t. Intel Cilk Plus. We speculate this is due to a loss of cache locality, combined with unpredictable contentions of DVFS. We decided that a more fair comparison would be to bind workers to cores in the identical fashion as Aequitas. This is the setting we used for all experiments reported in the paper.

### Table 1: Benchmark Descriptions.

<table>
<thead>
<tr>
<th>Benchmark Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-Nearest Neighbors (KNN)</td>
</tr>
<tr>
<td>Uses pattern recognition methods to classify objects based on closest training examples in the feature space.</td>
</tr>
<tr>
<td>Ray-Triangle Intersection (Ray)</td>
</tr>
<tr>
<td>For each ray, it calculates the first triangle it intersects given a set of triangles contained inside a 3D bounding box and a set of rays that penetrate the box.</td>
</tr>
<tr>
<td>Integer Sort (Sort)</td>
</tr>
<tr>
<td>Sorting fixed-length integer keys into ascending order with the ability to carry along fixed-length auxiliary data.</td>
</tr>
<tr>
<td>Comparison Sort (Compare)</td>
</tr>
<tr>
<td>Similar to Sort but uses sample sort.</td>
</tr>
<tr>
<td>Convex Hull (Hull)</td>
</tr>
<tr>
<td>Computational geometry benchmark.</td>
</tr>
</tbody>
</table>

### Table 2: Systems Configurations.

<table>
<thead>
<tr>
<th>System A</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 2×16-core AMD Opteron 6378 (Piledriver)</td>
</tr>
<tr>
<td>• Debian 7.0 Linux (Kernel 3.2.49)</td>
</tr>
<tr>
<td>• 64GB DDR3 1600 RAM</td>
</tr>
<tr>
<td>• Supports 5 distinct frequencies: 1.4GHz, 1.6GHz, 1.9GHz, 2.2GHz, 2.4GHz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System B</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 8-core AMD FX-8150 (Bulldozer)</td>
</tr>
<tr>
<td>• Debian 7.0 Linux (Kernel 3.2.49)</td>
</tr>
<tr>
<td>• 16GB DDR3 1600 RAM</td>
</tr>
<tr>
<td>• Supports 5 distinct frequencies: 1.4GHz, 2.1GHz, 2.7GHz, 3.3GHz, 3.6GHz</td>
</tr>
</tbody>
</table>

5. **EVALUATION**

This section presents detailed energy and time results of Aequitas on Systems A and B w.r.t. Intel Cilk Plus. To gain confidence in the applicability of Aequitas, we conducted design space explorations under different application, algorithm, and platform settings. In Section 5.5, we also report results for alternative contention resolutions policies.

5.1 **Overall Results**

Figure 7 shows the effectiveness of Aequitas w.r.t. Intel Cilk Plus in the presence of Co-EXISTING APPLICATIONS and Co-MANAGED POWER DOMAINS. In Figure 7, black bars are normalized energy savings and gray bars are normalized time losses, both w.r.t. Intel Cilk Plus. A label such as 4Ray means that four Ray instances are executed in parallel. Each of them has the same number of threads/workers as the number of CPU cores. We show the results when 2, 3, and 4 applications (from the same benchmark) execute in parallel. Recall that both System A and B are supported by architectures where every pair of CPU cores share a power domain.

Overall on System A, Aequitas achieves an average energy savings of 14% with performance loss averaging 2.5%. On System B, Aequitas achieves an average energy savings of 12-13% also with performance loss averaging 2.5%.
In single benchmark testing, the results show that Aequitas outperforms Hermes w.r.t. Intel Cilk Plus on System A and slightly outperforms Hermes on System B – recall that the optimized Hermes demonstrated 12-
13% energy savings with an average of 2-3% performance loss. AEQUITAS operates in a much more adversarial setting where multiple applications co-exist and multiple threads co-manage the same power domain. Improved energy efficiency is due to AEQUITAS’ ability to scale down unused CPU’s to a minimum frequency, avoid unnecessary DVFS calls, and its ability to use all available CPU cores. With the support of Co-MANAGED POWER DOMAINS, we can now deploy two workers to the cores that belong to the same power domain. By hardware design, the two cores also share more levels of caches, leading to more favorable cache locality and a lesser performance loss of AEQUITAS compared to HERMES w.r.t. Intel Cilk Plus.

Furthermore, unlike HERMES, where threads/workers can always execute power management decisions as they wish, a thread under AEQUITAS cannot manage power if it is not dominating. Experimental results show that the idea of time-sliced power management can be nearly as effective as exclusive power management.

Lastly, observe that AEQUITAS is stable across platforms. The patterns in relative savings/loss on System A and B are nearly identical, and their relative standings are specific to the benchmarks themselves. The stability across System A and B is manifested in nearly all of our experiments.

5.2 Homogeneous vs. Heterogeneous Applications

We also experimented in a setting where multiple applications are different benchmarks executing in parallel. Figure 8 presents the combinations of all benchmarks with legends/configurations identical to Figure 7. Energy data are reported based on the data collected between the common starting time of all applications and the completion time of the last application. On System A, AEQUITAS achieves an average energy savings of 13% with a slight performance loss averaging 2.8%. On System B, AEQUITAS achieves an average energy savings of 12% also with a slight performance loss averaging 2.5%. Unlike homogeneous settings where multiple applications complete roughly at the same time, different applications in the heterogeneous setting may complete at very different times. The most important observation is that the benefits of AEQUITAS in the homogeneous setting are retained for the heterogeneous combinations of applications.

5.3 Workload Level

We also studied the impact of various workload levels on AEQUITAS w.r.t Intel Cilk Plus with the results shown in Figure 10 (including the case where the number of application is 2x that of physical cores). There is a small but noticeable decline in the benefits produced by AEQUITAS with an increase in the number of applications executing in parallel and the degree of decline is benchmark-specific.

The decline does not come as a surprise. Consider 2KNN and 16KNN for instance. It is 8x more likely that an application in the latter execution – than the one in the former – wishes to adjust the CPU frequency to a particular level, but cannot because it is not a dominator. Figure 10 shows that despite the exponential decline in the odds of executing the “best wish”, the energy/performance benefits do not exponentially precipitate. From that angle, Figure 10 is reasonably good news. It also says 16KNN and 16Ray may remain a viable configurations for some energy-conscious users.

The more relevant question is whether executing 16 CPU-intensive applications concurrently represents a typical use scenario in the real world. To the best of our knowledge, such cases are most common where a portion or all such applications are “background” in nature, or are I/O-bound, or interactive (e.g., [22]). Work-stealing runtimes are naturally CPU-intensive – it is long known that frequent I/O or synchronization would greatly hamper the stealing process – we speculate a scenario such as 16KNN would be a rare event.

Figure 10: Energy/Performance in the Presence of Varying Workload Levels.

5.4 Time Slice Choices

For the results reported so far, the app/pd domination slices are fixed at 1second/0.2seconds respectively. We also studied the impact of varying time-slice sizes. Figure 9 shows the results of concurrently executing 4 of each KNN, Ray, and Hull benchmarks respectively, at various time slice choices w.r.t. Intel Cilk Plus. The two numbers at the bottom of each bar are the time slice choices. For instance, 0.5_125 means the app domination slice is 0.5 seconds, and the pd domination slice is 0.125 seconds. The rest of the legends/configurations are identical to Figure 7.

A common trend is that the results for performance loss exhibits an “inverse U” shape: both very small and very large time slices incur more performance loss than slices in the middle. For all 3 benchmarks, performance quickly deteriorates when the app domination slice is reduced to 0.5 seconds. Here, the overhead of DVFS becomes a factor. On the other hand, when time slices become too coarse, the effectiveness of AEQUITAS – its ability to allow each application to make the decisions it wishes – is discounted. As a result, as we move from the center to the right for each figure, performance also deteriorates for each benchmark.

Readers might wonder why a time slice such as 0.25 seconds for power domain domination does not incur excessive DVFS overhead (such as the naive solution may). The key observation here is that DVFS only occurs in the presence of Double Dominations (see Section 3) and frequency is only set if it differs from current CPU frequency i.e. even if requested, there is no need to honor DVFS if it is requesting the same frequency at which a core operates.

5.5 Alternative Contention Resolutions

Figure 11 shows normalized time loss of the 5 alternative contention resolution policies w.r.t. Intel Cilk as described in Section 3. We can quickly eliminate LCD, LFD and AVG...
from viable solutions. Even for a single application (with co-managed power domains), the three solutions already yield a performance loss from 7% to 37%. The poor performance of LCD and AVG results from frequent CPU scaling. These strategies blindly switch CPU frequencies for almost every work-stealing task scheduled. It comes, as no surprise LFD is sub-optimal as it always favors executing at the lowest frequency.

On the other hand, FCD and HFD remain competitive. Indeed, FCD shows an average of 2.5% performance loss, and for HFD only 0.6%, both w.r.t. Intel Cilk. Their energy savings however are less competitive when compared to the round-robin time-slicing strategy adopted by AEQUITAS. Even though HFD achieves the best performance w.r.t. Intel Cilk, it also saves the least energy.

Among the alternative policies studied, FCD is the most competitive when compared to round-robin time-slicing strategy. Figure 12, presents the results of a scenario where 4 work-stealing applications co-exist. In the presence of multiple applications, FCD does not scale as well. The root problem with FCD is its application blindness. Philosophically, FCD is blind to the intra-application coordination – a hallmark of white-box approaches such as AEQUITAS – and may inadvertently make counter-intuitive decisions. The experimental result shows the cost of application blindness. It experimentally explains the benefits of AEQUITAS’s two-tier domination design.

5.6 Operating System Scheduler

As mentioned before, OS’s on both system A and B use the CFS. Our Linux installations have also the O(1) scheduler available. Re-run of our experiments with O(1) scheduler showed a margin of ±0.5% on average across benchmarks for power and energy w.r.t. Intel Cilk Plus.

6. UNDERSTANDING AEQUITAS BEHAVIORS

In this section, we conduct an “under-the-hood” comparative analysis of AEQUITAS and HERMES. It is our belief that the reasons why time-sliced power management produces good results are just as important as – if not more important than – the results themselves.

6.1 Frequency Scaling Traces

We first present a time series for frequency scaling of a representative benchmark execution, in this case a heterogeneous co-execution of Ray and Compare. The entire trace is shown in Figure 13 as a “bird’s view.”

Figures 13(a)(b) present AEQUITAS behavior, whereas Figures 13(c)(d) present HERMES behavior. In both experiments, each application consists of 8 workers, one worker per core, and 2 cores share a power domain. Figures 13(a)(c) show the behaviors of worker 1 of Ray and Compare respectively, whereas Figures 13(b)(d) shows the behaviors of worker 2 of Ray and Compare respectively. We omit workers 3-8 to save space. The unit on the X-Axis is the execution time in 1ms intervals (1ms provides the best overview for explanation) and unit on the Y-Axis is the frequency in GHz.

The most striking difference when comparing AEQUITAS to HERMES is a significantly higher volume of frequency scaling by HERMES. Workers 1 and 2 of Compare spent nearly all of the first 5sec completing the first phase of their work in HERMES, which was completed in 1sec in AEQUITAS. We speculate that the root cause of this behavior is the contention raised by the two workers of Compare in HERMES while operating on the same power domain. Although each DVFS incurs only tens of microseconds of overhead, it quickly adds up. This is especially true when the two workers keep overriding each other’s frequency – a case of power domain “thrashing”. In AEQUITAS, workers take turns to dominate and thus reduce a significant portion of DVFS and eliminate “thrashing”.

In HERMES, workers are not only competing in the case of Co-Managed Power Domains but may also run into contentions with Co-Existing Applications. In the example presented, this can be observed at around the 6000th millisecond in Figure 13 where both applications are heavily competing to manage power.

The respective frequency scaling behaviors of HERMES and AEQUITAS also explain their different execution times. As shown in Figure 13, HERMES-based co-execution takes around 147 seconds for both to complete, whereas its AEQUITAS-based counterpart takes only 132 seconds. Energy consumption has a linear relationship with execution time. Readers might wonder why HERMES co-execution is not 5x longer than its AEQUITAS counterpart, considering that the first phase of Compare took nearly 5x longer for HERMES. This results from the fact that co-execution has extended phases of inactivity, which somewhat absorb the effects of slowdown and the fact that we extended HERMES to only
set frequency if it differs from current CPU frequency i.e. there is no need to honor DVFS if it is requesting the same frequency at which a core operates.

6.2 Scaling Counts
As previous subsection presented, the difference in the number of frequency scaling for HERMES and AEQUTAS plays an important role in understanding the effectiveness of AEQUTAS. This section presents a closer look at this matter in the presence of different benchmarking co-executions with the results shown in Figure 14. The more obvious observation is that as the number of co-executing benchmarks increases, the number of frequency scaling also increases, but the effects for HERMES and AEQUTAS are different. In AEQUTAS, the increase is mild, as the rotating domination can mask off many scaling attempts.

The effect for HERMES however is striking: the 4 Ray co-executions scale 8x more than single Ray execution. In both, HERMES (and AEQUTAS), a conditional check exists so that DVFS is not set if the newly requested frequency is the same as the current CPU operating frequency. As more benchmarks are involved, power domain “thrashing” becomes more prevalent in a HERMES co-execution; therefore, the likelihood of this check to succeed is significantly reduced. Furthermore, observe that heterogeneous co-executions fare better than homogeneous ones. As benchmarks demonstrate phased behaviors, homogeneous co-executions are likely to lead to more contentions because the involved applications are more likely to exhibit similar phased behaviors.

6.3 Underscaled/Overscaled Time
We also investigated the effect of time slicing on mitigating frequency scaling. Under the domination design of AEQUTAS, the non-dominating workers may suffer by operating on a non-desirable CPU frequency. However, this happens excessively to HERMES: when a worker eagerly scales the frequency to a level it desires in HERMES, other workers occupying the same power domain may operate at a non-desirable CPU frequency.

Figure 15 shows the normalized percentage of time Ray and Compare benchmarks – including all of their respective
workers—spend at a lower frequency than desired denoted as Under and at a higher frequency than desired Over.

Aequitas reduces both. Note that the data here are normalized, so the shortened execution time alone is not the factor as to why Aequitas fares better. In other words, not only does Aequitas reduce the number of frequency settings and shortens the execution time but it also helps reduce the percentage of time an application operates at an undesirable frequency. Energy consumption and frequency setting are strongly correlated. Figure 15 offers further evidence behind the energy savings retained by Aequitas.

7. RELATED WORK

Energy-aware coordination frameworks exist for improving the adaptability and quality of service (QoS) of energy-aware applications. A classic example is Odyssey [13]. Another early dynamic software management framework [12] considered the coordination of multiple applications, in a scenario where applications are both QoS-adaptable and energy-aware. OS efforts such as Ecosystem [46] and Cinder [35] take a bottom-up approach (from the view of the compute stack) to cooperative energy management. Energy and power are treated as first-class resources, and their provisioning is managed at the OS level. Overall, the goal of Aequitas is orthogonal with this line of research, in that execution is managed at the OS level. Overall, the goal of Aequitas is orthogonal with this line of research, in that execution is managed at the OS level. In other words, Aequitas is a top-down approach that adapts application/runtime-level energy management to existing OS and hardware.

Many systems have been developed to improve the efficiency of work-stealing algorithms. Recent examples include reducing sequential overhead [24], dynamic barrier overhead [23], deque management [1], and architectural support [31]. A more comprehensive review of earlier work was conducted by Hermes [34]. Among them, BWS [11] is the only work considering a time-shared multi-core environment. BWS focuses on improving system throughput and fairness, not energy efficiency.

The high-level goal of Aequitas is to help white-box energy optimization approaches adapt to time-shared operating systems and co-managed power domains. There is a large body of white-box approaches that the idea behind Aequitas may potentially impact on. First, it is known [43, 33] that different threads/tasks may form data or timing dependencies and hence can be scheduled intelligently for improved energy efficiency with DVFS (e.g., [47, 42, 45, 28]). The earlier systems along this line pre-dated the multi-threaded application era, and hence were developed in a (non-time-sliced) multi-tasking OS. In principle however, the core algorithms should be applicable to multi-threaded applications as well, especially with the help of an Aequitas-like system. Second, compiler-based energy optimization [20, 44, 16, 41, 3, 27] analyzes patterns of data/control flows and memory access patterns to enable judicious energy management. Third, a growing number of energy-aware programming language designs [2, 36, 8, 21, 48] harvest programmer knowledge to make application-specific energy management decisions. One common characteristic shared in this group of related work is energy optimization/management is performed on a per-application basis. Aequitas may serve as a concrete example to guide these systems to adapt to the reality of underlying OS and hardware.

8. CONCLUSION

This paper describes Aequitas, a practical system to coordinate energy management of multiple power-managing application runtimes. Through time-sliced power management with two-tiered domination, Aequitas achieves equalitarian coordination and efficiency in energy optimization among co-existing applications deployed on co-managed power domains. The minimalistic approach yields significant energy efficiency in a DVFS-enabled cooperating multi-environment with little overhead.

Acknowledgments.

We thank the anonymous reviewers for their useful comments. This project is sponsored by US NSF CCF-1526205, and CAREER Award CCF-1054515.

9. REFERENCES
