Fine-Grained Energy Efficiency Using Per-Core DVFS with an Adaptive Runtime System

Bilge Acun, Kavitha Chandrasekar, Laxmikant V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {acun2, kchndrs2, kale}@illinois.edu

***Abstract*—Dynamic voltage and frequency scaling (DVFS) is a well-known technique to reduce the power and/or energy consumption of various applications. While most processors provide chip-level DVFS, where the frequency and voltage of the cores in a chip can only be changed all together; core-level DVFS, where each core can be controlled independently, requires core-level voltage regulators in hardware and only is supported in production in Haswell generation among Intel processors. The finer grained control that per-core DVFS provides can lead to higher energy efficiency compared to chip-level DVFS especially for the unsynchronized, unstructured parallel applications when carefully applied.**

Ability to do per-core DVFS opens up new doors for different optimizations within runtime systems. We implement an intelligent energy efficient runtime module which uses a fine-grained function level per-core DVFS approach. Our module finds the energy-optimal frequency for each phase/function/kernel of the application over the first few iterations and applies the optimal frequency for each function. We test our implementation on Haswell processors and show that our algorithm enables 4% to 35% energy reduction over chip-level DVFS with as much as performance.

***Index Terms*—power, energy efficiency, DVFS, runtime systems**

I. Introduction

As the scale of the High Performance Computing (HPC) data centers keeps growing, power and energy efficient system design has became an important challenge. Much of the past research proposes Dynamic Voltage and Frequency Scaling (DVFS) and Running Average Power Limit (RAPL) [22] based solutions to optimize energy consumption [36], [31]. These solutions have often been done at chip-level, i.e. changing the whole chip's frequency or capping the whole chip's power, not the individual core frequencies or core power. The reason for that is the lack of core level voltage regulators in the commercial processors. For the first time, Intel Haswell generation processors introduced the support for per-core DVFS in production [20], [19]. Although this support has been discontinued on later generations Sky Lake and Kaby Lake, it is reported to return on Ice Lake generation [2]. The capability of doing per-core DVFS brings the premise of higher energy efficiency with finer-grained optimization.

Process variation is one of the major motivations for implementing core-level voltage regulators. The decreasing size of Complementary Metal-OxideSemiconductor (CMOS) transistors and lower voltage thresholds for energy efficient chip design are two major causes of manufacturing-related process variation [11]. This process variation causes differences in the power and temperature of the cores [6]. Therefore, adjusting the voltage at core-level can help getting the most performance out of the chip.

Another motivation for core-level DVFS is applications that execute code with different characteristics simultaneously on different cores. While some HPC applications are regular or structured with a uniform behavior, i.e. all cores or processes within a chip execute similar type of work, some HPC applications are irregular or unstructured with dynamic behavior. In such applications, at a given time cores within a processor might do different types of work and execute different types of functions. Moreover, each function/kernel/phase might have a different optimal frequency level. (Note that energy optimal frequency is not necessarily the fastest frequency, many examples of this are shown for HPC applications in the past [35], [7]). For example, it has been shown that MiniFE application have loops that are affected differently by the frequency levels. [39]. Some applications for performance reasons have dedicated I/O or communication threads that inherently have different behavior than the rest of the application. Yet, many commercial processors used in HPC systems do not have support for per-core voltage and frequency scaling. With chip-level DVFS, different kernels that happen simultaneously cannot run at their energy-optimal frequency levels.

Previous work in our research group has demonstrated utility of chip-level DVFS/RAPL based power optimization under the control of adaptive runtime systems like CHARM++ [36], [5]. In this paper, we develop a runtime method to realize the promise of core-level DVFS by doing fine-grained energy optimization. We develop a new runtime based energy optimization module in CHARM++ , that can learn the optimal frequency for each task or function over time and show how per-core DVFS enables finer-grained efficiency. Main contributions of this paper are:

- We provide an analysis of per-core DVFS capabilities in three different Intel and IBM processors (§ II).
- We identify use cases for per-core frequency and voltage regulation that motivates the need for implementing per-core voltage regulators in hardware (§ II).
- We implement a fine-grained runtime technique which provides automated function-level energy efficiency using the CHARM++ framework (§ III).
- We show that 4% to 35% better performance/reduction in energy can be obtained compared to per-chip frequency scaling using the micro-benchmarks we developed (§ IV-4).

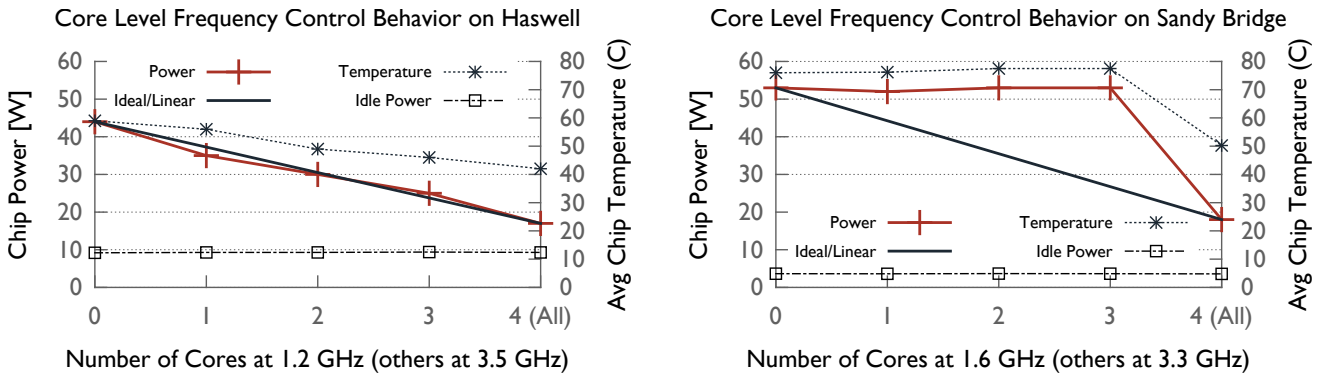

Fig. 1: Core level DVFS on Haswell architecture shows proportional/linear decrease in power when core frequencies are dropped one by one. On the other hand, since Sandy Bridge do not have per core voltage regulators, all core frequencies needs to be dropped together to for a reduction in power and temperature.

TABLE I: Platform hardware and software details

Processor	Intel Haswell	Intel Sandy Bridge	IBM POWER8
Model	Xeon®E5 -1620 v3	Xeon®E3 -1245	POWER8 8335-GTA
Cores	4		10
OS - Linux	Ubuntu v. 3.13.0		Red Hat Ent. 7.2
Turbo Speed	3.6 GHz	3.7 GHz	3.6 GHz
Max Non-Turbo Speed	3.5 GHz	3.3 GHz	3.5 GHz
Min Non-Turbo Speed	1.2 GHz	1.6 GHz	2.0 GHz
Per-Core Voltage Regulators(FIVR)	Yes	No	Yes, not in production

While there is prior research that shows the potential benefits of per-core DVFS in simulation environments [34], [24], including recently in graphics processors too [30]; to the best of our knowledge, we are not aware of any other work that does function level energy optimization through a runtime using production-level per-core DVFS capabilities (See related work in Section VI).

II. Motivation

In this section, first an analysis of the per-core DVFS support in the Haswell architecture is provided. Later, application use cases that can benefit from per-core DVFS are identified.

A. Per-Core Voltage and Frequency Scaling

For the first time among commercial Intel processors, Haswell generation introduced per-core voltage regulators [20]. Per-core voltage regulation, also called FIVR [12], give the premise of doing fine-grained power and energy control. In this section, we first show why per-core voltage regulators are important in doing core-level frequency scaling. We use an older Sandy Bridge generation processor which does not have FIVR to compare per-core DVFS behavior with the Haswell generation. The details of the processors that we use are given in Table I.

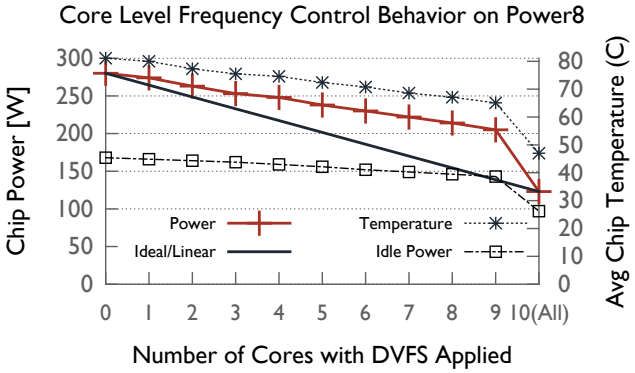

Fig. 2: Core level frequency scaling on IBM POWER8 processor.

Dynamic power of the CPU is proportional to the square of the CPU voltage and to the CPU frequency:

$$Power = C \times V^2 \times f$$

where C is capacitance, V is voltage, and f is frequency.

Voltage is a dominant factor in processor power and hence per-core frequency scaling without per-core voltage regulators is not effective in terms of performance-per-watt. Because, if the cores within the chip have different frequency levels, without per-core voltage regulators, the voltage level of the processor is determined by the highest frequency core.

Figure 1 compares per-core frequency scaling behavior of two processors; one with per-core voltage regulators (Intel Haswell) and one without (Intel Sandy Bridge). We use `cpufreq` kernel module to control the frequency of the cores. In this experiment, we first set the frequency of all cores to the highest level and then decrease the frequency level to minimum one core at a time. For Haswell processor, as we decrease frequency of more number of cores to minimum the chip power drops linearly. On the other hand, for the Sandy Bridge processor, the chip power does not change until all of the core frequencies are lowered together. The reason is that Sandy Bridge processor do not quite obey the per-core

`cpufreq` commands and executes all of the cores at the maximum level until all of the core-frequencies are reduced together.

Another observation from Figure 1 is that dynamic power of the Haswell chip constitutes 80% of the total chip power. Although this percentage is less than the 94% ratio in the Sandy Bridge generation, dynamic power is still a major amount that can be optimized via DVFS, whereas static power is not affected by the frequency of the cores.

Figure 2 shows the results of the same experiment (as Figure 1) on a 10 core IBM POWER8 processor. Despite having per-core voltage regulators in hardware of this processor, per-core DVFS is not supported in production use with `cpufreq` module. Although, unlike Sandy Bridge, POWER8 obeys per-core DVFS commands and changes the frequency core by core. As a result, the chip power gradually decreases as shown in the figure. However, it is still not a linear reduction as in the case of Haswell – there is still a bigger drop when all of the core-frequencies are changed all together. This shows per-core frequency scaling without per-core voltage regulation lacks effectiveness, i.e., it causes performance overhead without reducing the energy much.

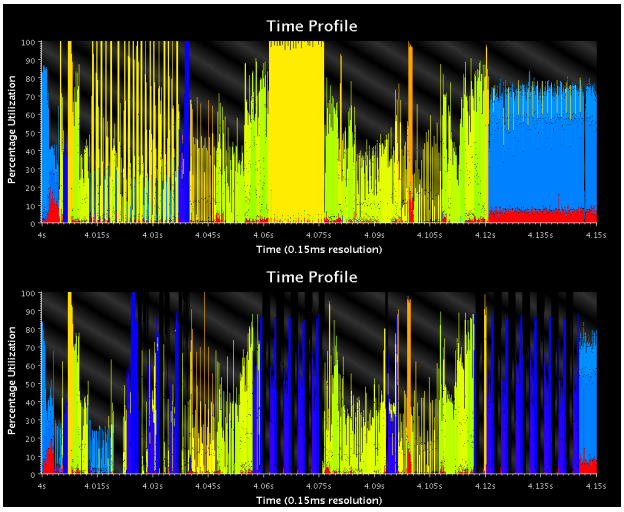

Fig. 3: Timeline of two processors running the OpenATOM benchmarks. Each color represents a CHARM++ entry method. Notice how different entry methods are executed on the two processes at the same time range.

B. Application Use Cases

Per-core DVFS would be beneficial for multiple different scenarios. We identify six categories of potential use cases that can benefit from per-core DVFS.

*1) **Applications with Different Kernel Types**:* Applications may have different kernels which have different optimal frequencies. Commonly, if a kernel is compute intensive, higher frequency levels lead to higher energy efficiency since the execution time is lower. On the other hand, for memory intensive kernel, higher frequency does not always lead to lower execution time. Therefore, energy optimal frequency levels can be lower. For example, *MiniFE* application has shown to have two different kernels that have different characteristics [39]. Moreover, those kernels do not necessarily execute at the same time. For example, *OpenATOM* is a quantum chemistry application which has many overlapping kernels and phases. Figure 3 shows the time profile graph of two processes mapped to two different cores running the benchmark. As each color represents a CHARM++ entry method, it can be seen that there are many phases where each process is executing a different type of function. Therefore, per-core DVFS can lead to higher energy efficiency for this type of applications.

*2) **Applications with Dedicated I/O Threads**:* Dedicated I/O threads are used in I/O intensive parallel applications to improve the performance by offloading the I/O work to dedicated threads. Many past works use this approach, including I/O frameworks [23], [26], fast asynchronous checkpoint/restart based fault tolerance mechanisms [16], machine-learning applications [18], high performance in-memory databases [10]. Energy-optimal frequency of the I/O threads can be different than rest of the threads in the application and controlling the frequency of the cores independently can lead to higher efficiency.

*3) **Applications with Dedicated Communication Threads**:* It has been shown that using dedicated communication threads to drive the network communication helps improve the performance of communication intensive applications especially at scale. MPI Endpoints [37] and SMP mode of CHARM++ [29] are two examples of this. MPI endpoints extension enables efficient multi-threaded communication by using dedicated cores that drive independent network communication. Although losing a core for communication might cause performance loss in a computation dominant application, at large-scale when the communication becomes the large fraction, having dedicated cores for communication improves performance [37]. CHARM++ implements a similar concept, called SMP mode. In the SMP mode, a logical node is formed by a custom number of threads and a dedicated communication thread to handle the communication between nodes. Commonly each physical node has one or more number of logical nodes. Can we control the frequency of the communication threads independently to have higher energy efficiency?

*4) **Applications Leaving Cores Idle**:* Parallel applications may leave idle cores for various reasons including, for performance reasons and for having specific core-count requirements. We will give three examples of this.

First, there are several applications that requires to be run on specific core counts, i.e. such as power of two number of cores, cubic number of cores. In such scenarios, some cores have to be left idle depending on the processor core count. For example, broadcast/reduction trees do not perform as well for numbers of process counts that are not a power of two [25]. Graph500 benchmark [1] only supports graphs with power-of-2 vertex counts and hence MPI version is required to be to run on power-of-2 number of cores. Similarly, LULESH [28] MPI version requires cubic number of ranks (hence cores) to map the 3D spatial domain. Whereas many supercomputing platforms has non-power-of-2 core counts per node such as

Summit supercomputer that has 42 cores per node [3].

Second, some applications, like NAMD [29], suffer from OS interference. To remove the interference, all OS processes and daemons are preferred to be bound to specific core, which is then excluded, i.e., not used, by the application. While some platforms provide this isolation feature as an optional parameter to the job scheduler/launcher, other platforms like Summit and Sierra enable this by default which leaves the application 42 cores out of the total 44 cores in the node [3].

Third, some applications that have high memory bandwidth requirements leave one or more cores idle in order to fit into the node or increase performance. PDES [32] is an example for the case where the application runs out of memory when doing large scale simulations and required to leave one or more cores per node idle.

Can we apply per-core DVFS, so that idle cores are run at lowest frequency, to save energy?

III. Runtime Guided Frequency Regulation

In this section, we first give background information on the runtime system that we use as a proof-of-concept for our work. Then, we explain our runtime guided frequency regulation approach.

A. Charm++ *Adaptive Runtime System*

Charm++ is a parallel programming framework used by many large-scale applications including NAMD for molecular dynamics, OpenATOM for quantum chemistry, ChaNGa for cosmology, Episimdemics for epidemic simulations and many others [4]. In Charm++ , the application data is decomposed into small task units (called *chares*) that communicate via asynchronous function calls (called *entry methods*). The runtime system (*RTS*) is responsible for placement and execution of the task units.

Chares are C++ objects that represent the data and task units in Charm++ . These objects can migrate from processor to processor by the runtime in order to create load balance. They can communicate with other objects via asynchronous function calls.

Entry Methods are asynchronous function calls on the chares. Chares can be located in a local or a remote processor, regardless of the location the runtime will deliver the function call as a message to its destination chare. Besides the application's entry methods, runtime itself also contains entry methods to do various tasks in the background such as communication, I/O, load balancing, tracing etc.

Charm++ RTS is responsible for the mapping of the objects to processors, sending the messages (entry method calls) to their destination chares, and executing the entry methods. Charm++ programmer needs to write an interface file to define the class types and the corresponding entry methods so that the runtime system can generate the corresponding structures. There is a natural division between the application phases.

Charm++ would be a great fit to make a proof-of-concept implementation of our ideas for two main reasons. First, programmer writes entry methods naturally in a way that different phases of the application are distinguished with different entry methods. This enables runtime to distinguish different types of work units automatically by simply controlling at each entry method-level. Second, the runtime has a transparent control over the full application from start to end. The approaches restricted to annotated loops or kernels do not give control over the whole application runtime which limits optimization scope and capabilities.

B. Fine-Grained Frequency Regulation in Runtime

Our approach fine-grained frequency regulation approach has three phases: 1) Statistics collection of power and performance for different frequency levels, 2) Calculating the optimal frequency based on the collected statistics, 3) Applying the optimal frequency on function basis in core-level. Next, we will explain each of the three steps in detail.

Our module can simply be enabled by building Charm++ with `--enable-energyOpt` flag and linking the application with `-module energyOpt`.

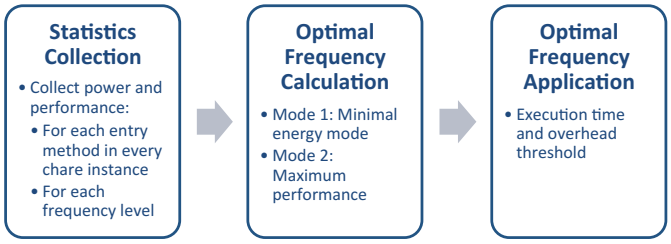

Fig. 4: Runtime control flow.

1) ***Statistics Collection****:* The first phase from the start of the application is the statistics collection phase. First, the runtime collects the power and performance characteristics of each entry method in the application under different frequency levels. The frequency levels are set by the runtime system as the application moves forward.

The entry method statistics are collected and stored per each instance of a chare element. Although every instance of a chare element have the same functions, different data size might cause different characteristics depending on the instance. Therefore, it is more accurate to collect the statistics per instance of chare elements. Particularly, the execution time and the energy consumption of each entry methods are collected under every supported frequency level.

There is, however, one obstacle in collecting power statistics of the functions. Despite the support of per-core DVFS, Haswell processors do not provide core-level power counters – only chip level power information is available through model-specific registers (MSRs). Therefore, a workaround is necessary. To be able to calculate the correct power information in core level, we execute the entry methods on the cores exclusively via locks during the statistics collection phase so that only one core is running at a time. An advantage of this approach is that it can capture core-to-core power or performance variations that might happen since profiling of the kernels are done on the same cores that are going to execute them after the profiling phase. A disadvantage of this approach

is that the measurements on single core may not represent the performance and power when other cores are active.

First, the static power of the other three cores needs to be subtracted from the total chip power. The static power of the three cores can simply be calculated as three quarters of the the idle power of the processor. Since there are four cores in the processor we use, exclusive entry methods execution causes a 4x slowdown in the application during this phase. This is a limitation of the processor and if there were core-level power counters, there would not be any need for a locking mechanism. Since the statistics collection phase constitutes a small fraction of the total application run, use of this exclusive execution method is still viable despite its overhead.

Second, we need to make sure the measurements we collect when kernels are running exclusively are correct when they run together with other cores. Figure 5 shows that for compute intensive benchmarks, like DGEMM, the optimal frequency calculated using one core is consistent with the results when other cores are active. However for memory intensive kernels, as more cores are running the kernel, the energy optimal frequency drops significantly since more cores are competing for the same memory bandwidth. To show this, we use MEMOPS kernel which consists of memory operations such as allocating/deallocating memory and copying data. Figure 6 shows the optimal frequency of MEMOPS kernel with varying allocated data sizes. From 128 MB to 1.5 GB data per process, the trend is more or less consistent: as more processes are active the optimal frequency drops. This shows that for memory intensive applications, what other cores are running affects the performance of the kernel, therefore exclusive measurements on a single core cannot represent the general case. Therefore, the count of memory operations (such as reads, writes, cache misses) needs to be collected for each kernel in addition to the execution time and the power consumption during the statistics collection phase. A method to prevent this problem could be using a power prediction model to predict the core-power consumption [13], [17]. However, such model can be error prone and it would increase the runtime algorithm complexity. The best way to overcome these limitations is to provide core-level power counters and perhaps this paper provides a motivation for hardware vendors to enable that support.

*2) **Optimal Frequency Calculation**:* After the statistics collection phase is done, optimal frequency is calculated for each entry method in each chare instance. We implement two options: the frequency that provides minimal energy (*MinE*) and the lowest frequency without sacrificing performance (*MaxP*). *MaxP* mode can also be used to find the lowest frequency with performance sacrifice of no more than a specified percentage (i.e, what is the lowest frequency that an application can use with a maximum of x% overhead?). We focus on evaluating *MinE* mode in this paper.

*3) **Optimal Frequency Application**:* After the optimal frequency levels are determined, the next phase is to apply the optimal frequency before the each method gets executed. There are some important issues that needs to be considered when applying DVFS (p-state change) on per function basis.

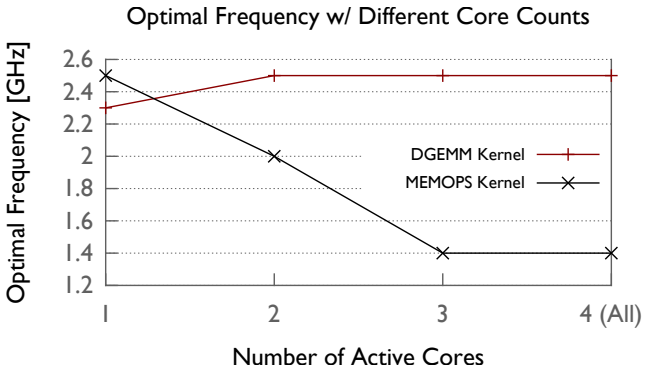

Fig. 5: Optimal frequency of DGEMM kernel remains more or less stable despite the number of active cores running the kernel, wheres optimal frequency of the MEMOPS kernel drops significantly as more cores are activated.

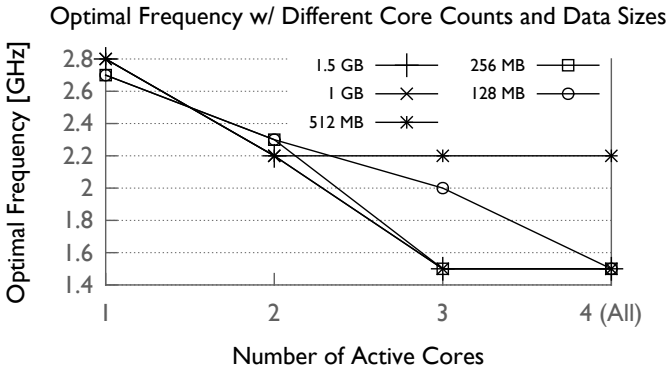

Fig. 6: Optimal frequency of the MEMOPS kernel depends more on the number of active cores than the data size.

There is a delay between sending p-state change request and the processor switching its frequency, this delay is called transition delay. This delay is important in making function level transition decisions because if the function duration is smaller than the delay, switching to a different level may not be useful. We have observed this delay can be up to around 500 µs. This confirms the earlier reported results on the transition delay [19]. Although the actual p-state transition may take much shorter than 500 µs, the p-state transition requests are not executed immediately, but in 500 µs periods in Haswell processors unlike previous generations (including Haswell-HE) [19]. Note that during the 500 µs, the processor is not blocked at the function call, it simply runs at the old frequency.

Another important concern is the overhead of applying DVFS frequently. We have measured the overhead to be between 2 to 5 µs. The overhead is mainly caused by writing the new frequency level to the appropriate system file (two file writes for the two SMT threads). Although this is not a significant overhead, our algorithm takes this effect into account when making frequency scaling decisions and and tries to minimize the overhead. If predicted function duration based on the historical data is less than a certain threshold, the runtime does not apply frequency change for that function.

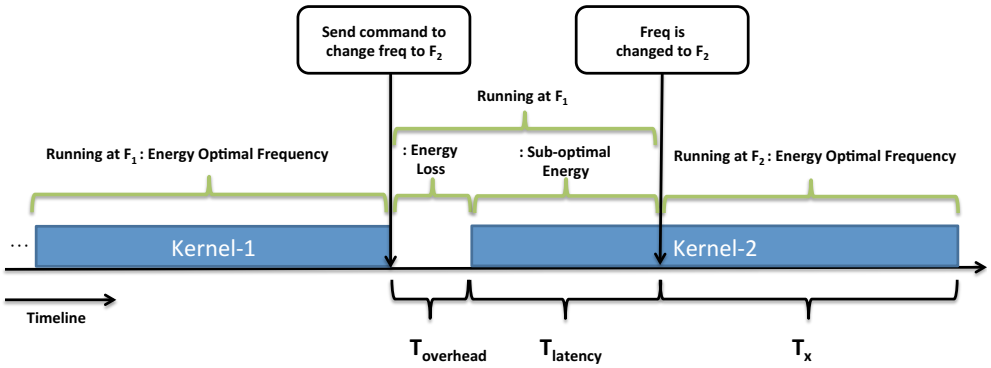

Fig. 7: Timeline of a core executing two kernels executing one after the other where the runtime applies optimal frequency for each kernel.

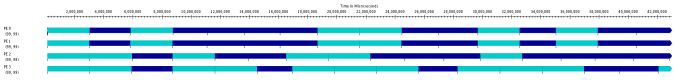

Fig. 8: Timeline of the synthetic benchmark having two kernels (represented by light blue and dark blue colors) randomly overlapping. Each row represent a process/core and x axis shows the time.

Figure 7 shows the timeline where two kernels (Kernel-1 and Kernel-2) are executing one after another in a core. Let's define:
$F1$: energy optimal frequency for Kernel-1,
$F2$: energy optimal Frequency for Kernel-2.

Applying the optimal frequency for Kernel-2 after the execution of Kernel-1 would be beneficial only if the following condition is satisfied:

$$E_{loss} + E_{F1\_portion} + E_{F2\_portion} < E_{F1}$$

E_{F1}: Energy consumption when whole Kernel-2 runs at *F1*
E_{loss}: Energy loss occurred during DVFS overhead
$E_{F1\_portion}$: Energy consumption when Kernel-2 runs at *F1* during the transition latency phase
$E_{F2\_portion}$: Energy consumption when a portion of Kernel-2 runs at *F2* after the transition is complete

Expanding the above formula in terms of power and execution time, we get:

$$T_{overhead} \times P_{F1} + T_{latency} \times P_{F1} + T_X \times P_{F2} < T_1 \times P_{F1}$$

E_{F1}: Energy consumption when all portion of Kernel-2 runs at *F1*
P_{F1}: Power consumption when Kernel-2 runs at *F1*
P_{F2}: Power consumption when Kernel-2 runs at *F2*
$T_{overhead}$: Constant, 5 microseconds
$T_{latency}$: Constant, 500 microseconds
T_X: Duration of the target kernel

Value of T_X depends largely on $F1$ and $F2$. In Figure 9, we show experimental results with a compute kernel where $F2$ is 2.3 GHz and $F1$ is labeled as transition frequency on the x-axis. The first observation is that the further away the transition latency is from 2.3 GHz, the more energy reduction is expected to happen. Second observation is that the smaller the kernel duration is, the less energy reduction happens because of the transition latency and overhead as described earlier in this section. Kernel duration label on each line in the plot represents the duration when run on the highest frequency. The duration of the kernel is adjusted simply by changing the loop counter to set how many times the kernel is executed.

We form a look-up table similar to the data in the Figure 9 for each application during the runtime and use this information in order to decide if frequency change for a particular kernel can lead to energy reduction.

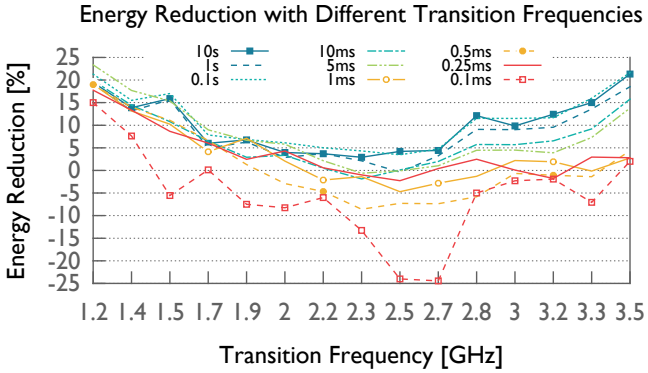

Fig. 9: Plot shows how much energy can be reduced if a kernel that has an energy optimal frequency of 2.3 GHz is transitioned from different frequency levels. Different lines represent different kernel durations.

IV. Experimental Results

In this section, we evaluate the effectiveness of our runtime module, mainly comparing per-core DVFS with per-chip DVFS using all of the application scenarios we mention in Section II.

*1) **Applications with Different Kernel Types**:* To evaluate this scenario, we implement a micro-benchmark, CompMem, that has two different kernels that are randomly called one after the other in every process. The goal of this benchmark is

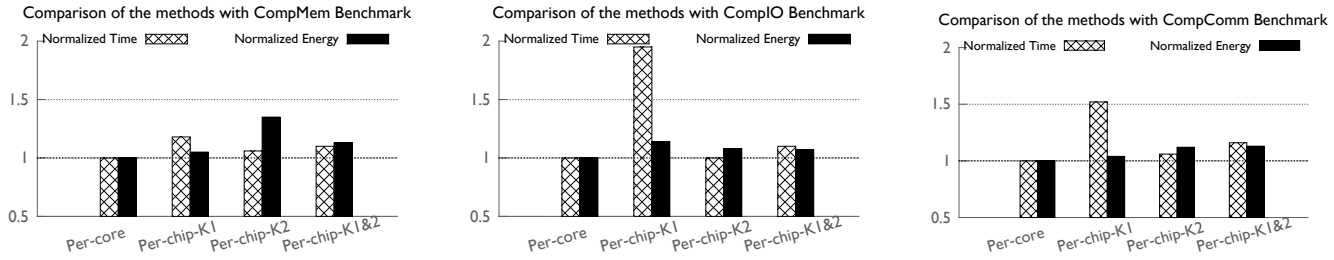

Fig. 10: *Per-core:* Uses energy optimal frequency for each kernel in core level. *Per-chip-K1:* Uses per-chip DVFS with optimal frequency of kernel-1, *Per-chip-K2:* Uses per-chip DVFS with optimal frequency of kernel-2, *Per-chip-AVG:* Uses per-chip DVFS with weighted average of optimal frequencies of kernel 1 and 2 of the corresponding benchmarks.

to have processes execute functions that have different characteristics at a given time. One kernel (*K1*) is a memory intensive kernel, MEMOPS, that consists of allocating memory, copying data into allocated memory via *memcpy*, and deallocating the memory. We use an array of size approximately 500MB per process. The second kernel (*K2*) is a compute intensive one (i.e., a DGEMM kernel with a matrix size that fits in cache). Figure 8 illustrates the timeline of this benchmark.

We compare the execution time and energy measurements of our fine-grained runtime based solution (labeled as *Per-core*) with other chip level solutions (labeled as *Per-chip-**) in Figure 10. *Per-core* DVFS always gives the minimum energy and minimum execution time. *Per-chip-K1* and *Per-chip-K2* optimizes only for kernel 1 and 2 respectively, therefore they have high execution time overhead or high energy. *Per-chip-K1&2* uses the mid-point of *Per-chip-K1* and *Per-chip-K2*, therefore creates a balance between the high execution time and the high energy of those methods, however it still is not better than *Per-core*. Overall, *Per-core* provides 5%, higher energy efficiency than *Per-chip-K1*, 35% higher than *Per-chip-K2* and 13% higher than *Per-chip-K1&2*.

We also evaluated the OpenAtom quantum chemistry benchmark. As we show the timeline of OpenAtom earlier in Figure 3, this application has different kernels that execute simultaneously on different cores. However our runtime mechanism did not lead to higher energy efficiency for OpenAtom compared to per-chip DVFS methods. The reason is that the duration of the kernels in OpenAtom were short and the optimal frequency of different kernels were not very different than each other. We discuss potential methods that can improve the effectiveness of our method in these types of scenarios further in the discussion Section (§V).

2) ***Applications with Dedicated I/O Threads****:* To evaluate this use case, we implement another micro-benchmark, CompIO. CompIO benchmark has one dedicated thread per processor doing IO operations such as file reads and writes to the local disk. The other threads are running a compute intensive kernel. As shown in Figure 10, our per-core DVFS mechanism provides higher efficiency compared to per-chip DVFS. Overall, *Per-core* provides 14%, higher energy efficiency than *Per-chip-K1*, 8% higher than *Per-chip-K2* and 10% higher than *Per-chip-K1&2*.

3) ***Applications with Dedicated Communication Threads****:* CompComm is the third micro-benchmark we use to demonstrate the usage of dedication communication threads. CompComm benchmark has one dedicated thread in the processor responsible for external communication (i.e., sending and receiving messages to and from other processors). We use CHARM++ SMP version to enable the usage of communication thread. As shown in Figure 10, our per-core DVFS mechanism provides higher efficiency compared to per-chip DVFS in this scenarios as well. Overall, *Per-core* provides 4%, higher energy efficiency than *Per-chip-K1*, 12% higher than *Per-chip-K2* and 13% higher than *Per-chip-K1&2*.

4) ***Applications Leaving Cores Idle****:* There might be several reasons for applications to leave idle cores, for performance reasons and for having specific core-count requirements. We focus on the former one.

For HPC applications with high memory bandwidth requirements we observe that in some cases running the application with all cores on each node need not necessarily be the best performing configuration. That is, utilizing only some number of cores on each node for a multi-node application can yield better performance. To motivate leaving cores idle and possibly applying per-core DVFS on the idle cores, we show performance improvements for CHARM++ Stencil3D application for input size of 23GB and 18GB.

We made runs for Stencil3D for 30 time steps using 12 cores (11 worker threads and 1 communication thread) on a single node on Campus Cluster at UIUC. The node configuration is Intel Xeon E5-2680 v3 with 2 sockets each with 12 cores at 2.5 GHz and 32GB DRAM. The application is profiled on a subset of core counts to select the best performing core count dynamically. We observe that the application performs best at 6-8 cores instead of using all available 11 cores. The results are shown in Figure 11. Using less cores leads to lower execution time, as well as 45% energy reduction. Can we get additional benefits by lowering the frequency of the unused cores by applying per-core DVFS?

Although this case may seem to be a good use case for per-core DVFS, in fact, if the idle core power is already optimized to be its lowest, there may not be additional room for energy reduction by applying per-core DVFS. As we show earlier in Figure 1, for Intel processors idle power is not effected by the frequency. On the other hand for POWER8 processors,

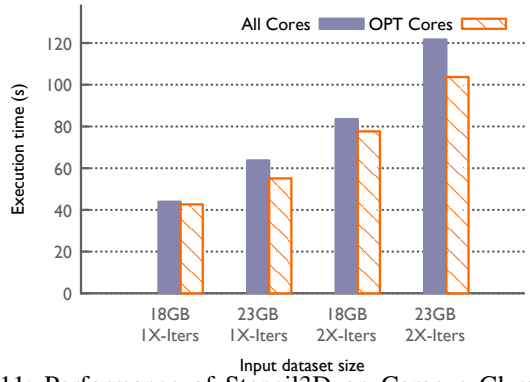

Fig. 11: Performance of Stencil3D on Campus Cluster with 23GB and 18GB dataset size with 1 and 2 iterations (to mimic gauss-seidel).

frequency effects idle power as shown in Figure 2, i.e., around 5 W per core (about 3%) can be reduced in application that leave cores idle if per-core DVFS is introduced. In the Stencil3D example, since 5 cores out of 11 cores are idle, per-core DVFS leads to additional 15% energy reduction.

V. Discussion and Limitations

We show clear benefits of using per-core DVFS using the micro-benchmarks we developed. However, we encountered several challenges when applying our method into production HPC applications. We discuss these here.

First challenge is the short duration of the kernels. 500 µs frequency transition latency and 5 µs overhead is quite high and some kernels might be mostly over by the time core is operating at their optimal frequency. If the transition latency and overhead reduces in the future generation processors, it would help our method to become more effective. Another solution to overcome this challenge is to combine tasks with the same optimal frequency altogether. A task queue look-up strategy in the runtime can identify and execute such kernels back-to-back to reduce the frequency transition overhead. Since Charm++ is an asynchronous programming model, the tasks in the queue can be executed in any order. The limitation we found in this approach is that the task queue may not contain enough kernels to combine at a given time. Another approach to address this can be using a machine learning approach, as used in some related work [21], to decide the optimal frequency allocation for each core in a more comprehensive manner.

Second challenge is that there are no core-level power measurements supported by the hardware. Therefore our optimal frequency selection mechanism might be less accurate than it can be. Support for core-level power measurements from the hardware would be helpful for the assessment of core-level DVFS.

Finally, our method does not support hyper-threading since hyper-threads can share the same physical core, i.e. execute instructions from different threads in the same core. If the hyper-threads within the same physical core is executing different application kernels or functions, there is no feasible way to change their frequency for each of threads separately.

VI. Related Work

Many of the past research uses chip-level DVFS to do energy or power optimizations [38], [14], [15], [40]. Sarood et. al. proposes a thermal aware load balancer [36], [31] which uses chip-level DVFS tecnique to restrain the temperature of the processors. Padoin et al. extends this approach and proposes a load balancer which utilizes per-core DFVS [33]. However this work is not done in a hardware that supports per-core DVFS, instead it simulates the environment of a per-core DVFS by placing only one process on each chip.

Another chip-level adaptive frequency selection approach has been proposed for GPU and CPU kernels [8]. This approach can select a kernel to run on GPU or CPU and at which frequency level based on pareto optimality. One drawback of this work is that it works at individual kernel level and assumes all threads within the processor are executing the same kernel, and does not do any optimization in between the kernels. Our approach does optimization for whole application and in a transparent way taking into account frequency switching latency and overheads.

Lim et al. proposes using DVFS in communication phases of MPI applications, such as `MPI_Send`, `MPI_Recv` etc., to reduce the energy consumption [27]. Their approach, like ours, is implemented within MPI and transparent to the application. However, their approach is also inherently limited to communication phases within MPI. On the other hand, our scope is the whole application and can optimize for all phases and kernels of the application. Bhalachandra et al. uses a similar approach for MPI; when there is slack before an `MPI_Barrier`, it applies DVFS to balance the arrival time of the processes to the barrier [9]. The same drawbacks of Lim et al.'s paper applies to this one as well.

VII. Conclusion

This paper proposes a fine grained runtime approach to fully optimize the energy efficiency of applications at function level considering function to function variations within the applications. We show how per-core DVFS support from the hardware can lead to higher energy efficiency for various use cases including benchmarks with different kernel characteristics, communication threads, I/O threads and idle cores.

We also discuss the limitations we encountered while applying per-core DVFS to HPC applications. Lack of core-level power measurement support from the hardware is one of the major limitations. Another limitation is the relatively high frequency transition latency which makes it harder to make fine-grained optimization. Having core-level power counters and lower transition latency would make our method more practical and effective for HPC workloads. A future direction of this work is to understand how local optimal frequency selection decisions effects the global application performance when there is work imbalance between the nodes and how to combine our method with a frequency aware load balancer.

REFERENCES

[1] Graph 500 benchmark. http://www.graph500.org/.

[2] Intel to bring back fivr after skylake and kaby lake cpus. https://www.overclock3d.net/news/cpu_mainboard/intel_to_bring_back_fivr_after_skylake_and_kaby_lake_cpus/1.

[3] SUMMIT, Oak Ridge National Laboratory. https://www.olcf.ornl.gov/summit/.

[4] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. SC, 2014.

[5] B. Acun, A. Langer, E. Meneses, H. Menon, O. Sarood, E. Totoni, and L. V. Kalé. Power, reliability, and performance: One system to rule them all. *Computer*, 49(10):30–37, 2016.

[6] B. Acun, P. Miller, and L. V. Kale. Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

[7] B. Austin and N. J. Wright. Measurement and interpretation of microbenchmark and application energy use on the cray xc30. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 51–59. IEEE Press, 2014.

[8] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. De Supinski. Adaptive configuration selection for power-constrained heterogeneous systems. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 371–380. IEEE, 2014.

[9] S. Bhalachandra, A. Porterfield, S. Olivier, and J. Prins. An adaptive core-specific runtime for energy efficiency. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2017.

[10] J. Bomfim and R. Rothstein. In-memory database for high performance, parallel transaction processing, July 12 2002. US Patent App. 10/193,672.

[11] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*, pages 338–342. IEEE, 2003.

[12] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill. Fivrfully integrated voltage regulators on 4th generation intel® core socs. In *Applied Power Electronics Conference and Exposition (APEC), 2014 Twenty-Ninth Annual IEEE*, pages 432–439. IEEE, 2014.

[13] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao. Performance and power modeling in a multi-programmed multi-core environment. In *Proceedings of the 47th Design Automation Conference*, pages 813–818. ACM, 2010.

[14] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[15] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. Coscale: Coordinating cpu and memory system dvfs in server systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 143–154. IEEE Computer Society, 2012.

[16] J. Fu, R. Latham, M. Min, and C. D. Carothers. I/o threads to reduce checkpoint blocking for an electromagnetics solver on blue gene/p and cray xk6. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2012.

[17] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *Green Computing Conference, 2010 International*, pages 135–146. IEEE, 2010.

[18] M. Grossman, M. Breternitz, and V. Sarkar. Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):762–775, 2016.

[19] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. 2015.

[20] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014.

[21] C. Imes, S. Hofmeyr, and H. Hoffmann. Energy-efficient application resource scheduling using machine learning classifiers. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 45:1–45:11, New York, NY, USA, 2018. ACM.

[22] Intel. Intel-64 and IA-32 Architectures Software Developer's Manual , Volume 3A and 3B: System Programming Guide, 2011.

[23] F. Isaila, J. Blas, J. Carretero, R. Latham, and R. Ross. Design and evaluation of multiple-level data staging for blue gene systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):946–959, 2011.

[24] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture*, pages 347–358. IEEE Computer Society, 2006.

[25] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. *ACM Sigplan Notices*, 34(8):131–140, 1999.

[26] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom. Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–11. IEEE, 2015.

[27] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC 2006 conference, proceedings of the ACM/IEEE*, pages 14–14. IEEE, 2006.

[28] Lulesh. http://computation.llnl.gov/casc/ShockHydro/.

[29] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[30] P. Meinerzhagen, C. Tokunaga, A. Malavasi, V. Vaidya, A. Mendon, D. Mathaikutty, J. Kulkarni, C. Augustine, M. Cho, S. Kim, et al. An energy-efficient graphics processor featuring fine-grain dvfs with integrated voltage regulators, execution-unit turbo, and retentive sleep in 14nm tri-gate cmos. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 38–40. IEEE, 2018.

[31] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé. Thermal aware automated load balancing for hpc applications. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[32] E. Mikida, N. Jain, L. Kale, E. Gonsiorowski, C. D. Carothers, P. D. Barnes Jr, and D. Jefferson. Towards pdes in a message-driven paradigm: A preliminary case study using charm++. In *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, pages 99–110. ACM, 2016.

[33] E. L. Padoin, M. Castro, L. L. Pilla, P. O. Navaux, and J.-F. Méhaut. Saving energy by exploiting residual imbalances on iterative applications. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE, 2014.

[34] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM, 2009.

[35] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya. Some Observations on Optimal Frequency Selection in DVFS-based Energy Consumption Minimization. *Journal of Parallel and Distributed Computing*, 71(8):1154–1164, 2011.

[36] O. Sarood, P. Miller, E. Totoni, and L. V. Kale. 'Cool' Load Balancing for High Performance Computing Data Centers. In *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[37] S. Sridharan, J. Dinan, and D. D. Kalamkar. Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498. IEEE Press, 2014.

[38] L. Wang, G. Von Laszewski, J. Dayal, and F. Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368–377. IEEE Computer Society, 2010.

[39] W. Wang and E. A. Leon. Evaluating dvfs and concurrency throttling on ibms power8 architecture.

[40] C.-M. Wu, R.-S. Chang, and H.-Y. Chan. A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters. *Future Generation Computer Systems*, 37:141–147, 2014.

APPENDIX

The code used in this paper is integrated into the CHARM++ repository at the `bilge/energyOpt` branch and can be found in the following url:
https://charm.cs.illinois.edu/gerrit/gitweb?p=charm.git;a=shortlog;h=refs/heads/bilge/energyopt.
The benchmarks used in the paper are also located in the same branch under `examples/charm++/per_core_scaling_benchmarks` folder.