Green Queue: Customized Large-scale Clock Frequency Scaling

Ananta Tiwari Michael Laurenzano Joshua Peraza Laura Carrington Allan Snavely
Performance Modeling and Characterization Laboratory, San Diego Supercomputer Center
University of California, San Diego, California 92093–5004
tiwari@sdsc.edu michaell@sdsc.edu jperaza@eng.ucsd.edu lcarring@sdsc.edu allans@sdsc.edu

Abstract—**We examine the scalability of a set of techniques related to Dynamic Voltage-Frequency Scaling (DVFS) on HPC systems to reduce the energy consumption of scientific applications through an application-aware analysis and runtime framework, Green Queue. Green Queue supports making CPU clock frequency changes in response to intra-node and inter-node observations about application behavior. Our intra-node approach reduces CPU clock frequencies and therefore power consumption while CPUs lacks computational work due to inefficient data movement. Our inter-node approach reduces clock frequencies for MPI ranks that lack computational work.**

We investigate these techniques on a set of large scientific applications on 1024 cores of Gordon, an Intel Sandybridge-based supercomputer at the San Diego Supercomputer Center. Our optimal intra-node technique showed an average measured energy savings of 10.6% and a maximum of 21.0% over regular application runs. Our optimal inter-node technique showed an average 17.4% and a maximum of 31.7% energy savings.

I. Introduction

The electricity costs of operating HPC systems and data centers have increased by 36% in the last 5 years (from 7 gigawatts to almost 10 gigawatts total in the United States), according to Jonathan Koomey's 2011 report [1] on recent electricity growth within data centers. This growth was actually dampened by the global recession; it had been forecast by the Environmental Protection Agency to double [2] and it is very likely to double within the next 5 years as the economy rebounds and the Department of Energy enters the "exascale" era of supercomputing. To help reduce these energy costs, we present Green Queue, a scalable scheduler deployed on Gordon [3] at the San Diego Supercomputer Center in which users and the facility operators work together to save as much as 31.7% of the operational energy bill. Green Queue utilizes application behavior and hardware configurables to discover customized energy efficient runtime configurations specifically for an application. It then deploys the application onto the machine and utilizes these runtime settings to run the application in a way that uses less energy.

Green Queue is based on a series of application-aware Dynamic Voltage-Frequency Scaling (DVFS) techniques for use within MPI applications. DVFS is a technique whose purpose is to lower the power draw of the computing system by putting the processor(s) of the system into lower frequency-voltage states. However, the undesirable side effect of lowering the clock frequency is a potential loss in performance when that lower clock frequency results in the delay of something on an application's critical path. This implies that there is a careful balance which must be struck between lowered power draw and loss of performance in order to achieve reduced energy consumption. Understanding this tradeoff requires understanding the complex effects that memory access patterns, computational behavior, load imbalances in a parallel job, and changes in CPU clock frequency have on performance and power draw. Our approach to gaining this understanding consists of a combination of lightweight static analysis and runtime tracing to automatically acquire characterizations of the patterns of execution for computational bottlenecks and load imbalances within an application. We then use this information to develop a customized DVFS strategy that is prepared, stored in a database and deployed on the application each time it is seen by Green Queue.

Through Green Queue we take a two-pronged approach to determine the customized strategy of CPU clock frequency for an application. First, for applications with good load balancing we attempt to increase energy efficiency with an intra-node clock frequency selection mechanism. This method utilizes a detailed, full characterization of the application to logically partition the application into *phases* of behavior; phases serve as the basic unit in the application for which clock frequency decisions will be made. For the application's phases, we query power consumption models for the system which yield the system-wide power draw for individual phases of the application at all available frequencies within the hardware. We then combine the power draw models with execution time data/models to transparently map application phases to their optimal clock frequency settings. Second, for applications with poor load balancing our strategy seeks to shift the clock frequency on the hardware running MPI ranks that have less work to do than their counterparts. These shifts are based on timing measurements of the MPI behavior of the application and reduce the power draw for the hardware running those ranks which appear to be off the critical path for the application and should therefore have little impact on overall runtime, saving power at little cost to performance.

We test the effectiveness of Green Queue on 1024 cores (a full rack) of Gordon, a novel Intel Sandybridge-based supercomputer at the San Diego Supercomputer Center. Using rack-level AC power measurements that are collected for the entire system and encompass all power drawn by the system, we show substantial energy benefits on a host of real scientific applications that are run using the Green Queue.

II. Related Work

DVFS as a technique for increasing the energy efficiency of HPC systems and applications is relatively well-studied. For the sake of clarity, we distinguish two categories of related work: *intra-node* and *inter-node* DVFS techniques for reducing energy consumption.

A. Intra-node DVFS

Intra-node DVFS schemes take advantage of the periods during an application run during which the performance of the application is to some degree independent of the CPU clock frequency running the application; periods which usually correspond to times when the application is waiting on some resource (e.g. memory or disk). Early work by Ge et al. [4] explores the opportunities to reduce energy consumption by running memory-bound applications either at a fixed frequency for the entire run or at hand-tuned dynamic frequency settings for different functions in an application.

Freeh et al. [5] collect an application profile then use that profile to manually divide a program into phases based on a memory pressure metric, operations per cache miss. Each of the identified phases is then run on several of the available frequencies to determine the most energy efficient frequency for that phase. Green Queue automates the process of identifying possible application phases, determining the optimal frequency for those targets using power models rather than trial and error, then instrumenting the application binary to automatically shift the frequency as the application runs.

A number of projects use power/energy models based on hardware counter data [6], [7], [8] to determine optimal frequency settings for an application. Green Queue relies on application characteristics and power models based on those characteristics to make frequency scaling decisions. Often these approaches are time interval-based rather than program structure-based like Green Queue. Time interval-based approaches take observations about the application from previous intervals to estimate the time/power requirements and workload of upcoming intervals. These estimations are mostly based on hardware counters aggregated in the previous intervals: cache accesses counters [9], MIPS (Millions of Instructions per Second) [10] and CPU stall cycles [11]. Time interval-based approaches can run into suboptimal behavior when pre-defined time-intervals do not happen to line up with changes in application behavior. Green Queue's application phase analysis and identification avoids this by utilizing the application's structure to construct phase boundaries.

B. Inter-node DVFS

Inter-node DVFS schemes attempt to identify MPI inter-node load imbalance or the time spent blocked in MPI routines and use that information to lower the clock frequency of the hardware running the slacking or blocking MPI ranks. Freeh et al. [12] present a runtime system called Jitter which influences the clock frequencies of the CPUs running iterative codes using observations about the behavior of previous iterations within a run to predict the likely behavior of upcoming iterations. Their scheme is introduced to the application by inserting a special MPI call at the top of the main loop in the application, then intercepting that call in a Profiling MPI (PMPI) layer. Rountree et al. take an approach in Adagio [13] which makes runtime clock frequency selection decisions at many of the "natural" MPI call entry points within the application, while attempting to reduce energy and minimize runtime delay. Adagio meets these goals, achieving up to 20% energy reductions in certain MPI applications while maintaining minimal slowdowns. Adagio also makes several other contributions to the state of the art, most notably it demonstrates that regions of the application can and should be split across multiple clock frequencies where none of the available frequencies closely match the ideal frequency.

These schemes demonstrate significant progress toward minimizing the energy required to run poorly balanced MPI applications, though to our knowledge they have never been shown to work at scale. Our current approach is simpler than more refined schemes like Jitter and Adagio but it is demonstrated at scale, which allows it to serve as a proof of concept for using DVFS on parallel scientific applications running on *large* number of CPUs in order to reduce their energy impact. Future work includes incorporating many of the novel concepts introduced by other works in order to refine our own efforts to exploit MPI imbalances to reap energy savings.

III. Green Queue Framework

We start this section by describing a typical workflow for an application within Green Queue, for which a high level illustration is provided in Figure 1. Users submit jobs to the queue to be scheduled on the batch nodes. In order for an application to be subjected to application-aware DVFS, the application first needs to be instrumented for data movement and MPI communication trace collection. Once the trace is collected, it is analyzed to determine which of the two approaches—inter-node (detailed in Section V-B) and intra-node (detailed in Section IV and V)—is beneficial for the application. Note that this process needs to be done only once per application provided that the size and nature of the dataset remains similar to the original[1]. The result of the analysis is recorded in an application trace database. Any subsequent runs of the application will result in those customized DVFS settings being retrieved from the trace database, which Green Queue will use to instrument and run the application with those settings.

A. Application Characterization

Green Queue relies on basic block analysis and cache simulation tools developed on top of the PEBIL [14], an open source binary instrumentation toolkit for x86/Linux. A basic

[1]Often applications work on different datasets but the size of dataset remains relatively constant. When different datasets are used for the same application binary, these can be delineated by setting an environment variable to identify the name of the dataset.

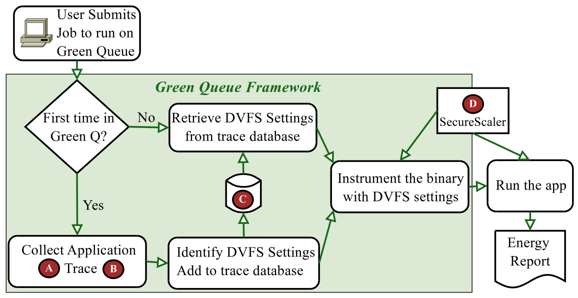
User Submits Job to run on Green Queue
Green Queue Framework
First time in Green Q?
No
Retrieve DVFS Settings from trace database
D
SecureScaler
Yes
C
Instrument the binary with DVFS settings
Run the app
Collect Application A Trace B
Identify DVFS Settings Add to trace database
Energy Report

Fig. 1. Green Queue Workflow (Letters in red circles correspond to the tools described in respective subsections)

block analysis tool provides static and dynamic information about each basic block in an application including instruction types and their respective counts, information on loop and function memberships, relationships to other basic blocks (control flow edges), memory access sizes, def-use distances, and basic block visit counts. Based on the basic block visit counts, the set of the most dominant basic blocks are selected as candidates for cache simulation. A cache simulation tool utilizes this list of candidates to produce cache hit rates per cache level for each candidate block.

B. MPI Communications Behavior

Green Queue uses PSiNSTracer [15], an open source MPI tracing and profiling library, to identify load imbalances in MPI applications. For this work, we extended PSiNSTracer to allow MPI profiling layer functions to be introduced to the application via the `LD_PRELOAD` environment variable on Linux systems, so it therefore requires no modifications to any stage of the application build process to use it. The PSiNSTracer library times and counts each type of MPI call in the application, then writes a summary of that information to disk when the application is finalized.

C. Storing Application Trace Data

Our strategies for clock frequency selection rely fundamentally on the trace data that we collect using the basic block analysis, cache simulation and MPI tracing tools that we described above. To ensure easy and fast access to this data, we have developed a system for storing and interacting with application trace data. This system uses an underlying SQL relational database as well as a high level interface which allows arbitrary queries to be run on the available static and dynamic application data, including queries on high-level control flow units such as basic blocks, loops and functions.

D. Frequency Scaling

To enable frequency scaling, we utilize the `cpufreq-utils` [16] package available on many Linux distributions. Since DVFS is a privileged operation, we have developed a light-weight and secure software layer called SecureScaler. SecureScaler is a daemon that accepts frequency change requests from user-level applications via a Unix-domain socket and optionally acts on those requests. This extra layer allows additional security policies to be used when applications issue frequency change requests.

In the next section, we discuss Green Queue's application analysis and phase discovery methodologies.

IV. Identifying Application Phases

A program phase is a path through the program's control flow graph which exhibits roughly uniform runtime behavior while on that path. Many runtime approaches use simple time slices because application specific information is not available. For intra-node DVFS we leverage detailed static and runtime application analysis and trace data collected using tools developed on top of PEBIL (see Section III-A) to customize phase granularity of the application. The trace data is used to reconstruct an abstract representation of the program. This representation is used by the phase analysis module within Green Queue to create an approximate context-sensitive call graph for the application. The call graph includes function summaries at each node as well as loop summaries within each function summary.

In this work, we utilize knowledge of the program structure to more accurately locate phase boundaries. We assume that all loops at the inner-most level are a single homogeneous phase and phase transitions occur at loop boundaries. The primary obstacle with this approach to phase identification is overcoming function calls. Phases are dynamic; they describe the execution of a program over time, so they easily break across function boundaries. If function calls are not taken into account, phases may be characterized inaccurately or missed entirely. For example, in a set of experiments done without accounting for functions no substantial phases were identified in two applications: MILC [17] and SWEEP3D [18]. Real applications are written to be modular so it is imperative that our analysis crosses function boundaries.

We use function inlining when analyzing trace data to create inter-procedural loop hierarchies. Inlining all the functions in a program is potentially a very expensive operation. The memory footprint required per function is multiplied by the number of callpoints it has throughout the application. Function inlining is further complicated by direct or indirect recursion. Attempting to inline functions that are part of a cycle in the call graph will result in an infinite loop of inlining. Our algorithm is designed to avoid large memory requirements by skipping functions that have a small number of dynamic instructions. These functions would have a very small impact on the result of our analysis, so eliminating them is acceptable. We choose not to attempt inlining on functions involved in a call-graph cycle. We avoid them using a worklist algorithm to only inline leaf functions, functions that make no function calls. The worklist initially contains all the functions in the application. As functions are taken off the worklist, they are considered for inlining. If they are either inlined or removed, every function that calls them is marked as changed and added to the worklist. When the algorithm completes, no leaf functions will remain in the call graph unless they were also root functions (i.e. main).

When a function is inlined, its dynamic statistics are aggregated into its new parent loop and function in proportion to the number of times it was called from that location out of the total number of calls. When inlining is complete, we are left with only root functions. Ideally there is only a single root function, the main function. Other root functions are possible if there are cycles in the call graph, if the calls escape our analysis (e.g. called by a function pointer), or if the functions are never called. We expect the first two cases to be rare; we have not yet witnessed their occurrence in any applications we have analyzed so far. The third case means that the function is of no further interest because the function is dead.

Each root function contains summaries to form an inter-procedural loop hierarchy. Initially, phase entry points are placed at the entry to the most deeply nested loops. Given our assumption that inner-most loops are homogeneous, this ensures that an actual phase follows each phase entry point and gives us a starting point for identifying larger phases. These phases may be too short to show energy gains with DVFS. The overhead of switching frequency has the potential of eliminating the energy savings of running at the lower clock frequency. To more suitably organize phases and the frequency scaling activity at their borders, we introduce a series of optimizations on top of the basic phase recognition scheme just described. These optimizations include eliminating small/noisy loops, merging neighboring phases that are of similar behavior, and eliminating frequency shifts at phase exit points and are detailed below.

Noisy Loop Elimination: To eliminate noisy loops, we make a pre-order traversal of the loop hierarchy. At each loop node, if the number of dynamic instructions in the loop is below some threshold, the loop and all of its children are eliminated from the analysis. If all children of some parent loop are eliminated, then the parent loop becomes an inner loop. Lower thresholds result in a larger number of phases and therefore more finely tuned frequency switching, though in turn this potentially results in higher overheads due to the cost of performing more frequency scaling operations. In Section VI we perform a series of experiments to evaluate different choices for this threshold.

Merging Neighboring Loops: Two phases are neighboring if they occur sequentially at runtime. We approximate this by considering all loops contained in the same parent loop to be neighbors. We use this approximation because application trace information containing dynamic inner loop ordering is impractical to gather. A depth first search algorithm is used on the loop hierarchy to combine phases. We traverse the loop hierarchy to the inner most loops, where if we find that all neighboring inner loops share the same optimal frequency their phases are eliminated and a new phase entry point is created at their parent loop's entry. This process is repeated until neighboring loops have different optimal frequencies or until an outermost loop is reached. Phase merging is also done at runtime; a call issued by the current phase to change to the frequency is only honored if the target frequency is different than the current frequency.

Phase Exit Optimization: The final optimization seeks to eliminate the overhead of restoring the original clock frequency upon exiting a phase by utilizing the observation that our phase-based analysis is generally very *complete*. That is, it covers the vast majority of the run of the application. Therefore, rather than resetting the frequency to its original value we eliminate the call to the frequency scaling library in order to leave the next phase responsible for setting its own optimal frequency. The risk in this optimization is that the analysis could be incomplete and result in executing long sections of the application (sections outside of phases) at suboptimal frequencies. However, if our analysis is complete, we can reduce the number of frequency changes by half or more. We evaluate the tradeoff involved in employing this optimization in Section VI.

V. Clock Frequency Selection

In this section, we describe Green Queue's method for the selection of a clock frequency strategy for the application, either by selecting the frequency for an application phase or by selecting a scaling strategy based on the detected MPI imbalance properties of the application. The decision about what clock frequency configuration is optimal is made based entirely on the amount of energy we can save with possible frequency configurations.

In order to select a clock frequency scaling strategy for a load-balanced application, Green Queue takes a phase profile, generated from the trace data as input, queries the power model for total system power draw estimation, and utilizes phase timing data to output the optimal frequency for the phases. The phase configuration is output to a file to be loaded at runtime by a DVFS instrumented application. For a load-imbalanced application, Green Queue bases the clock frequency selection on the results of an imbalance detection algorithm that uses the MPI profile collected for the application.

A. Intra-Node Frequency Scaling via Power Models

Direct measurement of power draw for different application phases can be inaccurate and sometimes impossible because power usually can only be measured at large granularity and at some expense/effort. The power draw measurement devices that we currently use only yield roughly one reading per second. Many application phases are too short to get useful power measurements given such a coarse power measurement granularity. A more practical approach is to relate important properties observed about an application with the total system power draw, then use that relationship to estimate power for an application phase with a given set of properties.

Earlier work [19] took this approach and introduced a loop generation framework called `pcubed`. The framework allows for the creation of a population of loops within a space of important application characteristics such as number of memory operations, floating point operations, working set size and various definition-use distances. Once this space is defined, `pcubed` generates the benchmark loops in order to populate it. Basic block analysis and cache simulation tools described in Section III-A are then used to derive an eightdimensional vector (see Equation 1) of observable properties for each of the generated benchmark cases.

Once traces are collected for the benchmarks, the benchmarks are run once per target system to measure the average power draw. One of the disadvantages of this approach is that for systems with a large number of configurable frequency settings, the number of benchmarks that have to be run to populate the characterization space explodes. A set of loops from pcubed that adequately covers the interesting characterization space, consisting of 2320 benchmarks, each configured to run for 5 seconds at the highest frequency, would take at least 3 days to explore on a Sandybridge node with 15 different CPU frequencies. This problem is made worse by random power or performance fluctuations that can affect measurements and force multiple collections of each data point, further increasing data collection time. In order to reduce the number of points that need to be measured, a machine learning approach is used to create power models based on a small subset of the pcubed space based on the problem formulation in Equation (1).

$$\begin{aligned} P_{sys} = f(&freq, l1\_p\_ins, l2\_p\_ins, l3\_p\_ins, \\ &fprat, mops\_ins, fops\_ins, int\_dud, fp\_dud) \end{aligned} \quad (1)$$

In Equation (1), $[l1, l2, l3]\_p\_ins$ are cache levels 1, 2 and 3 misses per instruction. $fprat$ is the ratio of the number of floating point operations to the number of memory operations, $mops\_ins$ is the number of memory operations per instruction. $fops\_ins$ is the number of floating point operations per instruction. int_dud and fp_dud are integer and floating point definition-use distances respectively. The specific machine learning algorithm used for this process is the gradient boosting method (gbm) [20].

Power prediction models are trained on a small subset of pcubed data that is supplemented with a set of 31 benchmark kernels to prevent over-training. These kernels are very prevalent in HPC applications and have been extensively used to evaluate intra-node auto-tuning techniques [21], [22]. Here we have categorized them into four categories following the scheme used in [23]: 1. Linear algebra computation kernels, which do different operations on scalars, vectors and matrices; 2. Linear algebra solvers, which solve a system of linear equations; 3. Stencil kernels, which update array elements following some fixed access patterns; 4. Data mining kernels, which do statistical analysis on random variables.

Green Queue takes 1400 data points from the kernels, varying working set sizes and frequencies, and combines them with 1400 data points from pcubed selected randomly from the set of all pcubed test cases on all frequencies. A 10-fold cross validated model is constructed using 600 random samples[2] from the total training set of 2800 samples and is able to predict the power draw of the remaining 2200 samples with an absolute mean error percentage of 2.5%. Figure 2 shows the modeled versus measured values for total system power draw for all 2800 samples in the combined pcubed

[2]We experimented with various training sizes, and for space reasons, we chose to present the model that gave us the most promising prediction results.

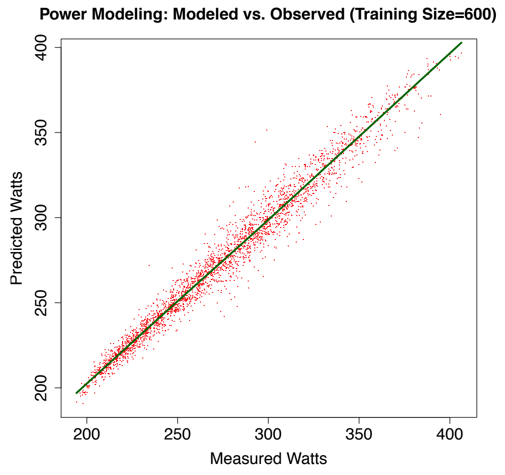

Fig. 2. Modeled vs. Observed Total System Power Draw

and HPC kernels set. The thick line in the graph is the trend line and for a well-behaved model, this line should be roughly a 45 degree line, which is the case in Figure 2.

When Green Queue needs to select a frequency for an application phase, the machine learning models are loaded and fed the phase profile. The models return predictions for the phase's power requirements at each available frequency. These predictions are combined with the phase timing data to estimate the optimal frequency.

B. Inter-Node Frequency Scaling on Load Imbalance

Load imbalance in MPI applications occurs when some MPI ranks are assigned more computational work than other ranks. Many MPI applications exhibit load imbalance issues due to the structure of the underlying scientific problem that is being solved or due to some artifact of the implementation of the solution. Remedies to the load balancing problem are well-developed in the literature and range from solutions which involve modification of the algorithm or implementation to doing dynamic load balancing. Despite these solutions load balancing remains a problem in HPC because application developers reasonably seek to avoid the complexity of introducing these solutions into their applications.

Green Queue absorbs imbalances by measuring them and introducing lower clock frequencies to the CPUs that are running MPI ranks which are running at less than capacity. The measurements of MPI behavior are collected using PSiNSTracer [15], an open source MPI tracing and profiling library.

We start by defining $CPUTime_i$ as the amount of time spent outside of MPI calls on rank i, then we define the excess computation of rank i for an MPI run on n ranks as follows.

$$excess_i = \frac{CPUTime_i}{MAX_{r=0}^{n}(CPUTime_r)} \quad (2)$$

$excess_i$ is therefore the ratio of the computation time of rank i to the most computationally intensive rank in the application. Note that Equation (2) is defined in such a way that the inequality $0 \leq excess_i \leq 1$ holds for all ranks. We then use the following formula to assign a clock frequency to some rank i, where p is a penalty factor that will be derived experimentally in Section VI-B.

$$Freq_i = (excess_i \times (1 + p)) \times Freq_{max} \tag{3}$$

Combining Equations (2) and (3) yields an equation directly relating the MPI profiling measurements to the clock frequency selections.

$$\frac{Freq_i}{Freq_{max}} = \frac{CPUTime_i \times (1 + p)}{MAX_{r=0}^{n}(CPUTime_r)} \tag{4}$$

That is, we select the clock frequency for a rank in such a way that the ratio of that clock frequency to the maximum frequency on the system is equal to the ratio of the $CPUTime$ of that rank to the maximum $CPUTime$ for all ranks in the run, subject to a penalty factor which can be used to tailor how aggressive the frequency selection is. $p = 0$ yields a clock frequency which equates these ratios, positive values of p yield higher clock frequencies, and negative values of p yield lower clock frequencies.

This scheme yields a clock frequency for each MPI rank, corresponding to a particular processing core. However, two factors prevent running these exact frequencies on each core. First, the set of clock frequencies available is generally discrete and limited to some small number of fixed values–15 frequency options in the case of the Sandybridge system we test in this paper. Because of this, we round the frequency produced by Equation 3 to the nearest available frequency. We also face the problem that clock frequency cannot be set for each core independently on a Sandybridge processor. Every core on a socket runs at the frequency of the maximum frequency that is set for any core attached to that socket.

We experimentally pursued several strategies with the goal of assigning ranks to cores in such a way that groups of similar frequency were assigned to the same socket. Such strategies have the effect of allowing the socket to achieve a lower overall frequency and result in lower power draws. However, rearranging ranks (e.g. remapping tasks) in this way risks destroying communication locality properties that are present within the application in addition to the risk of grouping resource intensive ranks together, pitting those ranks against one another for scarce processor resources. Empirically we found that the performance pitfalls of these strategies far outweighed the power draw improvements, so we omit the consideration of alternative mapping strategies in our results as we evaluate the effectiveness of the approach outlined in this section.

VI. Results

All experiments were conducted on Gordon, a supercomputer with 1024 compute nodes (16,384 total cores) recently deployed at the San Diego Supercomputer Center. Each compute node has 64GB of memory and consists of two sockets each running an 8-core Intel Sandybridge E5-2670 processor. There are 15 different clock frequency settings available in this processor model – 1.2GHz through 2.6GHz at 100MHz increments – and can be set independently for each socket. Gordon's compute nodes are configured as a 3D torus and are connected with a QDR Infiniband network. We obtain the rack-level AC power measurement directly from the APC Power Distribution Units (PDU) [24] that supply power to Gordon's compute racks. Each APC PDU supports remote power monitoring via an SNMP interface. This setup allows us to measure the power draw of any of the 16 1024-core racks of Gordon, though for the sake of consistency we use the same rack for all experiments in this work.

Green Queue was evaluated for 4 large-scale HPC applications and 3 benchmarks: MILC [17], SWEEP3D [18], CG [25], FT [25], MG [25], LAMMPS [26] and POP [27].

A. Intra-node Scaling

Recall from Section IV that our approach to intra-node DVFS involves identifying *phases* in the application. *Phases* are continuously executed sections in the application's control flow graph which have roughly uniform behavior. We start by exploring the effect of varying the number of dynamic instructions required to be executed in a phase in order to consider it valid. Phases below that threshold will be hidden from analysis since they have the potential to act as noise within a larger and more important phase. Lowering this threshold will identify more phases for DVFS at the margin, which in turn will usually incur higher overheads as the result of making a greater number of clock frequency switches. Raising this threshold will eliminate, as noise, the short phases at the margin. This will reduce the overheads resulting from performing clock frequency switches, but potentially misses valuable opportunities for DVFS.

In Figure 3 we evaluate this tradeoff empirically by running a set of application codes through Green Queue with threshold values ranging from 5×10^4 to 1.5×10^9. The results of lower thresholds are solid-colored bars of lighter color, while the results of higher thresholds are shown in darker solid colors. Examining the average energy savings for each threshold indicates that a threshold of 5×10^6 performed most suitably in reducing the energy required to run this set of application codes. However, note that there is a relatively wide range of threshold values that results in energy savings that are only very slightly worse than the energy saved by 5×10^6. We take this as evidence that the exact value of the threshold is relatively unimportant as long as the threshold is of approximately the right magnitude. We therefore proceed with the threshold 5×10^6 in the remainder of our experiments.

Having selected a minimum phase threshold, we now turn our attention to determining whether the optimization of skipping frequency scaling operations at the end of every phase is a useful optimization. This optimization reduces the overhead from performing frequency scaling operations. However, it also risks running the application in a sub-optimal frequency if the portion of the application following a phase is not covered by the analysis, is large, and runs optimally at a frequency that is different from the optimal frequency for the preceding phase. To evaluate the effects of this optimization we run a set of experiments (with a phase size threshold of 5×10^6) which disable this optimization, shown as hashed bars (labeled

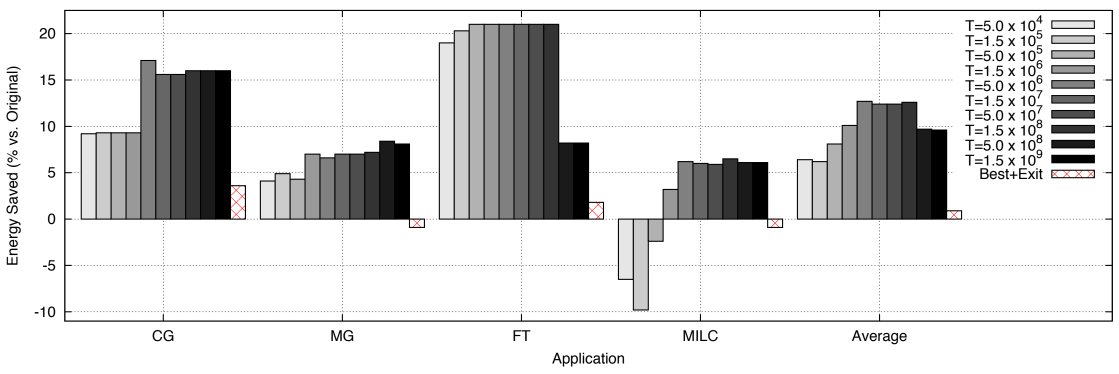

Fig. 3. Energy reductions of several application test codes using various phase length thresholds varying from 5×10^4 to 1.5×10^9 (solid bars ranging from light to dark respectively) and by disabling the phase exit point optimization (the hashed bar labelled Best+Exit).

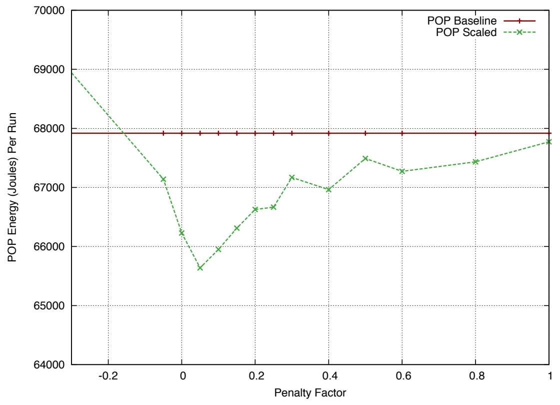

Fig. 4. POP inter-node clock frequency scaling based on varying levels of agression in the frequency selection strategy.

BestExit) in Figure 3. These results show that this optimization is overwhelmingly positive in the impact it has on energy savings, so our preferred configuration that we use for our final intra-node DVFS experiments is to use a threshold of 5×10^6 with the exit point optimization enabled. The results of running other applications through Green Queue with this configuration are shown in Table I.

B. Inter-node Scaling

In examining our inter-node frequency scaling scheme, we begin by presenting some empirical results relating to the selection of the value of the penalty factor p from Equation (4) in Section V-B. p is used within a strategy which attempts to exploit the imbalance of MPI applications to our advantage by noting that those ranks which are off the application's critical path can be run at lower clock rates, thereby lowering power draw while minimally impacting performance. Larger values for p result in more aggressive clock frequency strategies (that is, lower clock frequencies) and lower values result in more passive strategies (through higher clock frequencies). We evaluate a large range of values for p for POP, then use the results of this evaluation to select a single value for p that results in a well-performing tradeoff between the lower power draw and the potential loss in performance that can be the result of running at lower clock frequencies.

Figures 4 shows our evaluation for a variety of choices for p. The result is consistent with the conclusion that the choice of p as a small, positive value results in a nearly energy-optimal strategy[3]. In Section V-B, we use $p = 0.05$ to further validate the quality of this choice on the open source molecular dynamic application LAMMPS.

C. Discussion

Table I summarizes the intra-node and inter-node results for all the subject applications. Of all the applications that we considered for our intra-node application-aware DVFS, FT saves the most energy. Green Queue registers 6.2% energy saving for MILC, an application which uses a substantial number of dedicated allocation hours on many leadership class machines. It is also important to note the result for SWEEP3D, for which Green Queue directs us to always run at the highest available frequency, 2.6GHz. We validated this decision by running the phases detected for SWEEP3D at lower frequencies, and found that decreasing frequency quickly moved the energy usage upward, indicating that the decision made by Green Queue in that case is correct. Table I also shows the energy reductions realized when running our inter-node frequency scaling strategy with $p = 0.05$. The amount of energy that can be saved by selecting for frequencies based on those imbalances is based on the degree to which the code is imbalanced. Highly imbalanced codes (LAMMPS[4]) show more opportunity for energy improvement.

When we consider all the applications together, the average energy savings that we can achieve with Green Queue is 12.5%. This improvement in overall energy savings comes at the expense of average performance loss of 5.2%. The maximum energy savings that we achieve with the intra-node strategy is 21% for FT and this comes at the performance loss of 2.4%. The maximum energy savings from the inter-node strategy is 31.7% for LAMMPS, which comes with a performance penalty of 2.3%. When we consider that typical HPC system installations run well below full utilization, this makes a strong case for introducing marginal delays into application codes where such delays will show large reductions in the operating expenses of the system. Finally, the literature

[3]Similar experiments with other apps showed the same conclusion, though space limitations prevent us from presenting those evaluations.

[4]These imbalance properties are sensitive to the dataset used. That is, some datasets show severe imbalances while others do not.

TABLE I
SUMMARY OF OVERALL ENERGY SAVINGS WITH GREEN QUEUE

Application	FT	CG	MG	MILC	SWEEP3D	POP	LAMMPS
Technique	intra	intra	intra	intra	intra	inter	inter
Energy Savings	21.0%	17.1%	8.4%	6.5%	-	3.1%	31.7%

points to what is known as the *cascade effect* [28], which states that any energy reduction measured at the system level implies roughly similar amounts of energy saved throughout the center in the form of decreased cooling requirements and power transformer inefficiencies.

VII. CONCLUSION

In this work we examined the scalability of a set of techniques related to Dynamic Voltage-Frequency Scaling (DVFS) that we used on a modern supercomputer to reduce the energy footprint of running large-scale scientific applications and improve the reliability and lifespan of the hardware that runs them. We explored techniques which exploit opportunities to use frequency scaling in response to observations based on both intra-node and inter-node behavior.

We investigated representatives of these approaches on 1024 cores of Gordon, an Intel Sandybridge-based supercomputer at the San Diego Supercomputer Center, resulting in measured, full-system energy reductions that averaged 10.6% and 17.4% respectively for our intra-node and inter-node approaches respectively. These results show that DVFS is scalable and well-suited to reducing the energy impact of running large-scale scientific applications. This impact should not be understated; individual HPC center energy bills currently are in the millions of dollars per year, a level at which reductions of the of energy usage at the scale shown here would quickly realize significant and real resource savings.

ACKNOWLEDGMENT

The authors would like to extend sincere gratitude to Shawn Strande, Rick Wagner and Mahidhar Tatineni of the Gordon project at San Diego Supercomputer Center. This work was supported in part by the DOE Office of Science through the SciDAC award titled SUPER (Institute for Sustained Performance, Energy and Resilience) and in part by the National Science Foundation grant: OCI #0910847, Gordon: A Data Intensive Supercomputer.

REFERENCES

[1] J. Koomey, "Growth in data center electricity use 2005 to 2010," *Analytics Press*, August 2011.

[2] United States Environmental Protection Agency, "Report to congress on server and data center energy efficiency," August 2007.

[3] SDSC, "Gordon user guide: Technical summary," March 2011. [Online]. Available: http://www.sdsc.edu/us/resources/gordon/

[4] R. Ge, X. Feng, and K. Cameron, "Improvement of power-performance efficiency for high-end computing," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, april 2005.

[5] V. W. Freeh and D. K. Lowenthal, "Using multiple energy gears in mpi programs on a power-scalable cluster," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05, New York, NY, USA, 2005.

[6] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: a platform for os-level power management," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 289–302.

[7] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proceedings of the 2004 international symposium on Low power electronics and design*, ser. ISLPED '04. New York, NY, USA: ACM, 2004, pp. 174–179.

[8] C. Lively, X. Wu, V. Taylor, S. Moore, H.-C. Chang, C.-Y. Su, and K. Cameron, "Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems," *Computer Science - Research and Development*, pp. 1–9, 10.1007/s00450-011-0190-0.

[9] R. Ge, X. Feng, W. Feng, and K. Cameron, "Cpu miser: A performance-directed, run-time system for power-aware clusters," in *Parallel Processing, 2007. ICPP 2007. International Conference on.* IEEE, 2007.

[10] C.-h. Hsu and W.-c. Feng, "A power-aware run-time system for high-performance computing," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1–.

[11] S. Huang and W. Feng, "Energy-efficient cluster computing via accurate workload characterization," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009.

[12] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. K. Bletsch, "Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs," *J. Parallel Distrib. Comput.*, vol. 68, no. 9, pp. 1175–1185, Sep. 2008.

[13] 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*, ser. ICS '09. New York, NY, USA: ACM, 2009.

[14] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, march 2010, pp. 175 –183.

[15] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "Psins: An open source event tracer and execution simulator for mpi applications," *Euro-Par 2009 Parallel Processing*, pp. 135–148, 2009.

[16] "CPU Frequency Scaling," https://wiki.archlinux.org/index.php/Cpufrequtils.

[17] "MIMD Lattice Computation (MILC) Collaboration," http://www.physics.indiana.edu/~sg/milc.html.

[18] "ASCI Sweep3D Readme," http://wwwc3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html.

[19] M. A. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. M. Tikir, and S. Poole, "Reducing energy usage with memory and computation-aware dynamic frequency scaling," in *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 79–90.

[20] "Generalized Boosted Regression Models," http://cran.r-project.org/web/packages/gbm/.

[21] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, may 2009.

[22] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth, "A Scalable Auto-Tuning Framework for Compiler Optimization," in *IPDPS'09*, Rome, Italy, May 2009.

[23] "Polybench: Polyhedral Benchmark Suite," http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[24] "American Power Conversion Corp." www.apc.com.

[25] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks - summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165.

[26] "Large-scale Atomic/Molecular Massively Parallel Simulator," http://lammps.sandia.gov/.

[27] "Parallel Ocean Program," http://climate.lanl.gov/Models/POP/.

[28] "Energy Logic: Reducing Data Center Energy Consumption by Creating Savings that Cascade Across Systems," http://www.cisco.com/web/partners/downloads/765/other/Energy_Logic_Reducing_Data_Center_Energy_Consumption.pdf.