Exploiting Variability for Energy Optimization of Load Balanced Parallel Programs

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

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

ABSTRACT

In this paper we present optimizations that use DVFS mechanisms to reduce the total energy usage of the NWChem computational chemistry code. The analyses handle dynamically load balanced, well optimized code in a runtime and programming model independent manner. Our main insight is that noise is intrinsic to large scale executions and it appears whenever shared resources are contended. This criteria, particularly important when using one-sided communication, allows us to identify and manipulate any program regions amenable to DVFS. We validate our approach using offline and online analyses. When compared to previous MPI specific optimizations that make per core decisions, our scheme can determine a "global" system level frequency for any portion of the execution. This suits better the hierarchical nature of DVFS control in current systems and bodes well for eventual coarse grained (cabinet, system) hardware DVFS control in HPC systems.

We have applied our methods to NWChem, and we show improvements in energy use of 8%, at a cost of less than 1% in performance when using online optimizations.

1. INTRODUCTION

Optimizations using Dynamic Voltage and Frequency Scaling (DVFS) have been shown [24, 12, 27, 18, 13, 16] to reduce energy usage in HPC workloads. As we continue to scale systems up and out towards Exascale levels of performance, power becomes the most important system design constraint. There exists a fair incentive to deploy these optimizations in production, but many questions still require answers for the next generation of HPC codes and systems.

The rather impressive existing body of work developed energy optimizations for MPI codes, with the main purpose of saving energy without affecting performance. Optimizations try to exploit the combination of synchronous two-sided `Send`/`Recv` communication together with static domain decomposition and try to predict the "critical path" through the program execution. Ranks executing on the critical path need to run fast, while others can be slowed down.

In order to address hardware and software evolutionary trends, our research tries to expand on the intrinsic assumptions behind these MPI centric methods. We use NWChem [26], which provides open source computational chemistry software for simulations of chemical and biological systems. NWChem is built using a *one-sided* communication model and a *Global Address Space* (PGAS) abstraction, it uses non-blocking communication and overlap for latency hiding, it also provides its own internal *tasking* and *dynamic load balancing* mechanisms. The dynamic, adaptive behavior and hiding communication latency are desiderata for large scale performance that will become pervasive in scientific codes; handling them is mandatory for any future success of energy optimization approaches. We examine several questions:

1. As most codes previously considered used a static domain decomposition which leads to static load (im)balance and repeatable behavior, can we handle codes with dynamic load balancing?
2. As previous methods have been validated only under the assumption that per core DVFS is available, can we develop a control mechanism for the existing hierarchical nature of hardware control?
3. As application benchmarks previously considered use two-sided blocking communication, are there any opportunities in codes with one-sided communication with aggressive overlap?

As many others [24, 18, 13, 17], we exploit the idea that *slack* gives a natural indication of DVFS opportunities and that voltage or frequency can be lowered when tasks are waiting for external events. Unlike the existing tenet that slack is predictable, we argue that for dynamic codes (parallelism or load balancing) we can not make predictions on timings of individual processes because of random variability, but we can predict, and make use of, that variability itself. Our insight is that noise is intrinsic to large scale executions and we can recognize its signature whenever code amenable to DVFS executes. One of our contributions is a method that allows us to identify and manipulate any program regions containing: i) blocking or nonblocking one-sided or two-sided communication; ii) I/O; and iii) DRAM bandwidth limited execution.

Online, single pass context sensitive analyses [24, 18] that are input and concurrency independent are required for scalability and the long term success of DVFS optimizations. We have developed both offline, multi-pass approaches, as well as single pass, online optimizations. To handle the high latency of DVFS control on production systems, we use context sensitive program region classification and clustering algorithms. Our clustering algorithm extends the work by Lim et al [18] to merge multiple clusters for better tolerance of DVFS control latency and greatly increased energy savings.

As previous analyses assume per core DVFS control and compute per core assignments, another contribution of our algorithms is the ability to select a global frequency assignment for highly dynamic codes using either two- or one-sided communication.

We experiment with NWChem using ARMCI [19] one-sided communication, as well as two-sided MPI. We validate results on a small InfiniBand cluster and simulate our algorithms for a large scale Cray XC30 system. The combination of the variability criteria with clustering methods provides us with a programming model and runtime independent approach. When using one-sided communication, both the variability criteria and aggressive region clustering are required for performance. The importance of recognizing variability increases with scale. When using two-sided communication, due to its implicit synchronization semantics less variability is found in the NWChem execution. In this case, clustering reaps most of the optimization benefits.

With the online algorithm, we observe energy savings as high as 8.3% and slowdown of 1.5% for one-sided communication at high concurrency. For two-sided communication we observe energy savings as high as 20% with negligible slowdown. For reference, the offline algorithm is able to almost double the energy savings for one-sided communication, due to its ability to use the optimal frequency for any region cluster in the program. This indicates that we can further tune the online algorithm to increase the energy savings.

2. BACKGROUND AND MOTIVATION

This work has been motivated by the desire to develop effective application and runtime independent energy optimizations for the next generation of scientific codes which are likely to: 1) employ latency hiding and dynamic load balancing; 2) combine multiple runtimes and parallel libraries.

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. As MPI has been the de-facto programming standard for large scale scientific applications, most of the existing approaches [24, 12, 18, 13, 16] are tailored for its `MPI_Send/MPI_Recv` two sided communication semantics. For two-sided operations where communication has data transfer, as well as synchronization semantics, slack captures the network bandwidth and latency together with application load imbalance. Where global synchronization (barriers) operations are concerned, slack captures the application load imbalance.

MPI energy optimizations try to construct a critical path through the program execution and minimize slack: ranks executing on the critical path need to run fast, while all others can be slowed down. Initial studies [23] used offline, trace based analyses that often solve a global optimization problem.

For generality and scalability, later optimizations use online analyses. Rountree et al [24] present Adagio, which uses context sensitive analysis to compute the critical path at runtime. They try to minimize slack for each MPI operation in its calling context and use CPU performance counters to assess the efficacy (slowdown) of DVFS changes. This simple feedback loop allows Adagio to revert unprofitable decisions caused misprediction. In order to minimize the overhead of high DVFS latency, Lim et al [18] present a context sensitive analysis that uses clustering of MPI calls to coarsen the granularity of program regions subject to DVFS. Theirs is a more restricted approach, they try different frequencies for a region/cluster without any feedback, once a decision is made it never reverts. Their goal is to improve load imbalanced applications, without affecting load balanced applications.

We do embrace the notion that demonstrating successful online analyses is mandatory for the future adoption of DVFS techniques, either in hardware or software. Each of these last two efforts [24, 18] employs only pieces of the mechanisms we believe to be needed: 1) ability to handle dynamic behavior; and 2) mitigating high DVFS latency. The challenge is removing their current limitations and combining them in a manner able to handle the evolution of hardware design and software practices.

First, these and many other approaches [24, 18, 22, 16, 12, 23] have been validated under the assumption that per core DVFS is available, using only one core per node or per socket on multicore systems. As modern hardware (e.g. Intel Haswell) allows control only at the socket level, extensions to increase the granularity of hardware control are required. Second, all these techniques rely on the fact that slack is static per rank and region, therefore predictable. Our survey of the application codes in these studies shows a static domain decomposition with "static" load (im)balance. Third, they were evaluated mostly on benchmarks that use blocking `Send/Recv` communication. We have surveyed the application codes used [24, 18, 16, 12, 23] and we have found few to *none* that use non-blocking `MPI_Isend/MPI_Irecv` operations to *overlap communication with computation.*

One-sided communication has been recently embraced as a necessary paradigm to provide application scalability on the next generation of Peta- and Exascale supercomputers. Here communication slack is determined mostly by network performance. The notion of critical path in one-sided communication is also challenging as it may need to be inferred in an application specific manner that considers `Put/Get` operations in conjunction with ad-hoc inter-task synchronization mechanisms. Some large scale scientific codes [26, 2] already use non-blocking or one-sided communication in a very aggressive manner to hide communication latency and attempt to provide dynamic load balancing. One such example is NWChem [26] which can use as a communication transport either MPI two-sided, or one-sided communication with MPI 3.0, ARMCI [19], ComEx [6] or GASNet [9]. The code sends many messages with different sizes, overlapped with computation in a ever changing communication topology. It also contains tasking and dynamic load balancing mechanisms that determine a highly dynamic execution. To our knowledge, energy optimizations on one-sided communication *applications* have not been demonstrated successfully.

Slack and DVFS opportunity also appear in scientific codes whenever I/O operations [10] are performed. To our knowledge the existing HPC approaches are MPI specific and do not target dynamic parallelism or file I/O operations.

Most existing approaches work because inefficiencies (slack) have a static nature and can be predicted. Software and hardware evolutionary trends tend to diminish the potential of such schemes. One-sided communication, communication overlap optimizations, dynamic tasking and dynamic load balancing tend to reduce slack, and some make it even unpredictable. On the hardware side, imbalances in runs at scale are more often caused by over-commitment of resources such as memory bandwidth and network or file access, and these imbalances are stochastic in nature. Finally, the active power management in modern CPUs manipulates frequency and voltage based on load and overall power use. This not only adds a further element of unpredictability, but moreover, it works against software techniques. In our initial

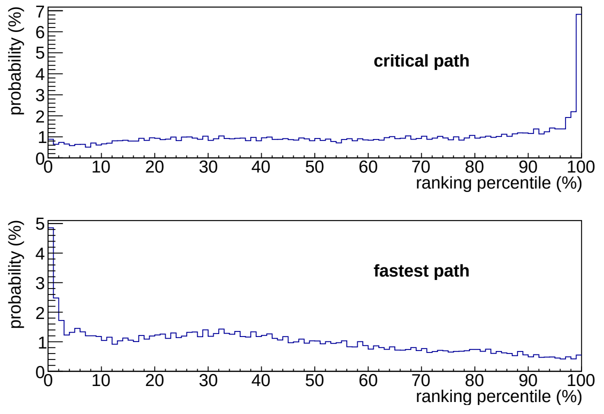


Figure 1: *Quality of prediction of the critical path based on calling context and process in NWChem, for a run of 1024 processes. Shown is the ranking of the critical (top) and fastest (bottom) process, in the subsequent re-occurrences of tasks. The probability of the critical path remaining critical,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

assessment, these considerations almost put paid to the idea that inter-communication timings have predictive value.

2.1 Identifying DVFS Opportunities for Dynamic Program Behavior

Most algorithmic operations in our target application perform a sequence of Read-Modify-Accumulate operations on a global data structure. Each operation is dynamically load balanced using logical tasks and barriers are executed at the end, to maintain data consistency.

Thus, we are interested in our ability to predict the reoccurrence and behavior of the execution between two barriers on any task. We configure NWChem to use one-sided communication and collect the durations of logical tasks between barriers, taken as the difference between the exit time stamp of the previous barrier and the entrance time stamp of the current one. Differences in these durations among processes are commonly referred to as *slack*. In the rest of this paper, a *region* refers to the code executed between the exit of a barrier until the exit of the next barrier. A region thus includes the algorithmic operations, any slack, and the communication time of one barrier. The DVFS opportunities of communication are obvious, and slack is a derived measure. Thus, we can focus our analysis on the duration of the algorithmic operations, then extend the conclusions to the whole region in a straightforward manner.

We run NWChem, and for each region, we sort the processes by duration of their logical tasks, label them according to their "ranking" in this, and collect these rankings per process and calling context (described in Section 3). We select only those regions that are long enough (at least $300\mu s$[1]) 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, meaning that there is sufficient repetition for meaningful predicting/learning and for scaling to have an effect. 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 1024 processes.

[1]This is about 3× the latency of DVFS control.

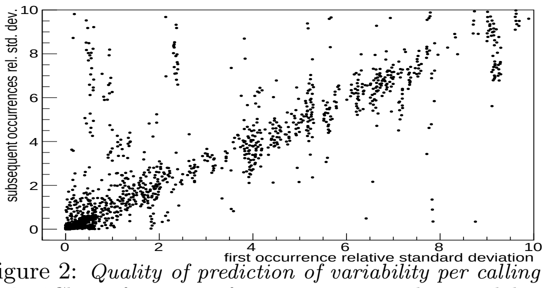


Figure 2: *Quality of prediction of variability per calling context in NWChem, for a run of 1024 processes. The variability of a reoccurring calling context is highly correlated with the variability of the first occurrence of that context, leading to good predictivity.*

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 critical or 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.

Still, mispredicting the critical path only leads to missed opportunities. But mispredicting the fastest path, i.e. the process that will be scaled down the most, can have dire performance consequences. We apply the same analysis to the fastest process for each context as we did for the critical one, and the results are in the bottom of Figure 1. The fastest path then, is just as hard to predict as the critical.

We conclude from this data that schemes relying on predicting the per rank duration of execution are likely to fail for our target application.

2.2 Employing Variability as a Predictor

In this paper, we will argue that the tables can be turned around: we can not make predictions on timings of individual processes because of random variability, but we can predict, and make use of, that variability itself. We conjecture that besides software causes such as dynamic load balancing, variability is caused by the intrinsic nature of the system and the computation. Identifying these causes, determining their variability "signature" and their amenability to DVFS optimizations, allows us to build energy optimizations for dynamic program behavior. 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.

Predicting Variability: Figure 2 shows the results of comparing the variability 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 variability increases than decreases when contexts reoccur. This makes the *presence* of variability a good indicator. Its magnitude clearly also has predictive value, but we will not use that in our analysis.

Causes of Variability: We use microbenchmarks that time code executed in between two barrier operations to understand where and how variability appears. The code is either communication, memory, I/O, or compute intensive, and each rank performs the same amount of work, i.e. the workload is seemingly load balanced. As previous work on CPU hardware level energy optimizations, indicates that memory intensive [7] codes are amenable to DVFS, we distinguish between Flops-limited and DRAM bandwidth-limited code in the computation benchmarks.

Intuitively, *pure* computation is expected to be more predictive and static in nature because the total number of processes is equal or less than the number of CPU cores available, while communication, synchronization and I/O are prime culprits for variability, therefore prime candidates for DVFS, since there processes have to share limited resources. As shown in Figure 3, in CPU-limited code the variation in execution time, measured as a standard deviation, is negligible at any scale, while the variation in DRAM bandwidth-limited code is more than 10× larger across the board. The shape of the distributions, measured with skewness, gives an extra distinction at scale: although memory limited code remains mostly normally distributed, CPU limited code grows a significant[2] right-side tail.

For brevity we omit detailed results for communication and I/O intensive codes, we note that their behavior is qualitatively similar to memory intensive codes.

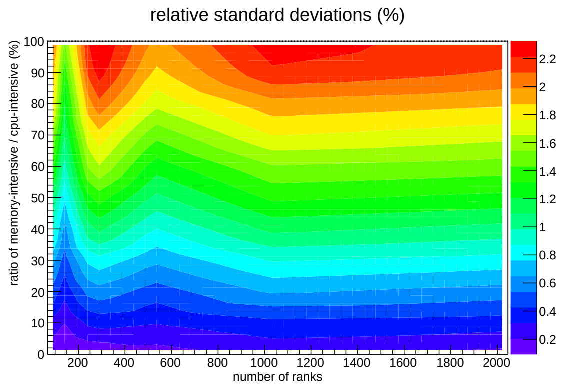
Selecting DVFS Candidates: The previous results indicate that the type of behavior amenable to DVFS (communication, I/O, memory intensity), also causes variability (variable load imbalance in our case). But from there we can not simply conclude that variability in NWChem allows us to select regions for DVFS, as regions with divergent logical tasks or different size workloads will obviously show variability as well. We must also show that our criteria handle those cases in a way that does not hurt the end energy saving goals or program performance.

While variance measures how far ranks have diverged from each other through random effects, skewness gives an indication of the shape (asymmetry) of the distribution, and thus an estimate of the behavior of the critical path (which in the case of true divergent logical tasks would not be random). Negative skew means the mass of the distribution is concentrated towards the right of the value range. A negative skew for the duration samples 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.

Memory-limited code uses many components of the hardware: the CPU, the full memory hierarchy of caches, memory controllers, the bus, and the DRAM chips. 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 Figure 3 (bottom). In contrast, variability in CPU-limited code comes from a single source, which leads to an asymmetric distribution with large positive skew, caused by a few stragglers.

Clearly, we want to exclude regions with large positive skew: because it could indicate a CPU-bound region, but also because it could indicate the existence of a determin-

[2]As normalized by the standard deviation; the tail is not significant in an absolute (i.e. wall time) sense.



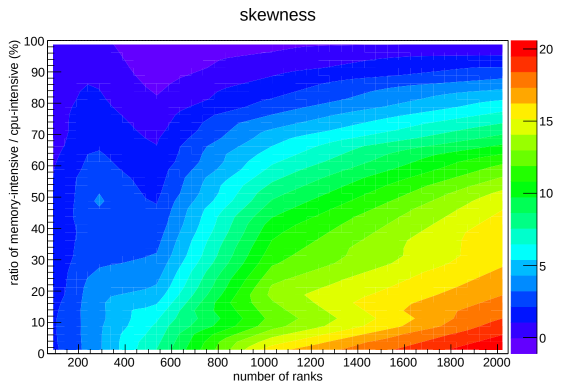
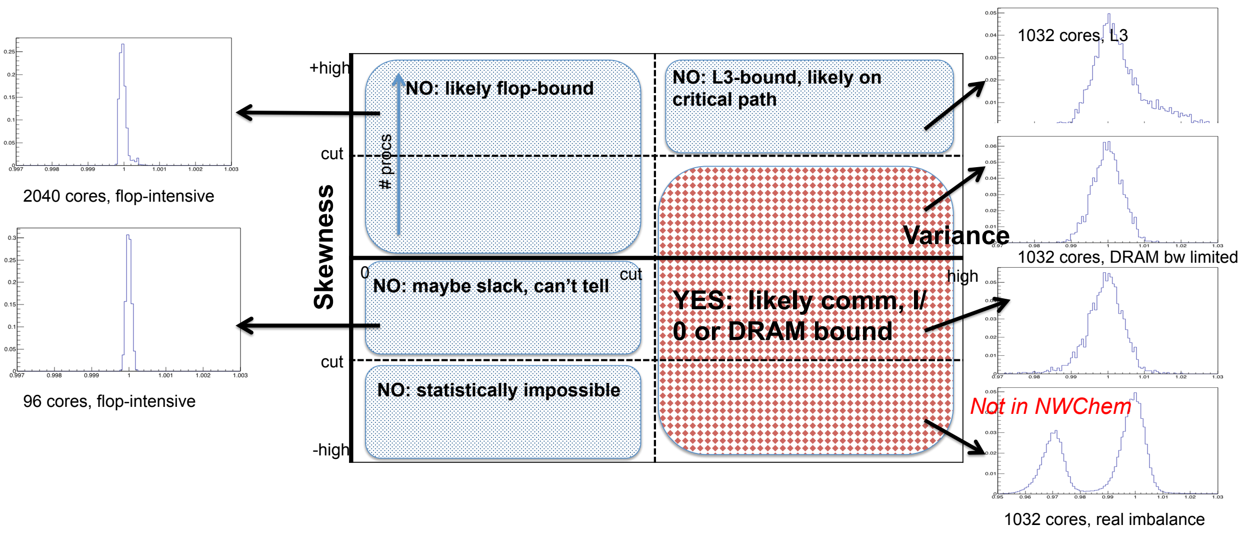


Figure 3: *Variability measures for CPU and memory bound parallel computations: standard deviation (top) and skewness (bottom) of the resulting distributions across ranks. The bottom of each figure is fully CPU intensive, mixing in memory bandwidth limited code until fully memory intensive at the top. Results are shown for increasing number of processes, left to right. Memory bound code has higher variability, but lower skewness, and the differences become greater at scale.*

istic critical path. Either case would see a large slowdown in performance if DVFS were applied. We also want to exclude large negative skew: unless the random variations are (much) larger than the region duration itself, they will not cause a significant negative skew, because the distribution is bound on the left by the minimum duration of the task. But a deterministic imbalance, with some tasks consistently faster, would cause a large negative skew.

Figure 4 summarizes our strategy of applying DVFS based on observed variance and skewness for the case of NWChem. For any observed execution falling in *Tile* **(a)** *we apply DVFS*. The high variance (larger than a threshold), small positive skew or negative skew are characteristics of communication, I/O or DRAM bandwidth limited codes. We do not apply a cut on large negative skew, because NWChem is well load balanced, so it is not needed. For any other execution summarized by Tiles **(b)**, **(c)**, **(d)** and **(e)** *we do not attempt DVFS*. For brevity, we give only the intuition behind our decision, without any benchmark quantification. Tile **(b)** captures code that executes in the last level of shared (contended) cache or code with a critical path. Tile **(c)** captures flop-bound code. In this case note that skewness increases with concurrency. Tile **(d)** captures code with real load imbalance, but we cannot determine where. Finally code in Tile **(e)** is unlikely to occur in the wild.

Note that it would be possible to use hardware performance counters and instrument system calls to get more



2040 cores, flop-intensive
96 cores, flop-intensive
+high
cut
cut
-high
Skewness
# procs
NO: likely flop-bound
NO: maybe slack, can't tell
NO: statistically impossible
NO: L3-bound, likely on critical path
YES: likely comm, I/0 or DRAM bound
0
cut
high
Variance
1032 cores, L3
1032 cores, DRAM bw limited
Not in NWChem
1032 cores, real imbalance

Figure 4: *Classification by skewness and variance. Timings obtained using microbenchmarks at the concurrency indicated in Figure. 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. Real, consistent, workload imbalance exhibits large negative skew with high variance, so it is possible to cut on it. However, such imbalance does not happen in NWChem.*

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 code, 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.

3. DESIGN AND IMPLEMENTATION

We develop single-pass, online optimizations, as well as offline optimizations. Both attempt to exploit 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. The online optimization is desired for scalability, while the offline analysis is more precise and can also be used for tuning of the online optimization engine.

Region Partition: We have started this study by first instrumenting barriers and `Put/Get` operations and understanding the dynamic behavior of NWChem. While the code uses heavily non-blocking communication overlapped with independent computation, barriers are inserted between phases to maintain data consistency. For NWChem, instrumenting at the barrier level captures enough, logically separate, program regions with DVFS potential.

To capture the dynamic nature of the execution we use a context sensitive approach where we identify regions with a hash, created from the return addresses, at the preceding barrier.

Frequency Selection: The number of frequency trials to determine an optimal value directly influences the efficacy of our online algorithm. As we are interested in a global frequency assignment, we select one frequency per trial. Choosing the wrong frequency will unnecessarily slow down during learning, trying too many will miss occurrences of regions that are good candidates. To make online learning faster, we use the offline version of our technique to understand the impact of different frequencies on the calibration microbenchmarks and NWChem performance.

We find that for the online algorithm we only have to consider two choices: the highest (or default frequency) for normal execution is a given; for regions where we switch to a lower frequency, an intermediate roughly in the between the maximum and the memory frequency works well. Regions that are fully memory bound can be run at the memory frequency, ones that are fully communication bound at the lowest. But most DVFS regions are somewhat mixed, leading to the intermediate choice. This is also substantiated by Austin et al [3] which shows that the energy-optimal frequency for a code that is mostly, but not wholly, memory-bandwidth limited is such an intermediate frequency.

Tuning Parameters: The algorithm choices are guided by several other parameters. To amortize the latency to change DVFS on the system, we consider as candidates only portions of code that are longer than a multiple ($5\times$) of this latency. The variability (standard deviation) and skewness thresholds we determine to first order with microbenchmark data such as shown in Figure 3. However, the thresholds cut the same way, so it is possible to create a trade-off. This is important, because the gain of being right is lower than the cost of being wrong. In practice therefore, we choose a tighter skewness threshold with a somewhat looser variability threshold. Finally, the thresholds vary moderately with concurrency and we use samples at different system sizes.

3.1 Region Selection and Clustering

To estimate the variation, we replace `barrier` calls with `allgather`, using linker wraps, send the execution times of all processes, allowing each process to calculate a standard deviation and skewness. A region is then considered a candidate for frequency scaling if these three criteria are met:

- The total region duration most be more than a threshold, $1ms$ in our experiments.

- The variability in the execution time across the relevant processes, must be more than 3% of the average duration of the execution.

- Skewness has to be less than 3.

As noted, each of the numeric parameters above parameters is tunable, and depends on the job size (e.g. variability increases with scale, and decreases for smaller concurrency). But in practice, we don't deviate all that much from the basic values above, because of clustering.

The purpose of clustering is to create larger regions to scale down, and thus save on switching overhead. There are three principle components to clustering: creating clusters of regions where all regions meet the necessary criteria; extending these clusters with regions of short duration which can be assumed to be communication bound even as they are not otherwise selected; and accepting a loss of performance to cover a gap between clusters, as switching would be even more expensive. Clustering for tolerating DVFS latency has been first proposed by Lim et al [18], but they do not extend or merge clusters.

Each of these components is tunable. A `THRESHOLD` sets the minimum size that shorter regions have to meet to form a cluster. This is the same $1ms$ mentioned above, as applied to single regions. 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. A `SHORT` cutoff on region duration is used, under which shorter regions (assumed to be dominated by communication) can be collected to extend a cluster, even if these small regions themselves do not meet the selection criteria. Finally a `BRIDGE` cutoff is used to allow clusters to combine to super-clusters, where the cost of the bridge is acceptable compared to the cost of switching frequencies twice. A bridge is build until it gets too large, after which normal clustering restarts, or reaches a new cluster.

Algorithm 1: Clustering of regions

```
Input: $context$, $region$, and $is\_noisy$
Result: program state update

/* decisions is a dictionary of <context, DVFS decision>, State is program state and
   either CLUSTERING, SCALING, or BRIDGING                                          */

/* SHORT, STDEV, SKEW, and BRIDGE are tunable parameters, see text                  */

if $is\_noisy$ then
    /* collect this region into a cluster                                           */
    COLLECT($context$, $region$)
    return
else
    /* reject, unless very short, or not too long and in between clusters           */
    if SHORT $<$ $region$ then
        if $State$ == SCALING then
            /* potential end of previous cluster, start of a new bridge             */
            $cluster \leftarrow 0$
            $State \leftarrow$ BRIDGING
        if $State$ == BRIDGING then
            BRIDGING($context$, $region$)
            return
        RESET_LOCAL_STATE( )
    else
        /* short region:  communication dominates, collect anyway                   */
        COLLECT($context$, $region$)
    return

procedure COLLECT($context$, $region$)
    if $State$ == BRIDGING then
        /* (potential) start of a new cluster                                       */
        $cluster \leftarrow 0$
        $State \leftarrow$ CLUSTERING
    switch $State$ do
        case CLUSTERING
            $cluster \leftarrow$ SUM($cluster$, $region$)
            $local\_decisons[context]$ = SCALE
            if THRESHOLD $<$ $cluster$ then
                /* cluster has grown large enough                                   */
                $State \leftarrow$ SCALING
                SCALING($context$, $region$)
        case SCALING
            SCALING($context$, $region$)

procedure SCALING($context$, $region$)
    if $pending\_decisions$ then
        $decisions.update(local\_decisions)$
        $decisions.update(bridge\_decisions)$
        RESET_LOCAL_STATE( )
    $decisons[context]$ = SCALE

procedure BRIDGING($context$, $region$)
    $cluster \leftarrow$ SUM($cluster$, $region$)
    $bridge\_decisons[context]$ = SCALE
    if BRIDGE $<$ $cluster$ then
        /* distance from last cluster has grown too large                           */
        RESET_LOCAL_STATE( )

```

The actual values chosen need not be highly tuned, and we don't change them with scale. The reason is interaction of the different parts of the algorithm. For example, choose the `SHORT` cutoff too tight, and bridging will cover most cases anyway. Likewise, the purpose of collecting short regions is not so much to increase the size of clusters (short regions, by definition, have little impact in overall energy savings or program performance), but rather to allow building longer bridges in parts of the program that are dominated by communication, and hence to save on frequency switching. We choose the `SHORT` threshold to be 5× the communication cost a barrier, and the `BRIDGE` cutoff to be 10× the cost of DVFS latency (i.e. the expected average bridge size would be half that, thus balancing the cost of DVFS against the cost of scaling a fully CPU-bound region).

3.2 Online and Offline Algorithms

In the online analysis we start executing the instrumented program and data is collected at each barrier operation. At the second and subsequent occurrences of any candidates, the algorithm chooses a different global frequency to try. As noted, we use only two target frequencies. The decision to scale is made at the second occurrence. The time is again collected after the execution and if a too large slowdown, determined by the ratio of the frequencies plus a margin, is seen in the average time across all ranks, the region will have to re-enter the decision process and may be reverted. Since the average is used, all processes will arrive at the same decision. If the decision gets reverted, that would apply to the next occurrence.

In the offline analysis, we run the same algorithm, but allow the decision to scale to apply retro-actively. Thus, contexts that occur only once, most importantly initialization, and the first iteration of a repeating context can scale as well in offline. Further, the offline analysis can select the most energy-optimal candidate, given a constraint on performance loss, from a static set of frequencies. Thus, the offline analysis gives us an indication of how much optimization is unexploited by the more conservative online optimization.

4. EVALUATION

We evaluate the efficacy of our optimizations on the NWChem application described in Section 4.2. We compare the online and offline optimization approaches on a small cluster and simulate results at scale using 1,034 cores, as described in Section 4.1. For completeness we have attempted to compare against MPI based approaches, Adagio [24] and the clustering [18] algorithm. The results are described in Section 4.6.

4.1 Experimental Methodology

Platforms: One experimental platform is the Teller [25] cluster with four AMD A10-5800K processors and 16GB main memory per node. There are seven available frequencies for scheduling, ranging from 1.4 GHz to 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` [1] library. The other platform is the Edison [8] Cray XC30 system at NERSC, 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.

Methodology: On both systems we are interested in at-the-wall power consumption, this being the ultimate indicator of any energy savings. We use micro-benchmarks to determine the DVFS latency to calibrate our optimizations. On Teller we use the PowerInsight [15] interface to collect power data, which is integrated with the application level time stamps to yield energy consumption. The DVFS switching latency is $\approx 100\mu s$. Results reported for Teller are from actual runs when applying our optimizations.

On Edison power is measured with by the Cray power monitoring counters, which sample energy with a frequency of about 10 Hz, and which can be read from a virtual filesystem under /proc. As Edison is a large scale production system, the only DVFS control allowed is at job startup time, when a single frequency can be selected. Therefore, our results on Edison are obtained through simulations. As a baseline we consider the application running unmodified and with the system default policy. This nominally runs at the highest available frequency, but also allows the hardware to clock down for energy savings, or up (turbo-boost) for performance. We run the application at selected frequency steps, statically requested at start-up. To account for optimization overhead, the application is linked with our instrumentation library, which replaces barriers with collective communication and performs the analysis as if scaling could occur. For any frequency, we build trace files obtained by averaging on each task the duration of a region across at least five runs.

To simulate the offline algorithm, we assume the application starts with the default system behavior and implement the algorithm already described, using the trace files. We process the traces, form clusters and select the best frequency for each cluster. Note that allgather overhead is already included in the execution and during the simulation we add a configurable DVFS change overhead as necessary. Each cluster and region in the resulting program is "run" at the frequency indicated by the simulation, as if a priori known. For the online algorithm, we use only one target frequency, and run the simulation with the extra DVFS latency added.

As an extra validation step for the simulation results, we have compared the clusters selected by the simulation for Edison with the clusters selected by the online algorithm when running on Teller at similar concurrency. There is a very high overlap in selected regions, which confirms that we do indeed select during simulation the portion of the execution amenable to DVFS.

4.2 NWChem

NWChem [26] delivers open source computational chemistry software using Coupled-Cluster (CC) methods, Density Functional Theory (DFT), time-dependent DFT, Plane Wave Density Functional Theory and Ab Initio Molecular Dynamics (AIMD). Due to its ability to run from small (laptop) to very large scale, NWChem supports a vibrant scientific community with impressive research output. The code base contains more than 6 Million lines of code written in Fortran, C and C++ implementing state-of-the art programming concepts mandatory for large performance. The main abstraction in NWChem is a globally addressable memory space provided by Global Arrays [20]. The code uses both one-sided communication paradigms (ARMCI [19], GASNet [?], ComEx [6], 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 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.

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 that perform message unpacking and Accumulate operations, driven by an interrupt based implementation. With GASNet [9] as a transport layer, progress is ensured by polling and accumulating inside the operations that wait for message completion. Accumulates are implemented using an Active Messages based approach.

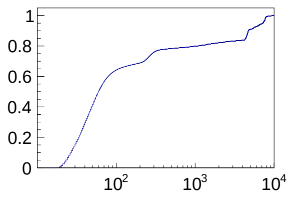
We have experimented with the MPI back-end in order to provide a comparison with other MPI tools. Our optimizations were able to handle the MPI back-end with good results, since the overhead of communication is higher than in the ARMCI case. For brevity we concentrate presenting ARMCI results, see Section 4.6 for MPI. The GASNet back-end is still in experimental stage and we are working on finishing its implementation, under a separate project. For the final version of the paper we plan to add GASNet based results, which can give us an interesting comparison point between the energy efficiency of polling and progress thread based network attentiveness schemes. However, note that the ARMCI back-end is more challenging due to its asymmetric runtime configuration.

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 different 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. We examine each method at increasing concurrency: 132 cores, 528 cores and 1034 cores scale (22 per node), as they exhibit different dynamic behaviors. For DFT, we simulate at 120 cores (20 per node). DFT execution is computation intensive.

4.3 Dynamic Behavior of NWChem

Figure 5 shows the cumulative distribution function (CDF) of the total region duration and imbalance for a 1034 cores run of NWChem. Imbalance (or slack) is computed as the difference in execution between the fastest and the slowest rank to enter a barrier. Slack is virtually non-existent, as a result of dynamic load balancing in NWChem. As indicated, 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.

When executing the program on Edison with different frequencies, the default system assignment provides the most performant execution at good energy efficiency. For reference, a static frequency assignment of 2.4 GHz is nominally the highest possible frequency, but setting it explicitly will switch off turbo-boost and DVFS by the hardware. In comparison to straight 2.4 GHz, the default assignment gains 8% in performance at a cost of 13% in energy, with most of that cost incurred during a single, long, region in initialization (and thus not available to recoup in an online analysis, which requires repetition).


(a) CDF of regions

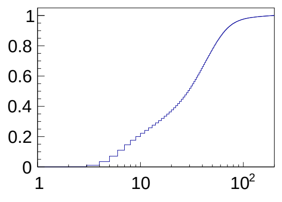

(b) CDF of slack times

Figure 5: *Region and slack time cumulative distribution functions, for 1034 cores, coupled cluster.*

4.4 Impact of Algorithmic Choices on NWChem Performance and Energy

On Teller, we can run our optimizations and perform DVFS at runtime using up to 128 cores. For brevity, we do not discuss these results in detail. The Edison CPUs have much better hardware power management and energy optimizations on the Cray platform 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.

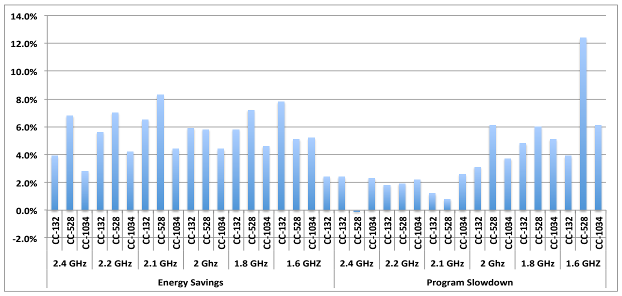


Figure 6: *Summary of results on Edison for CC running at increasing concurrency (132,518,1034) and with different target low frequency for the online algorithm.*

In the rest of this section we will use simulation results from the Edison 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. The typical execution is on the order or 20 minutes or more.

The execution of CC-132 is dominated by memory intensive computation 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% at the expense of a 1.5% slowdown. Low frequency is selected for about 23% of the execution, using $\approx 70,000$ switches. Although the overhead of DVFS switching is higher for this benchmark, the runtime overhead is caused mostly by our algorithm that replaces barrier operations with `Allgather` and accounts for roughly 1% slowdown.

The large scale runs illustrate that our design choice to compute a unique system-wide frequency carries inherent overhead. We have considered optimizations to improve performance by reverting `Allgather` back to barriers after online learning finishes. Due to the code structure in NWChem, this is non-trivial as barriers in the code are textually unaligned and contexts that have been decided may align with new contexts on different ranks. We believe to have a solution, full implementation not finished at the time of the writing.

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.

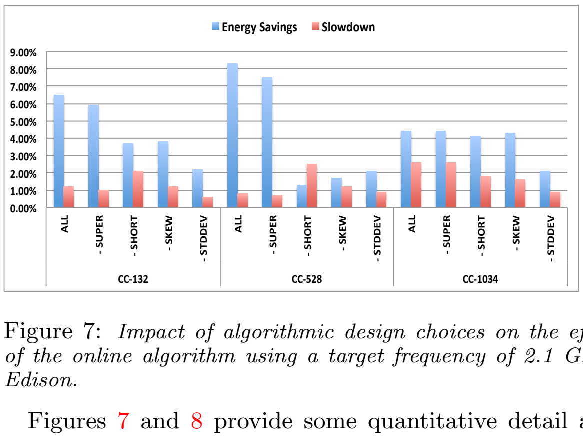


Figure 7: *Impact of algorithmic design choices on the efficacy of the online algorithm using a target frequency of 2.1 GHz on Edison.*

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 cri-

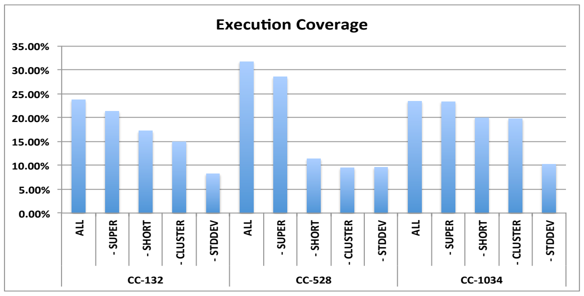


Figure 8: *Percentage of the execution time at low frequency as determined by algorithmic choices on Edison using a target of 2.1 GHz.*

teria 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 an approximation of the clustering algorithm presented by Lim et al [18].

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.

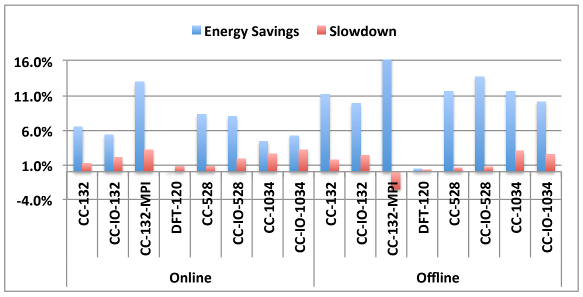


Figure 9: *Comparison of energy savings and slowdown for CC, CC-IO, DFT and CC-MPI when using online and offline optimizations on Edison with a target low frequency of 2.1 GHz.*

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 2.2 our approach does not identify many regions as DVFS candidates. MPI results are further discussed in Section 4.6.

4.5 Optimality of Online Algorithm

In order to converge rapidly, 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: any region could be lowered to any given frequency given a priori knowledge. In Figure 9 we include for reference the energy savings and the slowdown for the offline trace based optimization approach that uses a similar clustering and selection criteria, but it can choose 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. Note that in the offline approach `Allgather` operations are not necessary in the transformed program. This indicates that any method to improve the target frequency selection for a cluster is likely to improve the energy savings provided by our approach. We have started to consider several approaches.

4.6 Impact of Communication Paradigm

The results presented so far have been for NWChem configurations using one-sided communication. Our method can handle both one- and two-sided communication codes, for brevity we offer a summary of our findings when running NWChem configured to use two-sided MPI as a transport.

We have also attempted to compare against MPI based approaches, Adagio and the clustering implementation in [18]. As both handle only per core DVFS decisions, we have attempted first running with one MPI rank per socket. Adagio aborted on NWChem, due to its large number of `ISend/Probe/Wait` operation count. The clustering used by Lim et al can be directly applied to our benchmarks.

First, the application runs much slower, and for our configurations we have observed more than 4× slowdown on Edison when comparing MPI with ARMCI performance. A large contribution is attributed to slower communication. 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/Wait` operations performed between barriers.

When compared with the one-sided configuration, we observe much higher energy savings (up to 20%) when using MPI for communication. 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 validates the design of the algorithm presented by Lim et al [18], as it provides most of the energy savings available. 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%.

5. DISCUSSION

We believe that our approach works for codes that are load balanced as a result of static decomposition or dynamic mechanisms. Programming languages such as Chapel [4], X10 [5] and Habanero [14] embrace dynamic parallelism and load balancing as first class citizens. Their runtime implementation uses either MPI or one-sided communication libraries. The equivalent of a barrier in these cases is the "`finish`" of a parallel region. We believe our variability based approach provides the right framework for these new languages.

At the application level, instrumenting point-to-point communication calls (`Put/Get`, `Isend/Irecv`, I/O) provides an easy partition of the program execution into regions amenable to per core DVFS. At the other end of the spectrum, instrumenting global or group synchronization calls (barriers or collective operations) provides for easy partitioning into regions amenable to hierarchical system control. Due to the need for enforcing data dependencies, turns out that programmers tend to insert enough global synchronization even in heavily asynchronous codes. Were global or group synchronization not present, the question remains how to generalize our approach to different hardware DVFS control granularities.

During this research, we have striven to provide hardware and runtime independent mechanisms and we have tried to eschew the use of hardware performance counters. These may be used by any libraries and hijacking them hampers portability in large code bases. Furthermore, measuring hardware events between communication calls is misleading: polling, I/O, asymmetric execution inside progress threads in NWChem get misclassified without very detailed changes across the whole software stack.

Finally, we emphasized practicality and did not attempt to build optimal approaches. By refining the instrumentation level, specializing for a particular implementation or refining the granularity of control, there may be more potential for energy savings in NWChem.

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* [7] has been used to identify DVFS opportunities. To our knowledge, approaches similar in spirit are implemented in hardware in the Intel Haswell processors. Existing HPC techniques [24] consider computational intensity (instructions per cycle or second) as a measure of application speed under DVFS. As busy waiting, polling and remote invocations are widely used now inside implementations, it may be more challenging to attribute hardware cycles to software constructs.

Raghavendra et al [21] discuss coordinated power management for the data center level. They present a hierarchical control infrastructure for capping peak or average power. Their detailed simulation illustrates the challenges of tuning these schemes even for small scale systems.

In the HPC realm, most optimizations use slack in MPI as their optimization criteria. Initial studies [23] used offline, trace based analyses that often solve a global optimization problem using linear programming. In particular, these analyses do not handle non-blocking MPI communication `Isend/Wait`. 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.

Later approaches improve scalability using online [24, 18] analyses and mitigate high DVFS latency using clustering [18] techniques. We have already highlighted the differences and extensions allowed by our approach.

Other approaches that consider hybrid MPI+OpenMP codes [16], 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. The strategy preferred strategy for the OpenMP regions seems to be turning cores off completely. Extensions to handle codes with more dynamic parallelism or when running a parallel region over multiple clock domains are required.

Programming model independent approaches tend to be used by hardware or only within the node [7] 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 [11] employs this technique. Application dependent approaches identify and annotate algorithmic stages and iterations, using either online (Jitter [12]) or offline [13] 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 [27] using the ARMCI implementation. They design specific interrupt based mechanisms for the ARMCI progress thread and lower voltage when waiting for communication completion. The evaluation is performed using micro-benchmarks that perform blocking communication, e.g. put followed by fence with no overlap, and with per core DVFS. The ARMCI implementation of NWChem performs overlapped non-blocking communication.

Recent work by Ribic and Liu [22] describes the design of an energy efficient work-stealing 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*, the evaluation is performed running with only one core per domain, to avoid interference between decisions made on each core.

7. CONCLUSION

In this paper we have explored DVFS optimizations for codes using one-sided communication and dynamic load balancing. Previous successful approaches developed for two-sided MPI rely on the predictability of code behavior. Our work indicates that while behavior is not really predictable for NWChem and one-sided communication, we can recognize the statistical signature of portions of the execution amenable to DVFS. Using variability and skewness as primary indicators for a region's proclivity for execution at low frequency, we develop an online control algorithm based on region clustering. A distinguishing 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. We also provide a comparison with an offline trace based optimization, that uncovers even more energy savings potential.

8. REFERENCES

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

[2] Namd scalable molecular dynamics. http://www.ks.uiuc.edu/Research/namd/.

[3] 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*, E2SC '14, pages 51–59, Piscataway, NJ, USA, 2014. IEEE Press.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl*, pages 1094–3420.

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

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

[7] 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.

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

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

[10] R. Ge. Evaluating parallel i/o energy efficiency. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 213–220, Washington, DC, USA, 2010. IEEE Computer Society.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. H. Laros, P. Pokorny, and D. DeBonis. Powerinsight-a commodity power measurement capability. In *Green Computing Conference (IGCC), 2013 International*, pages 1–6. IEEE, 2013.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.