Exploiting Variability for Energy Optimization of Parallel Programs

Wim Lavrijsen, Costin Iancu, Wibe de Jong
Lawrence Berkeley National Laboratory
{wlavrijsen, cciancu, wadejong}@lbl.gov

Xin Chen, Karsten Schwan
Georgia Institute of Technology
{xchen384, schwan}@gatech.edu

In this paper we present optimizations that use DVFS mechanisms to reduce the total energy usage in scientific applications. Our main insight is that noise is intrinsic to large scale parallel executions and it appears whenever shared resources are contended. The presence of noise allows us to identify and manipulate any program regions amenable to DVFS. When compared to previous energy optimizations that make per core decisions using predictions of the running time, our scheme uses a qualitative approach to recognize the signature of executions amenable to DVFS. By recognizing the "shape of variability" we can optimize codes with highly dynamic behavior, which pose challenges to all existing DVFS techniques. We validate our approach using offline and online analyses for one-sided and two-sided communication paradigms. We have applied our methods to NWChem, and we show best case improvements in energy use of 12% at no loss in performance when using online optimizations running on 720 Haswell cores with one-sided communication. With NWChem on MPI two-sided and offline analysis, capturing the initialization, we find energy savings of up to 20%, with less than 1% performance cost.

1. Introduction

Optimizations using Dynamic Voltage and Frequency Scaling (DVFS) have been shown [16, 17, 19, 21, 30, 34] to reduce energy usage in HPC workloads. A rather impressive amount of work has focused on developing energy optimizations for MPI codes, with the main purpose of saving energy without affecting performance. As we continue to scale systems up and out towards Exascale performance, power becomes an important system design constraint and thus incentives grow to deploy these optimizations in production.

EuroSys '16, April 18 - 21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901329

This research was started as an effort to develop global control strategies and policies for large scale HPC systems. We were interested in developing *lightweight and practical* methods that can handle highly optimized codes that use modern programming practices, running on production systems with the latest hardware available.

Most existing techniques [17, 20, 21, 30] exploit the idea that the presence of *slack*, occurring when tasks wait at synchronization points, indicates that DVFS can be applied on those cores waiting for external events. Our survey of these state-of-the-art techniques indicates that: 1) results are demonstrated on codes that use static domain decomposition and exhibit static load imbalance; 2) decisions are made based on predictions of program running time; and 3) DVFS is always controlled per core. The current dogma states that dynamic load balancing and runtime adaptation techniques are required for performance on very large scale systems. Thus we are interested in developing efficient DVFS techniques for this class of codes. Furthermore, due to hardware or system idiosyncrasy, per core control may not be available or it has has limited impact when shared resources will not scale down unless all cores scale down. Thus, we were interested in coarser grained DVFS control; this functionality may also be required when the HPC community moves from flat Single Program Multiple Data (SPMD) parallelism, e.g. MPI, towards hierarchical and hybrid parallelism.

To handle dynamic program behavior, in this paper we put forth an alternative to the existing tenet that slack is predictable. We argue that making quantitative predictions on timings of individual processes in dynamic codes (parallelism or load balancing) is challenging because of random variability. But, one can use a qualitative approach and make use of variability itself. Our insight, and the main paper contribution, is that noise is intrinsic to large scale parallel executions, that it appears whenever shared resources are contended and that we can recognize its signature whenever code amenable to DVFS executes. As a signature we use the combination of the dispersion (measured as sample standard deviation) and skewness of task duration timings between two task group synchronization points. The DVFS amenable

regions are "dominated" by any of: 1) blocking or nonblocking one-sided or two-sided communication; 2) file I/O; and 3) DRAM bandwidth limited execution. In a production system, the first two are also affected by external events, outside of control available within one application.

We have developed single-pass online optimizations using context sensitive classification, as well as multi-pass offline approaches. Note that input and concurrency independent online analyses [21, 30] are required in practice for scalability and the long term success of DVFS optimizations.

To provide coarse grained DVFS, our algorithms were developed to select a global system frequency assignment for codes using either two- or one-sided communication. To handle the high latency of DVFS control on production systems we use clustering algorithms after candidate classification. We present extensions to previous work [21], mandatory for efficacy on modern hardware.

As a case study we have chosen an application that poses challenges to existing DVFS control techniques that use quantitative predictions of execution time. NWChem [33] is a framework for complex computational chemistry that provides a range of modules covering different theoretical models and computational approaches. NWChem is written using non-blocking *one-sided* communication (RDMA) for overlap and communication latency hiding, augmented with *dynamic load balancing* mechanisms. Ours is the first study able to handle this class of codes. To provide a comparison with state-of-the-art MPI centric techniques, we use UMT2K [32], an unstructured mesh radiation transport code and a common benchmark.

We validate results on three architectures: circa 2012 AMD A10-5800K and Intel Ivy Bridge deployed in current production systems, and 2015 Intel Haswell to be installed in the new generation of very large scale production [1, 4] systems. The combination of qualitative modeling with clustering methods provides us with a programming model and runtime independent approach. The variability criteria can handle both one- and two-sided communication paradigms, while timing predictions fail for one-sided or are less efficient for two-sided. Clustering is required in most cases for improved performance, but needs to be done with care. The importance of recognizing variability increases with scale.

With the online algorithm, we observe energy savings as high as 12% with a small speed up due to reduced congestion for one-sided communication at high concurrency. For two-sided communication we observe energy savings as high as 16% with negligible slowdown. The offline algorithm is able to double the energy savings for one-sided communication. Surprisingly, this is not due to its ability to use the optimal frequency for any region cluster in the program, but because it captures the one-off initialization stages, which are usually dominated by disk access. In all cases, savings are obtained where structural inefficiencies are causing contention, without the CPU going idle enough for long enough; or where the CPU itself is a driver of the contention.

We believe that hardware-driven scaling will soon usurp most opportunities for user-level DVFS within a node, but coarse-grained scaling may still have a role. Global assignments of DVFS settings have more benevolent statistics than per-core or per-node assignments. When a per-core assignment is correct, the gain is small (and is getting smaller in manycores); when it is wrong, the penalty is huge (and increasing with scale). They are therefore made conservatively, and favor long regions that allow for accurate measurements while covering enough of the program execution time to be worth it. That, however, reduces opportunity and increases the penalty for being wrong. Conversely, the benefits obtained from a correct global assignment are very similar to the penalty for being wrong, regardless of scale. Scaling can be applied more aggressively and on shorter regions, further reducing the cost of being wrong and enabling recovery, after learning through measuring, of bad decisions.

2. Background and Motivation

Slack is the most often used measure to identify DVFS opportunities in scientific codes and it is commonly defined as time spent blocked or waiting in communication calls: waiting tasks can be slowed down. As MPI two-sided has been dominant, most of the existing approaches [16, 17, 19, 21, 30] are tailored for its semantics.

Existing MPI energy optimizations try to construct a critical path through the program execution and minimize slack: ranks on the critical path (slowest ranks) need to run fast, while all others can be (down) scaled to "arrive just in time." At the heart of these approaches is the ability to predict the running time on each rank between two synchronization operations, either point to point (`MPI_Send/MPI_Recv`) or collective (e.g. `MPI_Barrier`). Initial studies [28] used offline, trace based analyses that often solve a global optimization problem. For generality and scalability, modern approaches use online analyses. Rountree et al [30] present Adagio, which uses context sensitive techniques to predict the critical path at runtime and minimize slack for each MPI communication call in its context.

We do embrace the notion that demonstrating successful online analyses is mandatory for the future adoption of DVFS techniques, either in hardware or software. Our conjecture when starting this work was that emerging programming models and optimization techniques together with modern hardware architecture all work against techniques using quantitative prediction of execution times at the granularity required for successful DVFS.

Our survey of the techniques proposed in [16, 19, 21, 28, 30] shows that they were evaluated only on codes with *static domain decomposition* and static load balancing: each rank in each iteration works on the same data partition, making execution time *predictable*. In newer codes that use *dynamic*

load balancing, each rank in each iteration can process different data partitions, leading to more *unpredictable* duration per rank per iteration. To our knowledge, DVFS optimizations for dynamically load balanced codes at scale have not yet been demonstrated.

All codes[1] surveyed in literature use blocking two-sided communication. Non-blocking communication and overlap add another dimension of noise and unpredictable [14] behavior. To our knowledge, energy optimizations on applications using one-sided communication have not yet been demonstrated. On these codes, the existing state-of-the-art of scaling fails: dynamic load balancing explicitly attempts to remove slack and makes any slack that is left-over unpredictable. Furthermore, one-sided communication removes a great many (implicit) synchronization points.

Finally, modern CPU hardware is tuned for energy efficiency and employs aggressive dynamic control. It begs the question how much software techniques (perhaps global) can still improve on existing hardware mechanisms in practice. From the hardware perspective many approaches [16, 19, 21, 27, 28, 30] have been validated assuming availability of per core DVFS, some even using a single core in configurations with as many as eight cores per socket. Modern hardware either offers only per socket control (e.g. Intel Ivy Bridge), or per-core control interferes with the execution on cores sharing resources within the socket (e.g. hyperthreads, rings on Intel Haswell[13], caches). Other system components (e.g. the Mellanox FDR InfiniBand[2] driver) may require a single frequency assignment per socket or node. Another hardware trend is the presence of "Turbo-boost", which allows individual cores to run at a higher frequency depending on system load. Not only does this reduce the amount of slack in a natural way, it also makes schedules harder to control, as the initial drop from Turbo-boosted frequencies to the first user-selectable frequency tends to be huge (of the order of 20%-35%, depending on load). An algorithm that relies on a detailed per-core schedule therefore needs to find "imbalance" of similar magnitude before it can apply DVFS. For practicality, extensions to existing techniques to provide coarser grained control over groups of cores are required. And these are even more desirable for whole system optimization goals.

2.1 Predicting Timings for Dynamic Behavior

We first examine what is predictable in an application that uses dynamic load balancing. NWChem [33] is a framework for large scale computational chemistry codes, written to use as a communication transport either MPI two-sided, or one-sided communication with MPI 3.0, ARMCI [24], ComEx [8] or GASNet [11]; see Section 5.3 for more details.

[1] Bar one trivial usage of `MPI_Isend/MPI_Irecv` in one code.

[2] During this work we have uncovered performance and functionality bugs in the current generation of Mellanox drivers.

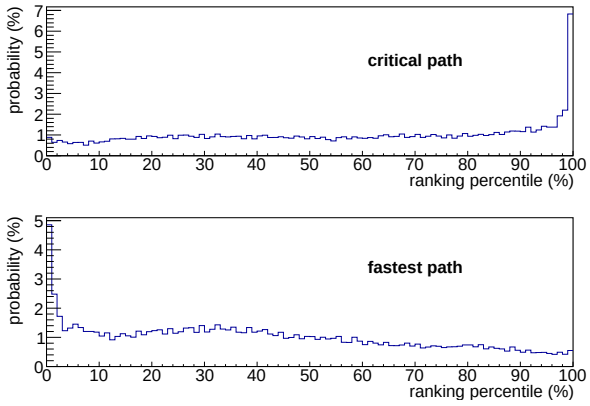

Figure 1: *Quality of prediction of the critical path based on calling context and process in NWChem, for a run of 1024 processes. Shown is the ranking of the critical (top) and fastest (bottom) process, in the subsequent re-occurrences of tasks. The probability of the critical path remaining critical, and of the fastest remaining so, is less than 10%.*

Given that one-sided communication (`Put` or `Get`) does not carry any inter-task synchronization semantics like `MPI_Send/MPI_Recv` pairs do, we have started by trying to predict slack between two barrier operations. We configure NWChem to use one-sided communication and collect the durations per task between barriers, taken as the difference between the exit time of one barrier and the entrance time of the next. The slack associated with a task is the difference in execution time when compared with the slowest task.

For each region (i.e. barrier pair), we label each task with its index in the sorted array of all timings for all tasks. We collect these rankings per process and calling context (described in Section 4). We select only those regions that are long enough (at least $300\mu s$[3]) and have a minimum of 5% difference in duration between the fastest and slowest process, compared to their average (i.e. there is a minimum 5% slack). This selects regions for which DVFS is potentially practical and beneficial.

We select those calling contexts that occur at least 10 times, providing enough repetition to allow both predicting/learning and scaling to occur. From these, we take the slowest process, i.e. the critical path, at the first occurrence of each, and plot their "ranking", expressed as a percentile from fastest to slowest, at each subsequent re-occurrence of the same context. The results are in the top of Figure 1, for a run with $1,024$ processes. If the time duration of the first occurrence of a calling context can be used to predict the durations on subsequent calls, then there should be a sharp peak at 100%, i.e. the critical path should remain close to critical. What we observe, however, is an almost flat distribution, that moderately tapers off towards zero, with less than 10% of the critical path "predictions" being on the mark.

[3] About 3× the latency of the system specific DVFS control.

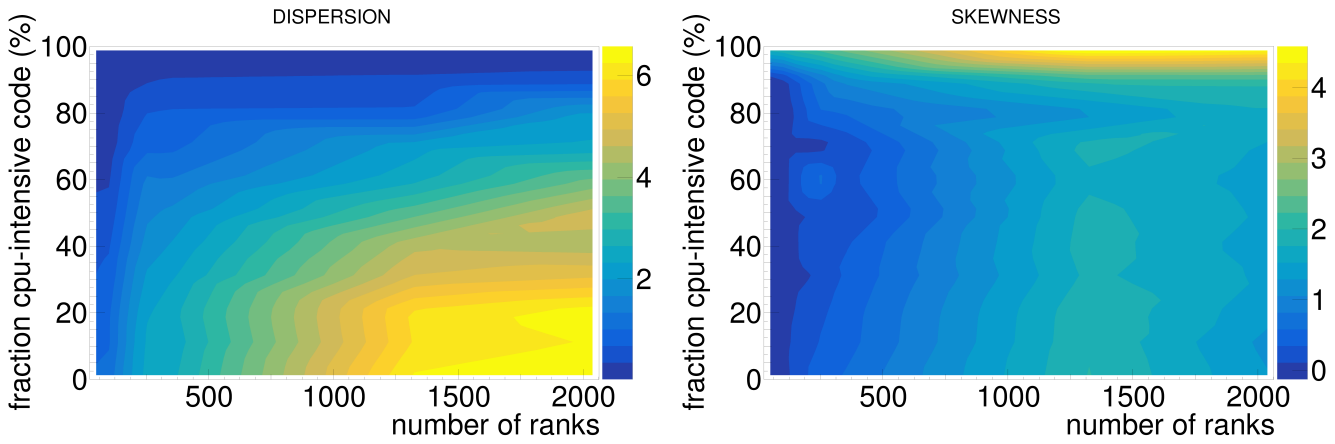

Figure 2: *Variability measures for a mixed parallel computations on Edison: dispersion (left) and skewness (right) of the timing distributions across ranks, as a function of scale and fraction of CPU-limited code. At 100%, the code is fully CPU intensive, mixing in memory- and network-limited code (at roughly a 3:1 ratio) until no longer CPU-limited.*

Mispredicting the critical path leads only to missed opportunities. Mispredicting the fastest path, i.e. the process to scale down the most, can have performance consequences. A similar analysis for the fastest path (Figure 1), shows that it is as hard to predict, with the same ~ 10% hit rate.

Overall, this data indicates that schemes relying on predicting the per rank duration of execution [16, 19, 21, 30] are likely to under-perform for our target application, as an incorrect schedule is likely on 90% of the tasks/cores affected.

3. Employing Variability as a Predictor

Fortunately, it turns out that the tables can be turned around: timings of individual processes/events may fluctuate, but we can predict, and make use of that variability itself. *For the rest of this study we use the term* **dispersion** *to refer to the sample standard deviation of all timings on all tasks executing between a pair of group synchronization operations.* Note that code between two barriers is also referred to as a **region**. For simplicity we will interchange the terms barrier and collective during the presentation. The implementation handles any flavor of operation, as described in Section 5.2.

Dispersion is induced by two main causes. Application dispersion is induced by domain decomposition and algorithms, which determine the execution duration on each task. Load balanced codes, either with static domain decomposition or dynamic load balancing, have low dispersion. Load imbalanced codes exhibit higher dispersion. The system itself contributes to dispersion through performance variability and noise introduced into the execution.

We conjecture that, given a load balanced code, dispersion is caused by the intrinsic nature of the system and the computation, when activities compete for the same shared resource[4]. Identifying these causes, determining their variability "signature" and their amenability to DVFS optimizations may allow us to build energy optimizations using a qualitative approach without any need to predict individual timings. Our insight is that no matter what the programmer's original intentions were, the stochastic nature of large scale computations allows us to predict the distribution of inefficiencies (e.g. timing of slack) in the code and react whenever "signature" distributions are identified.

System induced dispersion: We use micro-benchmarks that time code executed in between two barrier operations to understand where and how variability appears. We start with a statically load balanced benchmark, where each rank performs the same work, either communication, memory, I/O, or compute intensive. We then mix the types of work on ranks, as well as varying the amount of load imbalance. We distinguish between Flops-limited (CPU-limited) and DRAM bandwidth-limited code in the computation benchmarks, as the latter is amenable [9] to DVFS.

Intuitively, pure computation in SPMD codes is expected to be more predictive and static in nature. Communication, synchronization and I/O are prime culprits for variability, therefore prime candidates for DVFS, since there processes compete for hardware resources.

As shown in Figure 2 (left), dispersion in Flops-limited code is negligible at any scale up to 2,000 cores. Dispersion increases with DRAM pressure and inter-node communication, ending up more than 10× larger, at any scale. The shape of the distributions, measured with skewness and shown in Figure 2 (right), gives an extra distinction at scale. Memory-limited code uses many components of the hardware: the CPU, the hierarchy of caches, memory controllers, etc. Stochastic behavior combined from many sources leads to normal distributions, per the central limit theorem, and that is what we observe: skewness is small to non-existent. In contrast, variability in CPU-limited code comes from a single source, which leads to an asymmetric distribution with

[4] Resources are either hardware, e.g. memory, or software, e.g. locks inside runtime implementations.

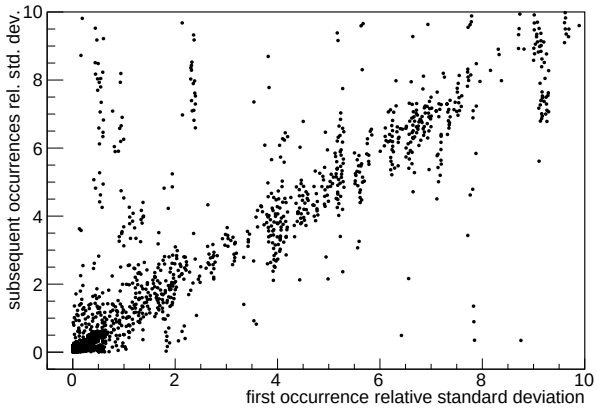

Figure 3: *Quality of prediction of variability per context in NWChem, using 1024 ranks. The dispersion of a re-occurring context is highly correlated with that of its first occurrence, leading to good predictivity.*

large positive skew, caused by a few stragglers. CPU limited code grows a statistically significant[5] right-side tail.

We have considered all combinations of code with different characteristics and assembled a set of micro-benchmarks to tune the optimization engine. For brevity we omit detailed results for communication and I/O intensive operations, we note that their behavior is qualitatively similar to memory intensive codes when combined.

Application induced dispersion: This is a measure of the intrinsic application load balance, and previous work relies on its predictability for static codes. Our insight is that "useful" imbalance can be recognized even for dynamically load balanced codes. For the NWChem experiment shown as unpredictable in Figure 1, we compute the dispersion for each region in its calling context, identified by the program stack traces. Figure 3 shows the results of comparing the dispersion of the first occurrence of each context with its subsequent occurrences, for contexts that repeat at least 10 times. The correlation is high and it is more likely that dispersion increases than decreases when contexts re-occur. Clearly its presence provides a good indicator of DVFS opportunities.

Selecting DVFS candidates: The previous results indicate that the type of behavior amenable to DVFS (communication, I/O, memory intensity) causes increased dispersion in "well" load balanced regions. These regions can be the result of either static domain decomposition or dynamic load balancing mechanisms. As misclassification may hamper performance, we still need criteria to handle the load imbalanced candidate regions.

While dispersion gives a measure of contention, skewness gives an indication of the shape (asymmetry) of the distribution, and thus an estimate of the behavior of the slowest tasks (critical path). For statically load imbalanced codes this behavior is quantitatively predictable, while for unsuccessful dynamic load balancing it is more or less random.

Negative skew means the mass of the distribution is concentrated towards the right of the value range: it indicates that most tasks take a long time, with a fat tail of fast tasks. Intuitively, this happens when most tasks execute on the "critical path". Conversely, positive skew means most tasks are fast, with a fat tail of slow tasks, meaning that one or a few tasks form the critical path.

Each task has some minimum duration, limiting the possible extent of tails on the left. When anything affects a task's execution (e.g. the operating system briefly pre-empting), it will invariably lead to a slowdown compared to the other, unaffected tasks. This creates a small tail on the right side of the value range, the likelihood of which is higher at scale. As more and more tasks are affected, such as happens on an increasingly contended resource, the distribution widens and the concentration of tasks moves to the right of the value range. This decreases skew, and may eventually flip it to negative. If the contention does not affect all tasks equally, e.g. in the case of a shared cache[6], the fat tail of stragglers continues to grow and skew remains positive. Thus, we want to exclude such regions, as it indicates no or uneven contention, such as CPU- or cache-bound code, and DVFS scaling will negatively impact performance.

Contention widens the distribution, skew is normalized to dispersion, and there is a lower limit on task duration. Thus unless the random variations are (much) greater than the region duration itself, system induced factors or poor dynamic load balancing will not cause a *large* negative skew. Large negative skew is more likely to correspond to static imbalance with some tasks consistently faster. We may either exclude such regions outright or apply a histogramming method to quickly look for bi-modal distributions. If found, only the dispersion of the slower (right-most) mode, which represents the tasks on the critical path, matters in principle. However, the presence of clearly detectable modes means that the dispersion of those sets of tasks that make up the modes is low, and that they are thus most likely not affected by contention.

Figure 4 summarizes our strategy of applying DVFS based on dispersion and skewness. The strategy is guided using the micro-benchmarks already described in this Section. These are supplemented, as described in Section 5.2, with micro-benchmarks to characterize the behavior of collective operations, as they can be either memory, CPU, or communication intensive. These cover all characteristics that we expect: uncontested, symmetrically (all tasks equally) and asymmetrically contested, balanced and imbalanced. The goal of this mapping is to find an easily identifiable region that indicates contention on resources and where we can thus apply DVFS scaling.

[5] Normalized by standard deviation; not in an absolute (i.e. wall time) sense.

[6] After a first miss, a (now slower) process faces a higher chance of seeing its cache lines evicted under any temporal locality eviction policy

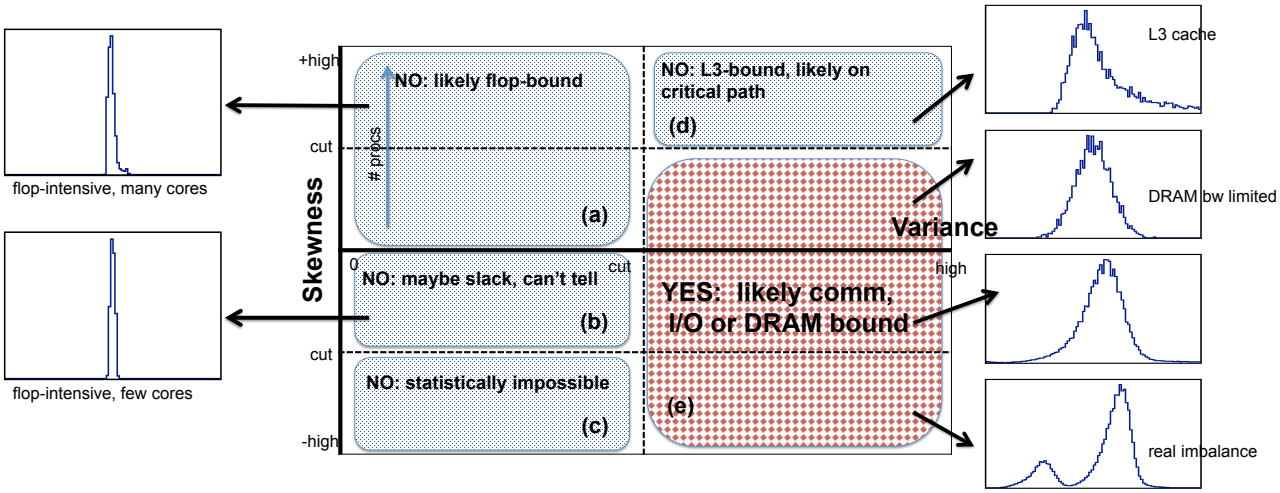

Figure 4: *Classification by skewness and variance. Regions using only uncontended resources, e.g. CPU, show low variance, whereas regions using contended resources, such as memory, have high variance. Flop-bound code has large positive skew at scale, as does any code that has a significant critical path. This leaves the lower right corner (high variance, low or negative skew) as most amenable to DVFS.*

For any observed execution falling in *Tile* **(e)** *we apply DVFS*. The high variance (larger than a threshold), small positive or negative skew are characteristics of communication, I/O or DRAM bandwidth limited codes. These correspond to contention that affects all tasks in the same way. Applying a cut on large negative skew is not needed in our target applications and we chose not to: the slower tasks drive the timing and if these indicate (because of their dispersion) contented resources, then applying DVFS works fine. As outlined above, strongly statically load-imbalanced CPU-bound tasks can exhibit a bi-modal distribution, and a more general implementation should detect those.

For any other execution summarized by Tiles **(a)**, **(b)**, **(c)** and **(d)** *we do not attempt DVFS*. For brevity, we give only the intuition behind our decision, without any benchmark quantification. Tile **(a)** captures flop-bound code. In this case note that skewness increases with concurrency. Tile **(b)** captures code with real load imbalance, but we cannot determine where, or what actual code mix is executed. Code in Tile **(d)** is unlikely to occur in practice (and doesn't in NWChem). Finally, Tile **(a)** captures code that executes in the last level of shared (contended) cache or code with a critical path.

Our approach can handle statically load balanced and dynamically load balanced codes. It can also handle static load imbalanced codes provided that imbalance is less that system induced variation. The algorithm is designed to filter out very large static load imbalance (see Figure 11). In contrast, existing work in energy optimizations for HPC codes handles codes with large static load imbalance.

It may seem worthwhile to use hardware performance counters and instrument system calls to get more detail of the programs behavior, and use that to refine the selection. However, we have found that logical program behaviors do not map so neatly. For example, a polling loop while waiting for one-sided communication to finish is fully CPU-bound, but does benefit from DVFS. In addition, regions contain a mixture of different behaviors, which combined with true stochastic behavior severely risks over-fitting when too many parameters are considered. Finally, the hardware itself and the operating system already use the information available from performance counters and system calls to apply DVFS, pocketing most of the energy savings accessible that way.

4. Design and Implementation

We develop single-pass, online optimizations, as well as offline optimizations. The online approach exploits the iterative nature of scientific codes: for each region first observe execution under different frequencies, then decide the optimal assignment and apply it when re-executed. Regions are distinguished in their calling context using the program stack traces. The online optimization is built for scalability and practicality, while we use the offline version to understand and tune the online optimization engine.

Region Partition: In one sided communication, inter-task synchronization is explicit and not associated with the regular `Put/Get` operations. Often these codes use non-blocking communication overlapped with independent computation, separated by group synchronization operations, such as barriers, to maintain data consistency. To capture the dynamic nature of the execution we use a context sensitive approach where we identify regions with a hash, created from the re-

turn addresses of the full stack trace, at the preceding collective operation.

Region Selection and Clustering: To estimate dispersion and skewness, we replace `Barrier` with `Allgather` and augment other collective calls such as `Allreduce` with `Allgather`, using linker wraps and gather the execution times of all processes. A region is then considered a candidate for frequency scaling if these three criteria are met:

- Total region duration above `THRESHOLD`.
- Dispersion above `STDEV` of average duration.
- Skewness below `SKEW`.

The numerical values of the parameters above depend on system and concurrency, e.g. dispersion increases with scale. See Section 5.2 below for a full discussion.

The purpose of clustering is to create larger sections to scale, and thus save on switching overhead. Clustering for tolerating DVFS latency was first proposed by Lim et al [21], which present an algorithm to identify chains of MPI calls that occur close in time during execution. In our case, to obtain results in practice we had to develop two extensions to this algorithm. The three principle components of our clustering are: 1) collect regions which each meet the necessary criteria, similar to Lim's approach; 2) extending these clusters with regions of short duration (assumes communication is not CPU limited); and 3) accepting a small loss of performance to cover a gap between clusters when switching would be even more expensive ("super-clusters").

With reference to Algorithm 1 it can be seen that attempts to build clusters are the normal case. Cluster building resets if a long region that does not meet the criteria is encountered, but does not stop. Once a cluster reaches `THRESHOLD`, it is marked for scaling, which will happen on the next repetition of the context that started the cluster. Any region with a duration less than `SHORT` will always be accepted into the current cluster, because of the assumption that it is dominated by communication. Simply setting `SHORT` to 0 will remove this clustering feature from the algorithm. Finally, a `BRIDGE` cutoff is used to allow clusters to combine to super-clusters, where the cost of the bridge is preferable compared to the cost of switching frequencies twice. A bridge is built until a new bona fide cluster forms, or if it gets too large, at which point the algorithm resets. Super-clustering can be switched off, by setting `BRIDGE` to 0.

Frequency Selection: For each cluster we want to determine a global frequency assignment to adjust all cores.

In the online analysis we execute the instrumented program and data is collected at each collective operation, then clustering runs and candidate clusters are created. On the second occurrence of a candidate, the algorithm attempts a different global frequency. As learning needs to be short for practical reasons, we use only two target frequencies. We try each frequency only once per cluster. The algorithm has a safety valve, tracking the maximum duration per context

Algorithm 1: Clustering of regions

```
   Input: context, region, and is_noisy
   Result: program state update
1
2  /* decisions is a map of <context, DVFS decision>; program State is either CLUSTERING,
      SCALING, or BRIDGING; SHORT, STDEV, SKEW, and BRIDGE are tunable (see text)    */
3
4  if is_noisy then
5      /* collect this region into a cluster                                        */
6      COLLECT(context, region)
7      return
8  else
9      /* reject, unless very short, or not too long and in between clusters       */
10     if SHORT < region then
11         if State == SCALING then
12             /* potential end of previous cluster, start of a new bridge         */
13             cluster ← 0
14             State ← BRIDGING
15         if State == BRIDGING then
16             BRIDGING(context, region)
17             return
18         RESET_LOCAL_STATE()
19     else
20         /* short region: communication dominates, collect anyway                */
21         COLLECT(context, region)
22     return
23
24 procedure COLLECT(context, region)
25     if State == BRIDGING then
26         /* (potential) start of a new cluster                                   */
27         cluster ← 0
28         State ← CLUSTERING
29     switch State do
30         case CLUSTERING
31             cluster ← SUM(cluster, region)
32             local_decisons[context] = SCALE
33             if THRESHOLD < cluster then
34                 /* cluster has grown large enough                               */
35                 State ← SCALING
36                 SCALING(context, region)
37         case SCALING
38             SCALING(context, region)
39
40 procedure SCALING(context, region)
41     if pending_decisions then
42         decisions.update(local_decisions)
43         decisions.update(bridge_decisions)
44         RESET_LOCAL_STATE()
45     decisons[context] = SCALE
46
47 procedure BRIDGING(context, region)
48     cluster ← SUM(cluster, region)
49     bridge_decisons[context] = SCALE
50     if BRIDGE < cluster then
51         /* distance from last cluster has grown too large                       */
52         RESET_LOCAL_STATE()
```

where DVFS is applied. If the scaling causes unwarranted slow-down (determined by the ratio of the frequencies plus a margin) on the slowest rank compared to the unscaled iteration, the region is reverted if it is above the `BRIDGE` threshold, or re-entered into the decision process if not. All processes have the same data, so reach the same conclusion. If the decision gets reverted, it applies to the next occurrence.

Putting a limit on the number of trial frequencies is motivated by several factors. First, the number of dynamic repetitions of a context may be small, e.g. at most ten repetitions in NWChem. Second, as some trials are bound to degrade performance, fewer trials eliminate potential slowdown. Finally, the region selection criteria and acceptable slowdown are not independent. The actual frequency values are quantified in Section 5.2.

While the online analysis is concurrency and input independent, the offline analysis gives us an indication of how much optimization is unexploited for a given problem (fixed concurrency and input), when comparing to online. Here, we execute the program once for each discrete frequency

level available for DVFS and collect traces. A trace analysis combines all experiments, forms clusters and selects the optimal frequency assignment: this can be any of the available frequencies on the system. Thus, contexts that occur only once, most importantly initialization, and the first iteration of a repeating context can scale when re-executing the program. Further, the offline analysis can select the most energy-optimal candidate, given a constraint on performance loss, from a static set of frequencies.

When using the offline analysis, the optimized execution does not augment collectives with `Allgather`, resulting in faster execution. Note that the scalability of `Allgather` is not a real concern for the online analysis: we can revert to the original collective after making a decision, but in doing so give up the safety valve. In our experiments, reversals do not happen after the first successful scaling, so we could revert to the original collective on the third occurrence. Furthermore, changes in collectives can only have a measurable effect over the full run, if the original collectives consumed a significant portion of the overall running time. If so, reversals are unlikely to happen.

5. Evaluation

We evaluate the efficacy of our optimizations on the NWChem and UMT2K applications described in Section 5.3. We compare the online and offline optimization approaches using up to 1,034 cores on two clusters and one large scale HPC production system. For completeness we have attempted to compare against MPI based quantitative approaches, Adagio [30] and the clustering [21] algorithm used by Lim et al. The results are described in Section 5.6.

5.1 Experimental Methodology

Platforms: The Teller [31] cluster has four AMD A10-5800K quad-core processors and 16GB main memory per node. There are seven available frequencies for scheduling, ranging from 1.4 GHz to 3.8 GHz. Each core can be individually controlled; in the idle state cores consume 45W, 55W at 1.4 GHz and 110W at 3.8 GHz. Each node runs on Red Hat Enterprise server 6.2 and Linux kernel 2.6.32, and the frequency switching is implemented on top of the `cpufreq` [5] library. The Edison [10] Cray XC30 system at NERSC [23] is a large scale production system, with two 12-core Intel Ivy Bridge 2.4 GHz processors per node and 5,576 total nodes, 133,824 cores in total. Frequencies can be set in increments of 0.1 GHz from 1.2 GHz to 2.4 GHz, only at socket granularity. At idle cores consume about 65W, to 115W at 1.2 GHz and up to 250W at 2.4 GHz. Shepard is a 34 node cluster with dual socket Intel Haswell 16-core processors. This processor is to be deployed in the next generation large scale systems at DOE Labs [1, 4] in early 2016. The frequency on Haswell can be selected per (hyper-threaded) core, from 1.2 GHz to 2.3 GHz. However, cores share many resources, which are not affected until all cores in the socket scale down. Additionally, on Shepard, the Mellanox InfiniBand FDR driver requires a single frequency to be set across all cores. At idle a node draws 76W, 104W at 1.2 GHz, 150W at 2.3 GHz and 220W in Turbo mode.

Methodology: We measure at-the-wall power consumption, as it is the ultimate indicator of any energy savings. We use micro-benchmarks to determine the DVFS latency. On Teller and Shepard we use the PowerInsight [**?**] interface to collect power data, and integrate it with application level time stamps to get the energy used. The DVFS switching latency is $\approx 100\mu s$ on Teller and Shepard [13]. Measured results on Teller and Shepard are with our optimizations during the application execution.

On Edison power is measured by the Cray power monitoring counters, which sample at a frequency of about 10 Hz and are read from a virtual file system under `/proc`. As Edison is a large scale production system, the only DVFS control allowed is at job startup, when a single frequency can be selected. Therefore, results on Edison are estimated using modeling on timing and trace data. We run the application at all available frequencies, including default Turbo mode. To account for optimization overhead, the algorithm is executed during each run, all except the final DVFS calls. To account for DVFS switching overhead we use $\approx 100\mu s$ delays, which is a conservative estimation [22] for this system. Trace files for each frequency are obtained by averaging on each task the duration of a region across at least five runs.

As an extra validation step for the Edison results, we have compared the clusters selected by the model with the clusters selected by the online algorithm on Teller and Shepard at similar concurrency. There is a very high overlap in selected regions, confirming that we do indeed select in the simulation that portion of the execution amenable to DVFS due to contended hardware resources.

5.2 Tuning Parameters

On each system, we select the high and low frequency thresholds using the performance micro-benchmarks described in Section 3.

For the *high frequency* we choose between Turbo mode and the highest static frequency available. On HPC production systems, the Turbo mode is enabled by default as it usually attains the best performance: this is the case on Teller and Edison where we select it. For reference, 2.4 GHz is the highest possible static frequency on Edison, but setting it explicitly will switch off Turbo-boost and DVFS by the hardware. In comparison to static 2.4 GHz, Turbo gains 8% in performance at a cost of 13% in energy for NWChem, running CC. On Shepard, best performance is obtained at a static frequency of 2.3 GHz due to performance bugs in the Mellanox InfiniBand driver, uncovered during this work. On Shepard we select static 2.3 GHz as the high frequency. In Turbo mode, communication is significantly slower than at static 2.3 GHz. The driver expects that all cores are set at the

same frequency as it reads the current CPU configuration file and uses it throughout to estimate delays. Turbo mode is logically denoted by the 2.301 GHz frequency, while in reality the cores will run anywhere between 2.8 GHz-3.4 GHz, depending on load. In our experiments we have modified the driver to configure the NIC with the correct high Turbo frequency. This recouped some of the loss, but a vendor fix is required before Turbo mode can match the performance at 2.3 GHz.

Note that all MPI quantitative approaches[29] cannot handle well Turbo mode since they rely on a static reference frequency.

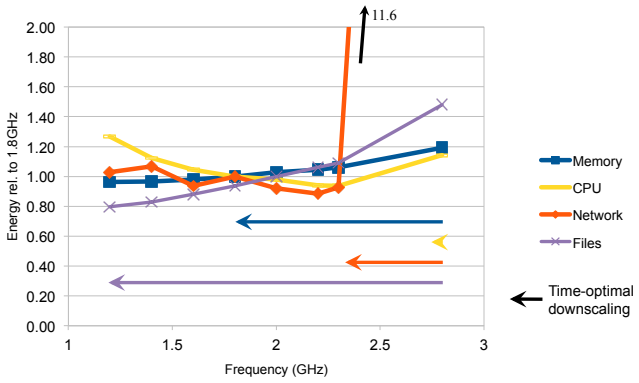

Figure 5: *Relative energy usage (1.8 GHz == 1) for micro-benchmarks that are CPU, memory, network, and file access limited, on Shepard. Network results are before driver fixes, and using all cores. Arrows indicate the lowest frequency possible while remaining time optimal.*

We select the *low frequency* based on the energy cost of codes that are limited by specific resources. Figure 5 shows such experimental results on Shepard. Different resources are impacted differently, thus the choice for low frequency determines what regions we want to select for DVFS, and that means it determines the dispersion and skewness cut-offs. In a few iterations this converges on an optimal set.

For example, start with a low frequency of 2.1 GHz. This scaling affects network- and CPU-bound codes, and the latter the most. Next, create dispersion and skewness plots for codes for different mixtures of part CPU- and network-bound, part not. See for example Figure 2 where we have done this for Edison. Scaling down to 2.1 GHz for CPU-bound code incurs a cost of 33% (assuming most cores are in use, i.e. an effective Turbo frequency of 2.8 GHz). The effect of intra-node communication is the same, the effect of inter-node communication is a speed-up, but only marginally so after our driver fixes. Assume in the first iteration that 20% of the program time will be scaled. Thus, the cost could be 20% of 33%, or 6.6%. We want to bound this by 2%, so the scaled regions may not be more than 30% CPU- or network-bound. We now read the dispersion and skewness cut-offs directly from their respective plots at the cross of 30% and the desired scale. Run a test, verify the 20% assumption, and iterate with an improved selection (possibly with an adjusted low frequency)[7] until convergence on an optimal set.

Since the parameter set is over-constrained, we can make trade-offs. For example, choosing a higher low frequency allows looser cut-offs, and thus a greater selection of program regions. The higher frequency results in lower savings, but these are applied to a larger portion of the program, netting the same result: the algorithm is very robust over a range of parameters. However, the trade-off between performance and energy is not a linear function of the CPU frequency for most resources: the largest gains are had by coming down from the highest frequencies, with only small gains as we approach the frequency at which the memory runs. We therefore choose a low frequency roughly mid-way between memory and max: 3.4 GHz for Teller, 2.1 GHz for Edison, and 2.1 GHz on Shepard.

Finally, for clustering, we choose the `SHORT` threshold to be $5\times$ the communication cost of a `Barrier`, and the `BRIDGE` cutoff to be $10\times$ the cost of DVFS latency (i.e. the expected average bridge size would be half that). The actual values chosen need not be highly tuned, and we don't change them with scale, because the different parts of the algorithm complement each other. For example, if `SHORT` is too tight, bridging will cover most cases anyway. Likewise, collecting short regions does not so much increase the size of clusters, but rather allows building longer bridges in parts of the program that are dominated by communication.

5.3 Application Benchmarks

NWChem [33] delivers open source computational chemistry software to a large user community. It uses Coupled-Cluster (CC) methods, Density Functional Theory (DFT), time-dependent DFT, Plane Wave Density Functional Theory and Ab Initio Molecular Dynamics (AIMD). It contains roughly 6M lines of code. The main abstraction in NWChem is a globally addressable memory space provided by Global Arrays [25]. The code uses both one-sided communication paradigms (ARMCI [24], GASNet [11], ComEx [8], MPI 3.0 RMA), as well as two sided communication (MPI). Most methods implement a Read-Modify-Update cycle in the global address space, using logical tasks that are dynamically load balanced. Communication is aggressively overlapped with other communication and computation operations. The code also provides its own resilience approach using application level checkpoint restart, thus file I/O dominates parts of the execution.

With two sided MPI as the transport layer, the code performs a sequence of overlapped `Isend` — `Probe` — `Recv(ANY_SOURCE)` operations. With ARMCI as the transport layer, the code runs in an asymmetric configuration where proper "ranks" are supplemented with progress threads

[7] This assumes a distribution in mixtures across regions; true in large, complex applications, but not in highly synchronized, simple codes.

that perform message unpacking and Accumulate operations, driven by an interrupt based implementation.

We have chosen NWChem since no previous work handled dynamic load balance mechanisms, nor one-sided communication in large scale application settings. Furthermore, note that the ARMCI back-end is challenging due to its asymmetric runtime configuration using progress threads. We have experimented with both MPI and ARMCI. For brevity we concentrate on presenting ARMCI results, see Section 5.6 for MPI.

For this paper we experiment with the CC and DFT methods, as these account for most usage of NWChem. CC is set to run with and without I/O (writing partial simulation results to disk for resilience reasons). We used two science production runs: simulation of the photodissociation dynamics and thermochemistry of the dichlorine oxide (Cl_2O) molecule, and of the core part of a large metalloprotein.

Dynamic Behavior of NWChem: Slack is virtually non-existent, as a result of dynamic load balancing in NWChem. When running on 1,034 cores on Edison, a very large fraction of regions (more than 50%) are very short, on the order of several tens of μs. On the other hand, these short regions account for at most 10% of the total execution time and need to be clustered to prevent excessive frequency switching. For reference, the program executes 204,452 barriers during this run. The most relevant regions for DVFS are longer (in the few ms range) and repeat only about 10 times, which indicates that online learning needs to be fast. This also means that an energy optimization approach needs to be able to handle both short execution regions, as well as long running program regions. As already shown, predictions of execution time between synchronization operations in NWChem are likely to fail. We examine each method at increasing concurrency, up to 1,034 cores. CC executions exhibit different dynamic behavior with increased concurrency. DFT execution is CPU-intensive.

UMT2K [32] is a 3D, deterministic, multigroup, photon transport code for unstructured meshes [32]. It is written in MPI and OpenMP and it provides its own checkpoint/restart mechanism. We have chosen this benchmark as it provides a common reference point with state-of-the-art quantitative approaches able to handle MPI codes. UMT2K contains roughly 170K lines of code.

Dynamic behavior of UMT2K: The benchmark is CPU-intensive and performs blocking MPI communication, i.e. no overlap. Communication regions account for about 20% of the execution time. It is largely load balanced, with some communication regions having increased work for MPI Rank 0. The time intervals between two collective operations are large ($\approx 250ms$), the benchmark performs about 1000 `Barrier` and `Allreduce` operations each over a 4 minutes long execution. Overall, UMT2K is reasonably well balanced with coarse grained computation between collective operations.

5.4 Impact of Algorithmic Choices

For brevity, we do not discuss the Teller results in detail. The Edison and Shepard CPUs are one and two generations ahead respectively, have better hardware power management and energy optimizations on these platforms are more challenging. For reference, on Teller for CC we observe as much as 7.4% energy savings for a 1.7% slowdown provided by an online optimization that uses 3.4 GHz as the target frequency, running on 128 cores.

We'll concentrate our interest on Edison, as this is a tightly integrated production HPC system. Figure 6 presents results for the online optimizations of the CC run. The labels contain the concurrency, e.g. CC-1034 refers to a run on 1,034 cores. For each configuration we allow the low frequency to take the values indicated on the $x - axis$. For reference, the runs perform 75,233, 75,085 and 204,452 regions when running on 132, 528 and 1,034 cores respectively. These occur in roughly 6,000 to 9,000 distinct calling contexts, depending on the input. The typical execution is on the order or 20 minutes or more.

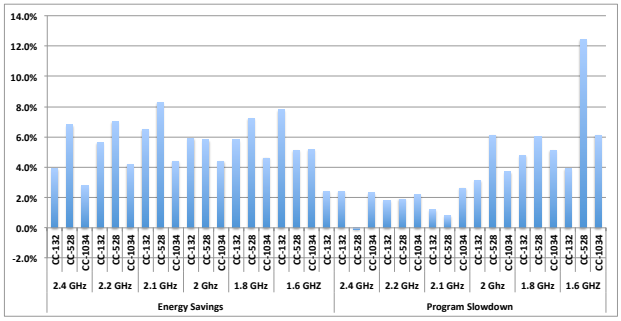

Figure 6: *Summary of results on Edison for CC running at increasing concurrency (132,518,1034) and with different target low frequency for the online algorithm.*

The execution of CC-132 is memory bandwidth limited and we observe energy savings as high as 7.8% for a slowdown of 3.9% when varying to 1.6 GHz. When using the "default" target 2.1 GHz frequency, the optimization saves 6.5% energy for a 1% slowdown. A low frequency is selected for about 25% of the total program execution using $\approx 3,600$ DVFS switches. The execution of CC-528 is dominated by a combination of memory intensive execution and communication. The energy savings are as high as 8.3%, for a slowdown of only 0.8%. Low frequency is selected roughly for 33% of the execution, using $\approx 8,000$ switches. The execution of CC-1034 is dominated by a combination of I/O and communication. In the best case we observe savings of 4.4% with a 1.5% slowdown. Low frequency is selected for about 23% of the execution, using $\approx 70,000$ switches.

Figure 6 also illustrates that most of the benefits are obtained when lowering the frequency to 2 GHz or 2.1 GHz,

close to the memory frequency but not lower. This validates our choice of considering only one target frequency and in practice we use 2.1 GHz as default.

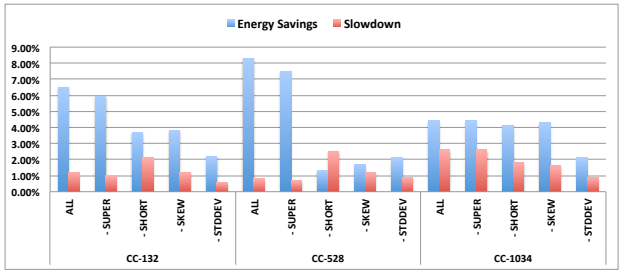

Figure 7: *Impact of algorithmic design choices on the efficacy of the online algorithm using a target frequency of 2.1 GHz on Edison.*

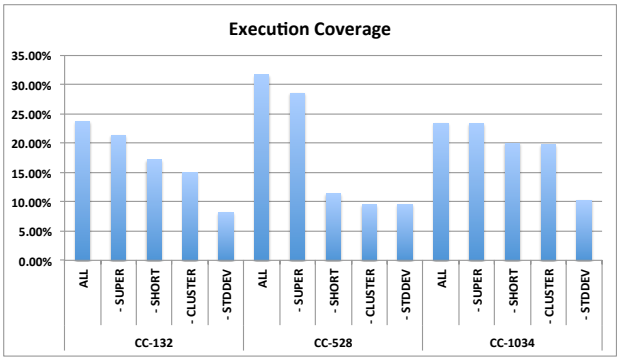

Figure 8: *Percentage of the execution time at low frequency as determined by algorithmic choices on Edison using a target of 2.1 GHz.*

Figures 7 and 8 provide some quantitative detail about the influence of our algorithm design choices. Overall, they illustrate the fact that both clustering and the variability criteria are required in practice to cover the spectrum of code dynamic behavior. We present energy savings, slowdown and execution coverage for multiple algorithms, compared to the complete online algorithm labeled ALL. Each label designates the criteria we subtract from the full algorithm in a progressive manner: "-SUPER" denotes lack of forming super-clusters (i.e. `BRIDGE` = 0 Algorithm 1), "-SHORT" denotes ignoring short regions (i.e. `SHORT` = 0), "-CLUSTER" denotes no clustering at all. Finally "-STDDEV" denotes an algorithm that ignores completely variability, but still cuts on skewness. For reference, the series labeled "-SHORT" (i.e. no refinement beyond clustering) is the equivalent of the algorithm presented by Lim et al [21].

For CC-132 and CC-528, most of the benefits are provided by the clustering, rather than the variability selection criteria. Also note that forming super-clusters is mandatory for performance, as illustrated by the increase in energy savings and decrease in overhead for "-SUPER" when compared to "-SHORT" for CC-132 and CC-528. When increasing concurrency, the variability selection criteria provides most of the benefits of the optimization, and clustering improves behavior only slightly. Similar conclusions can be drawn when examining the execution coverage in Figure 8, rather than performance improvements.

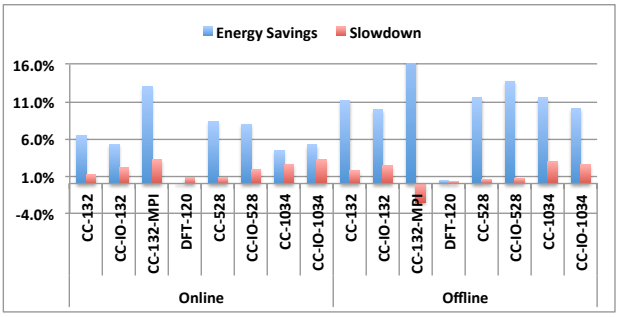

Figure 9: *Comparison of energy savings and slowdown for CC, CC-IO, DFT and CC-MPI when using online and offline optimizations on Edison with a target low frequency of 2.1 GHz.*

For brevity, we did not provide detailed results for all benchmarks as they show similar trends to the CC runs. In Figure 9 we show results for runs of CC-IO, which runs the resilience methods, DFT and selected configurations running on MPI. For CC-IO we observe even better energy savings than for CC, due to extra optimization potential during I/O operations. The DFT execution is flops-limited and as already described in Section 3 our approach does not identify many regions as DVFS candidates. MPI results are further discussed in Section 5.6.

On Shepard, like on Teller, results are obtained with DVFS during application execution and summarized in Figure 10. For CC, we used molecules of increasing sizes as input: (Cl_2O) (OCT) and $(C_4H_6N_3O_2)$ (DCO). As shown, we observe as much as 12% energy savings with in most cases only a small performance penalty.

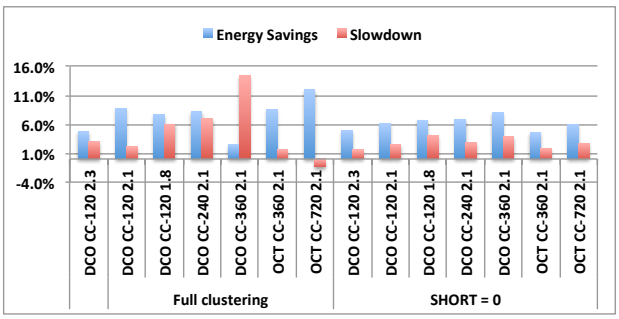

Figure 10: *Comparison of energy savings and slowdown for CC (a small and large molecule), at different low frequencies and with and without clustering of short regions.*

As we scale up the small molecule input, it becomes increasingly dominated by communication, which is CPU-dependent if intra-node. As a result, "blindly" accepting short regions carries an increasingly large cost, requiring `SHORT` set to 0 to get good results.

For DFT, timings are dominated by file access on Shepard (even at small scale) and their range is huge (up to 4x), putting doubt in the usefulness of the results. Nevertheless, we ran a large number of jobs and averaged them, achieving savings of 9% for no appreciable loss in performance. That result is encouraging, but much larger savings can be had by installing a better I/O system.

As expected, we did not achieve energy savings on the UMT2K benchmark on Shepard, nor did we slow it down. The load balanced parts are CPU-intensive and rejected by our criteria. Some communication regions have increased work for Rank 0: the imbalance is within our threshold and the algorithm decides to apply DVFS. However, at subsequent executions of the context the variation in the execution time of Rank 0 is too large and the algorithm disables DVFS. Compared to 9000 in NWChem, the dynamic behavior is much simpler: we observe only 40 distinct calling contexts.

5.5 Optimality of Online Algorithm

The online algorithm observes the execution of any region for a small number of repetitions and if necessary varies the frequency to a unique predetermined value. This leaves untapped optimization potential. In Figure 9 we include for reference the energy savings and the slowdown for the offline trace based optimization approach which chooses the optimal frequency for any given cluster. As illustrated, the offline approach almost doubles the energy savings for similar or slightly lower runtime overhead. In particular, we now observe 11% savings at 1,034 cores for CC-1034. More than half of the additional energy gains come from the ability to perform DVFS on the first occurrence of any region, rather than from choosing the optimal frequency assignment for a particular region. To us, this validates the algorithm robustness and relative lack of sensitivity to the selection of the low frequency threshold; it is better to act fast but imprecise, rather than wait for enough data for statistical significance to compute an optimal solution.

5.6 Impact of Communication Paradigm

The results presented so far have been for NWChem configurations using one-sided communication. For brevity we offer a summary of our findings when running NWChem configured to use MPI. First, the application runs slower, and we have observed more than 4× slowdown on Edison when comparing MPI with ARMCI performance on 1,034 cores. A large contribution is attributed to slower communication, which implies larger possible gains from DVFS. Due to the two-sided nature, instrumenting at barrier granularity shows that the code exhibits less imbalance than runs with ARMCI, as induced by the multiple `ISend/IRecv/Probe` operations performed between barriers.

With MPI we observe much higher energy savings (up to 20% using offline analysis, up to 16% using online), relative to the one-sided results. The portion of the execution affected is also higher, up to 50%. Since the MPI code is seemingly balanced, clustering provides most of the optimization benefits, while the variability criteria provides a safety valve. This is in contrast with the one-sided behavior, where recognizing variability is required at scale. When compared to Lim's approach, we use different selection criteria when clustering, namely variability, but after that, we do add back short regions, which is what their algorithm uses for its clustering criteria. For example, our full optimization (ALL) provides energy savings of 13% with a slowdown of 3.3%, while an algorithm similar to Lim's that does pure clustering (-SHORT) attains 9.3% savings with 0.9% slowdown. Note that adding the (presumed communication bound) short regions (-SUPER) so that the selection criteria are similar, too, results in 11.8% savings, for a cost of 2.9%.

The network hardware also emphasizes the differences between communication paradigms. InfiniBand on Shepard has little hardware assist for message injection, while Cray Aries [2] on Edison provides FMA and BTE hardware support for small and large messages respectively.

On InfiniBand, one-sided small transfers and collective operations are dominated by CPU overhead: our approach classifies them accordingly and does not attempt DVFS. Indeed, scaling down the frequency during these operations leads to execution slowdown. Our approach classifies MPI two-sided transfers as amenable to DVFS: lowering the frequency during these transfers does not affect execution time. Two-sided communication combines data transfer and inter-task synchronization semantics: here waiting and network latency dominates the message initiation CPU overhead.

On Edison, our criteria identifies both one-sided and two-sided small transfers and collectives as amenable to DVFS: indeed this is the case and lowering the frequency does not affect their performance.

Overall, it seems that MPI codes exhibit more DVFS potential due to their over-synchronized behavior. This becomes apparent when strong scaling NWChem on Shepard enough that messages become small. Our method still finds DVFS opportunities, however, because file system access is heavily contented at scale.

5.7 Comparison to Adagio

Adagio [30] provides state-of-the-art scaling of MPI applications using a quantitative approach. To provide a fair comparison, we started with the official source [3], fixed the baked-in assumptions about the hardware (e.g. four cores per node, no hyperthreads), made the code "hyperthread-aware", extended it to handle Turbo, optimized frequency setting, and built it in optimized (-O3) mode. With that, we verified with the provided tests and under the same original conditions of one MPI rank per DVFS/clock domain that the algorithm indeed works as expected. When applied to our test applications Adagio finds no energy saving potential in NWChem and causes a 60% slowdown (at an 28% energy *increase)* in UMT2K, due to selecting a too low fre-

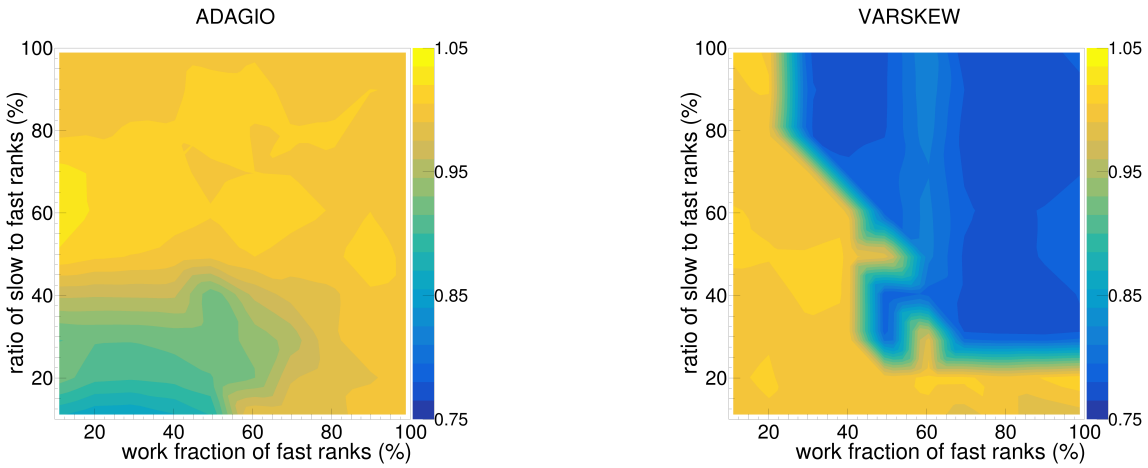

Figure 11: *The savings effect as a function of imbalance for Adagio (left) and our algorithm (right) for a memory bandwidth limited micro-benchmark. The smaller structures are an artifact of the plotting and limited sampling.*

quency in communication regions, which are CPU sensitive on Shepard.

The results showcase the differences between quantitative and qualitative approaches. Adagio predicts the timing for each region and computes its ideal frequency as a linear combination of the discrete frequencies available. Frequency on each core is changed during a region using interrupts and this poses scalability problems with the number of cores. To amortize overheads, it requires relatively long regions (a multiple of 100ms) executed at a given frequency. Such long region times do not occur often in NWChem: the typical region times are too short to even accept the overhead of the additional frequency shifts, and clustering is necessary. Besides coarse grained regions, Adagio requires load imbalance between ranks proportional to the frequency differential between two consecutive assignments. When lacking an optimal choice, the algorithm conservatively assumes a CPU-bound code, or a slowdown equal to the frequency differential. The original Adagio implementation uses the highest static frequency as reference. With our extensions to handle Turbo mode, the differential to next frequency is large and depends on the number of cores in use (e.g. from 18% to 32% on Shepard), and none of our applications exhibit imbalances of that magnitude. Figure 11 shows results for a memory bandwidth limited micro-benchmark designed to favor Adagio with regions of several seconds long. Adagio (left) works best when only a few ranks are slow (i.e. most cores can be scaled down) and the imbalance is significant enough. When most cores are slow, the energy savings effect of scaling down only a few cores is negligible. Our algorithm scales down more aggressively, which allows it to capture larger savings even in much smaller regions, but conversely, it can not find savings if there is significant imbalance. Note that Adagio performs similarly for a CPU-limited micro-benchmark, whereas our algorithm will not achieve any savings at all in that case. Contrarily, on Shepard where small to medium communication and collectives are CPU-bound, Adagio indiscriminately scales down communication resulting in large performance penalties for NWChem and UMT2K.

6. Other Related Work

Energy and power optimizations have been explored from different perspectives: hardware, data center, commercial workloads and HPC. At the hardware level, *memory intensity* [9] has been used to identify DVFS opportunities. To our knowledge, approaches similar in spirit are implemented in hardware in the Intel Haswell processors.

Raghavendra et al [26] discuss coordinated power management for the data center level. They present a hierarchical control infrastructure for capping peak or average power. Their detailed simulation for enterprise workloads illustrates the challenges of tuning these schemes even at small scale.

In the HPC realm, most optimizations use slack in MPI communication to identify optimization opportunities. Initial studies [28] used offline, trace based analyses that often solve a global optimization problem using linear programming. An optimization strategy is built for a single input at a given concurrency. While more generality can be attained by merging solutions across multiple inputs or scales, these methods suffer an intrinsic scalability problem introduced by tracing. Furthermore, these analyses do not handle non-blocking MPI communication `Isend/Wait`.

Later approaches improve scalability using online analyses [21, 30] and mitigate high DVFS latency using clustering [21] techniques. Some of these techniques [30] consider in addition computational intensity (instructions per cycle or second) as a measure of application speed under DVFS. We have already highlighted the differences and extensions introduced in our work.

Other approaches that consider hybrid MPI+OpenMP codes [19], are able to make per DVFS domain (socket) decisions, as long as *only one MPI rank* is constrained within a DVFS domain. The initial studies use offline analyses

and consider MPI and OpenMP in disjunction, running in a master-slave configuration. The preferred strategy for the OpenMP regions is to reduce parallelism and turn cores off completely, referred to as dynamic concurrency throttling. Extensions are required to handle codes with more dynamic parallelism, or when running either multiple MPI ranks per clock domain or OpenMP over multiple clock domains.

Programming model independent approaches tend to be used by hardware or only within the node [9] and use time slicing. The program is profiled for a short period of time and a DVFS decision is made for the rest of the time slice. For HPC, CPU-Miser [12] employs this technique. Application dependent approaches identify and annotate algorithmic stages and iterations, using either online (Jitter [16]) or offline [17] analyses. The workloads considered in these studies are MPI with static (im)balance and DVFS control per core.

Energy optimizations for one-sided communication is examined by Vishnu [34] using the ARMCI implementation. They design interrupt based mechanisms for the ARMCI progress thread and lower voltage when waiting for communication completion. The evaluation is performed on micro-benchmarks doing blocking communication, e.g. Put followed by Fence (quiesce) with no overlap, and with per core DVFS. The ARMCI implementation of NWChem performs overlapped non-blocking communication.

Recent work by Ribic and Liu [27] describes the design of an energy efficient work-stealing shared memory only runtime for Cilk. Their approach tries to optimize for the critical path (workpath-sensitive), while keeping in mind the relative speed of workers (workload-sensitive). While the hardware used for evaluation supports *clock domains* and hyperthreading, the evaluation is performed running with only one thread per domain, to avoid interference between decisions made on each core.

7. Discussion

Our approach works for SPMD codes that use one- and two-sided communication and dynamic load balancing: none were successfully handled by previous DVFS optimizations for distributed memory programming. We can also handle static load balance, provided no extreme imbalance. Previous approaches are best suited for codes with large static imbalance. During this research, we have striven to provide hardware and runtime independent mechanisms and eschewed the use of hardware performance counters. These may be used by libraries and hijacking them hampers portability in large code bases. By sampling hardware counters we believe we can extend out the variability criteria to recognize and avoid CPU intensive regions in codes with static load imbalance. Although expected gains are small, sampling counters is also likely to allow choosing a better low frequency for any region, using the methodology described in Section 5.2.

We can further improve the efficacy of the online algorithm by lowering its runtime overhead: augmenting collectives with `Allgather` in large scale runs introduces as much as 1% runtime overhead in our experiments. One can easily imagine approaches where collectives revert to their original operation after frequency learning stops.

Programming languages such as Chapel [6], X10 [7] and Habanero [18] embrace dynamic parallelism and load balancing as first class citizens. Their runtime implementation uses either MPI or one-sided communication libraries: we have shown the ability to handle both. The equivalent of group synchronization in these cases is the "`finish`" of a parallel region. We believe our variability based approach provides the right framework for these new languages: per core behavior is likely to be unpredictable and measuring dispersion of a `finish` construct is likely to uncover the DVFS potential during its execution, without need for tracking communications or sampling performance counters.

Recognizing the shape of variability is useful in other optimization scenarios. One can easily imagine building other global control algorithms, e.g. communication or I/O congestion avoidance schemes.

Comparing the three generations of hardware, margins are diminishing on the newer machines due to better hardware power management. Furthermore, cores share more and more resources which are not only outside the control of software DVFS, but also remain at higher clock speeds as long as some of the cores that share them are not scaling down. Scaling individual cores has a much smaller impact and only a global assignment achieves the desired energy savings, as illustrated by Figure 11.

8. Conclusion

To ensure scalability, developers spend significant effort to load balance their codes. Previous DVFS-based energy optimizations have been successful at handling statically load balanced codes using two sided communication and relied on the predictability of runtime behavior. In this paper we propose a qualitative approach designed to handle codes using dynamic load balancing whose behavior is hard to predict. Our main insight is that we can recognize the statistical signature of executions that can benefit from DVFS. Using variability and skewness as primary indicators for proclivity for execution at low frequency, we develop DVFS control algorithms. A good feature of our algorithm is that it can compute a unique system-wide frequency assignment, and it therefore eliminates the limitations of previous work that relies on per core DVFS control. Our evaluation for NWChem shows that our online approach is able to provide good energy savings at high concurrency in production runs, with little runtime overhead. The method is built using pragmatism and we believe that the variability criteria is likely to be useful in building global control mechanisms besides energy.

References

[1] Cori. https://www.nersc.gov/users/computational-systems/cori/.

[2] The cray xc30 network.

[3] A power aware runtime. Available at https://github.com/scalability-llnl/Adagio.

[4] The trinity advanced technology system. http://www.lanl.gov/projects/trinity/specifications.php.

[5] D. Brodowski. Linux cpufreq governors. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[6] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl*, 63(3):291–312, 2007.

[7] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10), Oct. 2005.

[8] ComEx: Communications Runtime for Exascale. http://hpc.pnl.gov/comex/.

[9] 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*, MICRO-45, pages 143–154, Washington, DC, USA, 2012. IEEE Computer Society.

[10] Edison. https://www.nersc.gov/users/computational_systems/edison/.

[11] GASNet: Global-Address Space Networking. http://gasnet.lbl.gov/.

[12] R. Ge, X. Feng, W.-c. Feng, and K. W. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 18–, Washington, DC, USA, 2007. IEEE Computer Society.

[13] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. In *Proceedings of the 11th Workshop on High-Performance, Power-Aware Computing*, HPPAC '15, 2015.

[14] C. Iancu and E. Strohmaier. Optimizing communication overlap for high-speed networks. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 2007.

[15] D. D. James H. Laros III, Phil Pokorny. Powerinsight - a commodity power measurement capability. In *The Third International Workshop on Power Measurement and Profiling in conjunction with IEEE IGCC 2013*, Arlington Va, 2013.

[16] N. Kappiah, V. W. Freeh, and D. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 33–33, Nov 2005.

[17] D. Kerbyson, A. Vishnu, and K. Barker. Energy templates: Exploiting application information to save energy. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 225–233, Sept 2011.

[18] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar. Habaneroupc++: A compiler-free pgas library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

[19] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[20] J. Li, L. Zhang, C. Lefurgy, R. Treumann, and W. E. Denzel. Thrifty interconnection network for hpc systems. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 505–506, New York, NY, USA, 2009. ACM.

[21] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, Transparent CPU Scaling Algorithms Leveraging MPI Communication Regions. In *Parallel Computing, 37(10-11)*, 2011.

[22] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of cpu frequency transition latency. *Computer Science - Research and Development*, 29(3-4):187–195, 2014.

[23] NERSC. https://www.nersc.gov/.

[24] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy libray for distributed array libraries and compiler run-time systems. In *Proc. of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th Intl. Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing*, 1999.

[25] J. Nieplocha et al. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2), May 2006.

[26] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "power" struggles: Coordinated multi-level power management for the data center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 48–59, New York, NY, USA, 2008. ACM.

[27] H. Ribic and Y. D. Liu. Energy-efficient work-stealing language runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[28] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 49:1–49:9, New York, NY, USA, 2007. ACM.

[29] B. Rountree, D. K. Lownenthal, B. R. de Supinski, and M. Schulz. Practical performance prediction under dynamic voltage frequency scaling. In *Proceedings of the International Green Computing Conference and Workshops*, IGCC '11, pages 1–8. IEE, 2011.

[30] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM.

[31] Teller. http://www.sandia.gov/asc/computational_systems/HAAPS.html.

[32] The umt benchmark code. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/umt/.

[33] M. Valiev et al. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[34] A. Vishnu, S. Song, A. Marquez, K. Barker, D. Kerbyson, K. Cameron, and P. Balaji. Designing energy efficient communication runtime systems: a view from pgas models. *The Journal of Supercomputing*, 63(3):691–709, 2013.