CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

PMaC's Green Queue: A Framework for Selecting Energy Optimal DVFS Configurations in Large Scale MPI Applications

Joshua Peraza*, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Allan Snavely

Performance Modeling and Characterization (PMaC) Laboratory
San Diego Supercomputer Center
University of California, San Diego
San Diego, California 92093–5004

SUMMARY

This article presents Green Queue, a production quality tracing and analysis framework for implementing application aware Dynamic Voltage-Frequency Scaling (DVFS) for MPI applications in high performance computing (HPC). Green Queue makes use of both intertask and intratask DVFS techniques. The intertask technique targets applications where the workload is imbalanced by reducing CPU clock frequency and therefore power draw for ranks with lighter workloads. The intratask technique targets balanced workloads where all tasks are synchronously running the same code. The strategy identifies program phases and selects the energy-optimal frequency for each by predicting power and measuring the performance responses of each phase to frequency changes. The success of these techniques is evaluated on 1024 cores on Gordon, a supercomputer at the San Diego Supercomputer Center built using Intel Xeon E5-2670 (Sandybridge) processors. Green Queue achieves up to 21% and 32% energy savings for the intratask and intertask DVFS strategies respectively. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Energy Efficiency; Dynamic Voltage-Frequency Scaling; High Performance Computing; Green Computing; Application Characterization; Workload Characterization; Performance Modeling; Power Modeling

1. INTRODUCTION

As the size of supercomputers increases, so too do their power requirements. With the current move towards exascale computing, the number of processor cores in a system is ever increasing, imposing additional power requirements on the system both for operational and cooling purposes. The cost of powering supercomputers has become substantial. The K computer, currently heading the Top500, consumes 12.7 MW of power [1]. Using a conservative estimate, $0.10/kWh, the cost of electricity could reach up to $11 million annually. Over the lifetime of certain supercomputing systems, the cost of energy can actually exceed the initial hardware cost of the system [2].

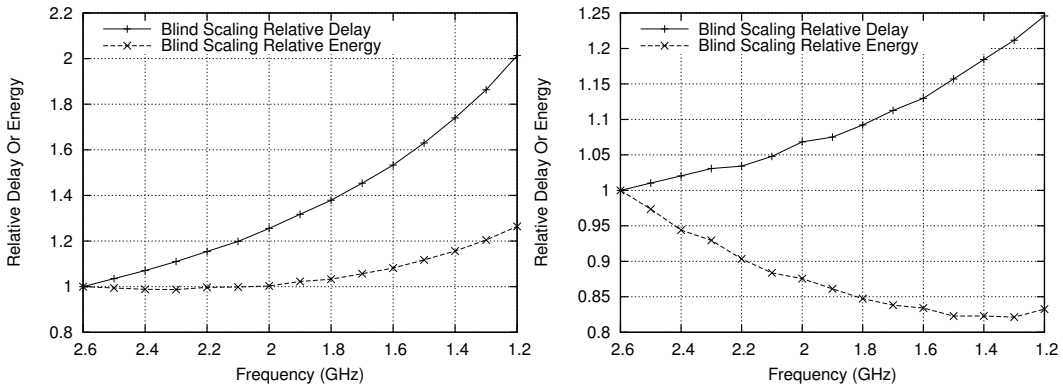
Energy efficient computing requires controlling how much energy is committed to a task and understanding what benefit can be expected in return for completing that task. In many cases this presents the opportunity to make informed tradeoffs between performance and energy consumption. In this article, Dynamic Voltage-Frequency Scaling (DVFS) is the mechanism used for controlling this trade-off. Many researchers have used DVFS to either reduce total energy costs or to cap dynamic power consumption. DVFS allows the speed of a CPU to be reduced in exchange for reduced power consumption. The key challenge to using DVFS effectively is to understand what

*Correspondence to: jperaza@cs.ucsd.edu

the effect on both power and performance will be for a change in CPU frequency. Without this understanding one could certainly reduce dynamic power consumption, but it may come at the cost of unacceptable performance loss and possibly even increased energy usage than if no DVFS were used at all [3].

1.1. Motivation

Validation of this possibility of running into suboptimal energy usage behavior, as well as motivation for the application aware DVFS techniques explored in this work, is shown in Figure 1. This figure shows the delay and energy measured for two application kernels, details of which are available in Section 4, when subjected to each available clock frequency on an Intel Xeon E5-2670 for the entire application run.



(a) Frequency scaling results for Sweep3D.

(b) Frequency scaling results for FT.

Figure 1. The energy usage and runtime delay for two test applications being run at a series of fixed clock frequencies.

In Figure 1a, Sweep3D [4] shows that reduced clock frequency does not always result in reduced energy usage. The CPU benefits from a cubic power reduction as clock frequency is reduced linearly, but the application suffers such a large performance penalty that the additional energy consumed by other components dwarfs the energy savings of the CPU. In Figure 1b, FT [5] demonstrates that, while it is possible to save energy via simple frequency scaling, the relationship of clock frequency to performance and power is not necessarily a simple one. FT achieves a minimum energy consumption with a fixed CPU clock frequency of 1.3 GHz, which is close to but above the minimum frequency available on the system. The goal of Green Queue is to use an application's specific characteristics identify the clock frequency which minimizes energy consumption for a particular section of code and then use this ability to customize a dynamic voltage-frequency scaling for the application.

The focus of this article is on exploring techniques for determining optimal frequency configurations for pure MPI applications. Optimality in this context refers to the set of CPU clock frequencies that minimizes total energy consumption for the hardware running an HPC application. However, other metrics derived from combinations of performance and power, such as energy-delay product, can be minimized similarly. Resulting from this exploration is Green Queue, a framework supporting the analysis and instrumentation of MPI programs for DVFS. Green Queue was evaluated using an array of full scale HPC applications and benchmarks running in various configurations on a single rack (1024 cores) of Gordon, the latest supercomputer to be installed at the San Diego Supercomputer Center. Green Queue demonstrates energy savings of up to 21% with intratask DVFS and up to 32% with intertask DVFS.

1.2. Contributions

The contributions of this work include:

DOI: 10.1002/cpe

Detecting and characterizing program phases using interprocedural analysis on application trace data. A structural phase detection algorithm, as opposed to a time slicing method, allows phase length and location to adapt to boundaries naturally present in programs, such as loop and function boundaries. Interprocedural analysis further improves upon this because these boundaries are not limited to the scope of single loops or functions.

Efficiently determining optimal frequency configurations over all phases in a program. Selecting an optimal configuration by direct measurement requires several runs for each phase to locate the optimal frequencies because the length of a phase is short compared to the time granularity needed to measure power. This work proposes a method of combining power modeling and performance measurements to predict energy consumption for frequency configurations using one application run per frequency setting.

A production-quality tracing and analysis framework. Green Queue provides a robust framework for generating and analyzing application trace data. The production, organization and processing of application trace data is automatic and largely opaque to the user, managed by a few simple command line tools. This allows for comprehensive energy management strategies to be built to leverage a wide array of detailed application statistics, and for those strategies to be easily utilized.

An evaluation of Green Queue on a full compute rack of Gordon. Gordon is the latest supercomputer at the San Diego Supercomputer Center built with Intel Xeon E5-2670 (Sandybridge) processors. With 1024 cores per rack, this work demonstrates that DVFS is a viable strategy for achieving significant energy savings at scale on modern hardware.

Experiments demonstrating the value of application aware DVFS over blind DVFS. This work demonstrates that application aware DVFS can significantly outperform techniques that do not respond to the behavior of the application. Green Queue shows increased energy savings coupled with a decreased performance loss compared to application unaware DVFS.

2. BACKGROUND AND RELATED WORK

DVFS has been explored by many researchers as a mechanism for reducing energy consumption in HPC workloads. Generally these DVFS strategies have fallen into two broad categories: intertask and intratask. Green Queue utilizes both intertask and intratask strategies for DVFS, and we discuss research related to these two strategies seperately.

2.1. Intertask DVFS

Intertask DVFS schemes attempt to identify MPI load imbalance or the time spent blocked in MPI routines and use that information to selectively lower the clock frequency of the hardware running the slacking or blocking MPI ranks, since by definition some other rank is the bottleneck of application progress. Freeh et al. [6] 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 particular 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 [7] 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 is

DOI: 10.1002/cpe

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 for Green Queue 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.

2.2. *Intratask DVFS*

Intratask DVFS attempts to reduce energy consumption by varying the CPU frequency within a single task as it moves between program phases. If a region of code is heavily utilizing the memory subsystem, the CPU may be spending much of its time stalled while waiting for memory requests to complete. When the memory subsystem becomes the bottleneck, scaling down the CPU frequency can result in sub-linear performance loss, allowing for a net energy savings.

For much of the past decade of DVFS research, predicting performance loss under DVFS and comparing it with total system power draw has not been necessary to save energy. Because CPU power has historically dominated total system power, a decrease in frequency resulted in a net energy savings, despite linear performance degradation. Consequently, many strategies have been CPU centric; they focus on estimating memory boundedness and then scaling the CPU aggressively to some acceptable performance loss. Choi et al. use an analytical model of off-chip vs. on-chip workloads using hardware counters to estimate cycles per instruction [8]. A similar approach has been taken by Dhiman et al. [9], Isci et al. [10] and Wu et al. [11].

There are two high level questions any strategy must answer to perform intratask DVFS: when should the frequency be adjusted and to what frequency? The research discussed so far had similar solutions to the second question; they use analytical models for the balance of on-chip vs off-chip workloads then the frequency is scaled to match some target performance loss.

Strategies for solving the first question are somewhat more variable. If we define a phase as a window in the execution of a program where certain relevant characteristics are homogeneous, this question becomes one of identifying phases in the application. In the case of DVFS, the homogeneity of the characteristics during a particular phase implies that there is a single optimal clock frequency for that phase. The brevity of many particular phases and sheer number of phases in real programs precludes determining the optimal frequency for all phases experimentally.

Simple strategies for finding phases are reactive. They record hardware counters on time sliced intervals and re-evaluate the optimal frequency at each based on the assumption that phases exhibit temporal locality [8][9]. These strategies are well-behaved when phases are longer running than several time slices. They make the assumption that the properties observed in the previous time slice are a good predictor for the properties that will occur in the next. However, if phases are short lived relative to the length of the time slice then the frequency selected for each time slice may be suboptimal for a substantial fraction of the phase.

Time slicing strategies may also be predictive. Isci et al. use a phase history table to predict what phase will occur in the next time slice [10]. In this strategy, phases are discretized by equating a phase to a particular range of values of the frequency determining metric, memory operations per micro-op. Each range corresponds to one frequency setting. A small number of frequency settings and coarse grained time slicing, therefore, allow a small enough table to efficiently predict phase patterns.

Time slicing has the advantage of being amenable to dynamic phase detection, but phases may be more precisely detected with more information about the program being executed. Wu et al. achieve this using dynamic compilation to detect phases at the function or loop granularity [11]. The intuition behind this strategy is that programs are structured and phases will tend to be related to program constructs. In their dynamic compiler framework, Wu et al. instrument the program to measure memory operations per micro-op at loops and functions and again use an analytical model to select frequency according to some acceptable performance loss.

The aforementioned strategies are CPU centric. They use DVFS to reduce CPU dynamic power and ignore total system power. This is acceptable when one assumes that CPU power dominates total system power draw. Linear reductions in frequency and voltage allow cubic dynamic power savings. Given this assumption, any reduction in frequency can be expected to lead to a reduction in energy,

DOI: 10.1002/cpe

even for entirely CPU bound applications. The primary concern has been estimating performance loss at each frequency and then scaling down as aggressively as possible within the bounds of some parametrized acceptable performance loss.

The assumption that CPU power dominates total system power is becoming increasingly dubious in modern computing systems. As CPU voltage and feature size shrink, so too does the range of dynamic power and opportunity for power savings via DVFS. Leakage and power from other components are taking up a larger portion of total system power. Now, the energy cost of a performance loss due to static power is enough to overtake some dynamic power savings [3][12]. One can no longer simply reduce CPU frequency to save energy because a linear performance loss times the static power for the whole system dwarfs the energy saved by the CPU alone. Consequently, saving energy in modern and future systems requires techniques that can accurately assess the total system power and performance characteristics of an application to make an optimal power-performance tradeoff.

Freeh et al. use a strategy that accounts for total system power by directly measuring the energy usage for different frequency configurations [13]. If there are l phases and f available frequencies, then an exhaustive search requires f runs per phase for a total of $f * l$ runs. This is because the length of an application phase is often too short for the power to be measured in isolation; power measurements are affected by the behavior of neighboring phases. The energy-optimal frequency for a phase is selected by altering the frequency of that single phase for multiple runs and attributing the difference in energy consumption between runs to that phase. Freeh et al. use a heuristic to reduce the number of runs to $O(l)$. First, a program is divided into phases using traces which report operations per cache miss for blocks of code. As with previous strategies, the model assumes that operations per cache miss correlates strongly with optimal frequency. Frequency selection is performed for each phase in order of increasing operations per cache miss. For the first phase, energy consumption is first measured at the highest frequency and then at each lower frequency until the optimal frequency for that phase is found. All subsequent phases are assumed to have an optimal frequency equal to or lower than this frequency, reducing the size of the search space.

There are two primary weaknesses with this strategy that Green Queue addresses. First, operations per miss is not always a good predictor for optimal frequency. Optimal frequency depends on an applications performance response to frequency which has been found difficult to predict generally with any simple metric. This can result in missed energy savings in phases that were assumed unsuitable for DVFS or that were scaled lower than their actual energy-optimal frequency. Second, this method still requires $O(l)$ application runs to determine a frequency configuration. Large applications may have many phases requiring a significant energy investment. Green Queue's method requires $O(f)$ application runs taking advantage of the fact that the number of frequency settings is often much smaller than the number of phases in real programs.

Snowdon et al. use hardware counters to model both system power consumption and performance response to frequency scaling [14][15]. In the model, a training set of hand picked applications are used to correlate hardware counter measurements with power and performance responses to frequency scaling. The counters with the highest correlations to power and performance measurements are then used at runtime to estimate both power and performance response to potential frequency changes.

This model, however, is not portable. Different CPU models provide different counters which will require different models. Furthermore, micro-architectural advancements and the move to multi-core complicate modeling. There has been some success modeling power on more recent multi-core processors [16], but performance modeling, and therefore energy modeling, may be more challenging.

One approach to creating portable models is to create the model in software. Instead of using hardware counters, Laurenzano et al. use application traces to derive loop signatures [17]. These signatures are compared against signatures collected from a training set which covers a space of power- and performance-relevant metrics. Because the model is in software, it is not restricted by the availability of hardware counters. Since the domain of possibly relevant software properties is boundless, this allows modeling behaviors that are difficult or impossible to capture with hardware

DOI: 10.1002/cpe

counters alone. While an artificial benchmark suite also provides control over the training space, it can be prone to systematic errors. The space covered by the benchmark can be decisively explored, but the model loses variability in properties that aren't specifically accounted for by the benchmark.

3. GREEN QUEUE

The Green Queue project is a production-quality framework of tracing, database, and analysis tools to facilitate running user applications at reduced energy costs using a variety of techniques. Figure 2 gives an overview of the Green Queue framework.

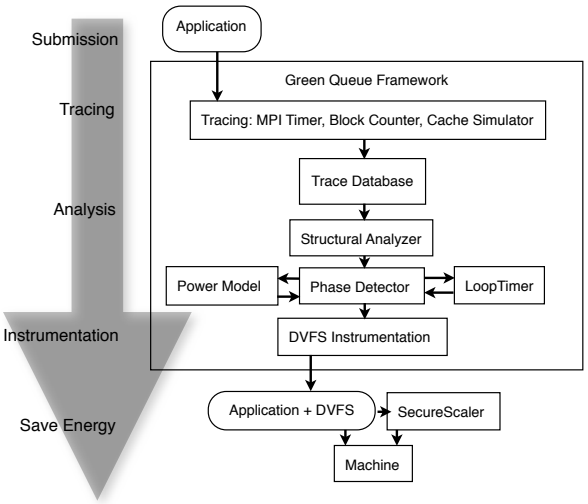


Figure 2. The Green Queue Framework

The first time an application is submitted to Green Queue, it is examined by several analysis tools to learn about both its static properties and runtime behavior. Runtime behavior is examined by gathering traces while the submitted application is executed. The next time the application is submitted to the Green Queue, the analysis from its previous submission is retrieved from a database. That analysis is used to instrument the application with DVFS controls, which is then run using those controls to reduce the application's energy footprint.

3.1. Characterizing Applications

The first step in an application's interaction with Green Queue is to characterize the application. At this stage, the application is analyzed to find its static properties then is run using binary instrumentation and MPI profiling tools to measure its runtime behavior.

3.1.1. MPI Load Imbalance — MPI load imbalance is measured using PSiNSTracer [18], a tool that provides wrappers for instrumenting MPI routines. These instrumented routines are introduced to the application using the `LD_PRELOAD` mechanism, which has the effect of timing and counting the calls to all MPI routines during an application run. The results of these timings can be used to determine the amount of communication and CPU time as well as detect imbalances of CPU time between MPI ranks. When imbalances beyond a certain threshold are detected, the application is identified as a candidate for intertask DVFS.

3.1.2. Static Analysis — The binary instrumentation framework PEBIL [19] is used in its capacity as a static analysis tool to produce information about the structure of the program as well as what operations occur within those structures. This results in a detailed picture of the boundaries of

DOI: 10.1002/cpe

various control flow structures such as functions and loops as well as the type and size of important classes of operations that reside in those structures, such as floating point or memory operations. PEBIL also records the average size of memory operands in each block and measures the number of instructions between register or memory definitions and their usage. Blocks are associated with the loops and functions that contain them. The tool also records, when possible, the targets of function calls for each block.

3.1.3. Runtime Execution/Memory Tracing — Green Queue uses PEBIL in two seperate instrumented application runs to collect two types of traces for an application: execution counts and memory tracing. In the first run, Green Queue collects counts for all of the basic blocks and loops in an application, showing its hot spots and how the various control structures in the program are exercised. Green Queue then makes a second application run during which the hot spots are instrumented to simulate the application's memory address streams interacting with the memory subsystem of the machine. It is important to note that simulation of this kind provides far more detail about the memory behavior than can be obtained with hardware counters. In particular, simulation allows Green Queue to attach memory behavior statistics to the control structures (functions, loops, basic blocks and instructions) in the program.

3.2. Application Trace Analysis

After trace data is collected, Green Queue uses this data to construct a representation of the program. Each phase in the program is detected and analyzed to determine an optimal frequency configuration.

3.2.1. Trace Database — The application trace database is at the center of Green Queue's application analysis capabilities. This database provides a clean interface to the trace data gathered for an application. On the backend, this trace data is stored in an SQL database to allow for arbitrary and powerful querying of that data. A java class frontend allows for the re-use of many common high level queries across the several analysis modules in Green Queue and other related projects. The trace database also supports querying and aggregating data by function, loop, or basic block and selection of data particular to specific MPI ranks.

3.2.2. Interprocedural Loop Analysis — An interprocedural analysis is critical to the phase detection accuracy of Green Queue for real applications. In order to make DVFS decisions about loops, the loops must have an accurate characterization profile. If function calls that are made from within loops are not counted toward those loops, the resulting characterization can be innacurate. Consider the loop in Figure 3a.

```
for ( i = 0; i < N; i++ ) {
  a[i] += b[i]
  f()
}
```

(a) Demonstration of how loop characterization can be inaccurate without interprocedural analysis

```
for ( i = 0; i < N; i++ ) {
  f()
}
```

(b) Demonstration of how phases can go undetected without interprocedural analysis

If the function call in this loop were not considered, this loop may be characterized as a very memory intensive loop and could erroneously result in a low clock frequency being used to run the loop. The actual behavior of this loop, however, depends on what the function `f` does. If `f` works the CPU more heavily than memory and is non-trivial, then it is likely that the loop would be best run at a higher frequency.

A second motivation for performing interprocedural analysis is the detection of phases in modular programs. To ensure that DVFS is used effectively, only phases which contain more than a certain number of dynamic instructions are considered. Now consider the loop in Figure 3b.

DOI: 10.1002/cpe

In this case a simple intraprocedural analysis would find an empty loop unsuitable for DVFS. However, if `f` has a large number of instructions or if the loop has a large number of iterations, a more sophisticated view of this loop may show that it forms a substantial phase. Green Queue uses a structural analyzer to support interprocedural analysis and to provide a mechanism for high level navigation of the application traces. The structural analyzer creates loop and function summaries and approximates a context-sensitive call graph, allowing analysis of inter-procedural loop hierarchies.

Function inlining is used to perform the interprocedural analysis. However indiscriminate function inlining can cause an explosion in memory usage by the structural analyzer because each time a function is inlined, the data structures accounting for its trace data are duplicated at each point where it is inlined. For many small, insignificant functions, inlining does not improve the accuracy of phase characterization because the number of dynamic instructions that they contribute is too small to significantly affect the code that calls them. To mitigate this effect the inlining algorithm prunes functions whose dynamic instruction count is below a certain threshold.

A second concern when performing inlining is handling cycles in the call graph resulting from recursion. To simplify the analysis, functions that are members of cycles are never inlined. Green Queue targets HPC applications where it is expected that significant recursion will be rare, and in practice this simplification has not been a problem thus far.

Algorithm 1 Function Inlining

```
Add all functions to the worklist
while worklist is not empty do
    Remove a function f from the worklist
    if f is not a leaf function then
        continue
    end if
    if number of dynamic instructions in f is below a threshold then
        Remove f from call-graph
        Remove all calls to f
    else
        Inline f at all callers
    end if
    Add all callers of f to the worklist
end while
```

The algorithm used to perform function inlining within Green Queue's structural analyzer is a worklist algorithm, shown in Algorithm 1. Initially, all functions are added to the worklist. At each iteration, a function is removed from the worklist. If the function contains any function calls, it is not a leaf function and is not ready for inlining. It is removed from the worklist and no inlining is done. Otherwise, if the function is smaller than a threshold, it is pruned. It is removed from the worklist and all references made to it by other functions are removed. The effect is the same as if the function were inlined, but no additional resources are used. If the function is above the threshold, then it is considered significant. It is inlined at all functions that call it. When a function is pruned or inlined, all functions that contain references to the function are added to the worklist.

When a function is inlined, all of its dynamic statistics are aggregated into the appropriate level of the loop hierarchy in each function that called it. One challenge in this process is that the summary for the function being inlined contains context insensitive dynamic statistics; it contains aggregated information for *all* call stacks leading to this function. If the function's statistics were aggregated entirely into the caller function, the caller function's statistics would become distorted. In the absence of sufficient information in the trace data to guarantee accurate attribution of this information to the correct caller, the assumption is made that the portion of instructions attributable to any caller of a function is proportional the number of times the call originated from that call site. This information is readily available from the execution trace data discussed in Section 3.1.3 and is a useful approximation that greatly reduces the time and space complexity of the required trace data.

When the worklist is empty, most non-cyclical functions will have been inlined. The function will not be inlined if it participates in a cycle, if it is not called by any other function, or if the analysis could not locate where it was called. If a function is never called, it is either insignificant

DOI: 10.1002/cpe

because it does not effect the analysis or it is a root function such as `main`. For simplicity, the analyis framework does not support detecting calls via function pointers. These types of calls will go uninlined in the analysis but they are expected to be rare in HPC applications and have so far not been found to interfere significantly with the analyses.

3.2.3. Phase Detection — In prior work, phases have often been identified by sudden changes in certain application characteristics. The approach offered here differs markedly in that no assumption is made about which properties are used to determine phase boundaries. The definition of a phase in Green Queue is a contiguous execution of code where the energy optimal frequency is approximately homogeneous. The characterization of code may change, but if optimal frequency remains the same, no phase change has been made. Consequent to this approach, frequency selection occurs midway through Green Queue's phase detection process.

To bootstrap the phase detection algorithm, the assumption is made that each inner-most loop is homogeneous and is therefore a phase unto itself. If a loop has no nested sub-loops the whole loop is marked as belonging to a single phase. Initially, the phase entry point is the head of the loop but may move when phases are merged.

There is an overhead to switching frequencies and if a phase is too short, the cost of switching to an energy-improving frequency may surpass the benefit of running at that frequency. To avoid this, a pass over the loop hierarchy is made which eliminates small loops as noise. The analysis makes a pre-order traversal over the loop hierarchy and at each loop examines the number of dynamic instructions contained within it compared to the loop entry count gathered during the application's execution trace. If the loop has too few dynamic instructions per entry, the loop and all of its children are removed from the analysis. This may allow phase entry points to be moved to outer loops when all of their children are eliminated. Additionally, if a loop has no siblings (its parent loop has no other sub-loops) then the phase entry point is moved to the parent loop. This allows the call to the frequency scaling library to be placed higher in the loop hierarchy and therefore introduces less overhead. Once phase entry points have been moved to sufficiently sized loops, frequency selection analysis is performed in order to mark each phase with an optimal frequency. More on this process is described in section 3.5.2.

After frequency selection is complete, a second pass over the loop hierarchy is made to merge neighboring phases with the same frequency. This is done via a depth first traversal of the loop hierarchy. When the traversal reaches the most deeply nested loop, its optimal frequency is compared to the frequency selected for each of its siblings. If all sibling loops share the same frequency, they are merged and the phase entry point is moved to their parent loop. This process continues up the loop hierarchy until two or more sibling loops have differing optimal frequencies, indicating a phase transition.

Since the trace data does not contain information about the order in which loops are executed, there may still remain unmerged neighboring phases with the same frequency. This is addressed with dynamic phase merging, discussed further in section 3.3.1.

3.3. Deploying Applications with DVFS

The result of the phase detection and frequency selection algorithms is a frequency configuration file for the application which describes the boundaries and optimal clock frequencies for the application's phases. Given this frequency configuration, more infrastructure is needed to employ DVFS on an application on a production system. This section describes the mechanisms put in place to accomplish these frequency scaling operations.

3.3.1. Instrumenting with a Customized DVFS Scheme — An instrumentation tool written with PEBIL is used to instrument an application for DVFS. At instrumentation time, calls to a frequency scaling library are programmed into the application's phase entry points. At the start of the application run, this library loads the frequency configuration file, which contains a mapping of phases to frequencies. Deferring the handling of frequency settings until runtime in this fashion allows frequency settings to be modified without re-instrumenting the application. The frequency

DOI: 10.1002/cpe

scaling library can track statistics about the performance of the DVFS configuration such as the number of frequency switches and can perform certain optimizations that were not possible offline. By tracking the current frequency on any given CPU, Green Queue is able to skip the frequency scaling operation when the current frequency matches the target frequency. This has the effect of dynamically merging neighboring homogenous phases that were too complex for our offline phase merging algorithm to find, reducing the overhead introduced by the DVFS scheme and thereby increasing its effectiveness.

3.3.2. Introducing DVFS Securely — Green Queue uses a mediator between the application and the operating system to handle frequency scaling requests. This mediator, SecureScaler, is a daemon which accepts requests on Unix domain sockets and can be used to implement security policies with regard to how frequency scaling requests are honored, such as allowing only particular users or groups to modify clock frequency or to honor requests only on a particular subset of the system. This makes Green Queue far easier to integrate into a supercomputer's software and middleware.

3.4. Characterization of Training Benchmarks

For our intratask frequency selection method, described in Section 3.5.2, models are used to make predictions about power or performance of application phases at different frequency settings. These models are based on characterizations and measurements collected from sets of training benchmarks. A characterization profile for one of these benchmarks currently includes cache misses per instruction for each level of cache, ratios of floating point and memory operations to each other and to the total number of instructions, and average value def-use distances for integer and floating point operations. The time and power response to different frequency settings is directly measured during each of the training benchmark runs.

3.4.1. pcubed — The first training set used is `pcubed` [17]. `pcubed` is a framework for automated, systematic generation of micro-benchmarks across a parameter space. The parameters to `pcubed` allow generation of benchmarks that cover the characterization space and have a configurable runtime length. `pcubed` therefore allows a controlled exploration of the characterization space. However, while `pcubed` can generate benchmarks with specified property values, unspecified characteristics such as memory access patterns and dependencies lack variation and are thus overtrained. Models that use `pcubed` alone can accurately predict the entire parameter space of `pcubed` benchmarks with a relatively small number of data points while still mispredicting for real applications.

3.4.2. Other Micro-Benchmark Kernels — To supplement `pcubed` a suite of 31 HPC kernels are added to the training set. These kernels add variety and complexity to the training set to help prevent models from becoming overtrained to the particular chacacteristics explored explicitly by `pcubed`. These kernels come from a variety of different domains, are very prevalent in HPC applications, and have been extensively used in the past to investigate and evaluate intranode auto-tuning techniques [20][21][22]. Here we have categorized them into four categories following the scheme used in [21] and [23] – (1) Linear algebra computation kernels, which do different operations on scalars, vectors and matrices such as matrix-matrix or matrix-vector multiplication; (2) Linear algebra solvers, which solve a system of linear equations; (3) Stencil kernels, which update array elements following some fixed access pattern such as the jacobi relaxation method; and (4) Data mining kernels, which do simple statistical analysis on random variables.

3.5. Frequency Selection within Green Queue

In this section, we describe Green Queue's method for the selection of a clock frequency strategy for the application, either by way of 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 based entirely on choosing the

DOI: 10.1002/cpe

frequency which we predict can save the maximum amount of energy out of all possible frequency configurations.

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 (refer to Section 3.1.1) collected for the application. In order to select a clock frequency scaling strategy for a load-balanced application, Green Queue takes a phase profile, generated from the input application trace data (refer to Section 3.1.3) queries a power model to estimate total system power draw during that phase, and utilizes phase timing data to arrive at the optimal frequency for the phases. The phase configuration is output to a file that can be loaded at runtime by a DVFS-instrumented application.

3.5.1. Intertask Frequency Selection — The first step in analyzing application traces within Green Queue is to determine whether the workload is balanced across MPI ranks. If the workload is balanced, the application is analyzed further for intratask DVFS. If it is imbalanced, it is a candidate for intertask DVFS.

Running on homogenous hardware, load imbalance within MPI applications typically 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 to that problem. Solutions to the load balancing problem are well-developed in the literature and range from solutions which involve modification of the algorithm or implementation of dynamic load balancing in the application. 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.

The load imbalance problem persists and is unlikely to disappear any time soon, resulting in excess computation on some ranks and imbalances within the computation distributed to MPI tasks within applications. Green Queue takes advantage of these excesses by measuring them then introducing lower clock frequencies to the CPUs that are hosting MPI ranks which are running at less than full capacity. The measurements of MPI behavior are collected using PSiNSTracer [18], 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.

$$excess_i = \frac{CPUTime_i}{MAX_{r=0}^{n}(CPUTime_r)} \tag{1}$$

$excess_i$ is therefore the ratio of the computation time of rank i to the most computationally intensive rank in the application. Note that Equation (1) 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 empirically in Section 4.1.

$$Freq_i = (excess_i \times (1 + p)) \times Freq_{max} \tag{2}$$

Combining Equations (1) and (2) yields an equation directly relating the MPI profiling measurements to the clock frequency selections.

$$\frac{Freq_i}{Freq_{max}} = \frac{CPUTime_i \times (1 + p)}{MAX_{r=0}^{n}(CPUTime_r)} \tag{3}$$

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

DOI: 10.1002/cpe

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. For the sake of simplicity, we currently round the frequency produced by Equation 2 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. On the other hand, 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 intense ranks together, pitting those ranks in competition with 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 this approach.

3.5.2. Intratask Frequency Selection — The most challenging problem addressed by this work is selection of an optimal frequency for a phase. The approach used within Green Queue is to take a phase characterization profile as input, then utilize a power model combined with performance measurements to arrive at an estimate of the optimal frequency for the given phase. This approach is based on the observations that (1) power is modeled more easily than performance and (2) very fine-grained performance is more easily measured than power. The result is an approach that combines power modeling with performance measurement.

Modeling Application Power Consumption Direct measurement of power draw for different application phases can be inaccurate and sometimes impossible because power usually can only be measured over time scales that are far larger than typical, smaller application phases. The power consumption measurement devices used in this work yield roughly one reading per second[24]; even state of the art devices increasing the measurement granularity by several orders of magnitude produce measurements at over time frames that are large compared to typical application phase length[25]. 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.

A basic power consumption model can be constructed directly from the `pcubed` benchmark set. Once traces are collected for the benchmarks, they 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 Sandy Bridge node with 15 different CPU frequencies. This problem is made worse by random power or performance fluctuations that can affect measurements and which force multiple collections of each data point, further increasing runtime.

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 `pcubed` space based on the problem formulation in Equation (4). In Equation (4), $l1\_p\_ins$, $l2\_p\_ins$ and $l3\_p\_ins$ are cache levels 1, 2 and 3 misses per instruction. $fprat$ is the 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) [26]. `gbm` is based on the idea that combined predictions made by an ensemble of weak, but fairly simple to construct, models can

DOI: 10.1002/cpe

give better overall predictions for the phenomenon that is being modeled. The weak models are constructed in stages and each new model is refined using some specified loss (or cost) function.

$$P_{sys} = f(freq, l1\_p\_ins, l2\_p\_ins, l3\_p\_ins, fprat, mops\_ins, fops\_ins, int\_dud, fp\_dud) \quad (4)$$

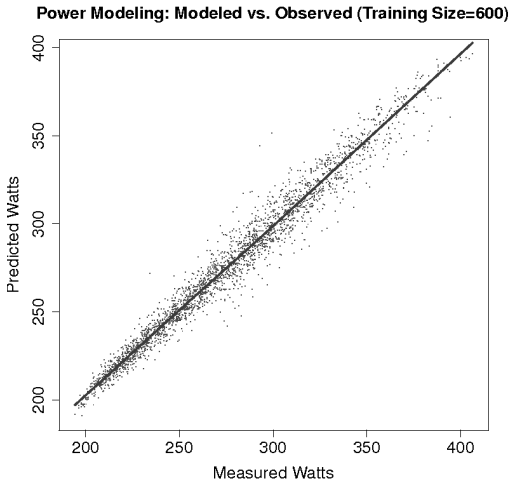


Figure 3. Modeled vs. Observed Total System Power Draw

To prevent overtraining, the `pcubed` benchmark set is supplemented with a set of 31 micro-benchmark kernels which are derived from a variety of different domains, are very prevalent in HPC applications, and have been extensively used in the past to investigate and evaluate intra-node auto-tuning techniques [20][21][22]. See Section 3.4.2 for further details on the composition of these micro-benchmark kernels. The statistical computing package R is used to train the power draw models based on the characterization profiles (and measured associated power draw) of `pcubed` and kernels. When Green Queue needs to select a frequency for an application phase, the models are loaded and fed the phase profile. The model returns predictions for the phase's power and performance at each available frequency which can then be used to estimate the optimal frequency. This method avoids the need to hand tune weights and can make use of `pcubed`, kernel training loops, or any other training inputs that are supplied.

Green Queue currently uses 1400 data points covering different working set sizes, kernel types and frequencies are measured for the kernels and are combined 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[†] from the total training set of 2800 samples is able to predict the power draw of the remaining 2200 samples with an absolute mean error percentage of 2.5%. Figure 3 shows the modeled versus measured values for total system power draw for all 2800 samples in the combined `pcubed` and micro-benchmark kernel 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 3.

Measuing Application and Loop Performance Performance measurements for the frequency selection algorithm are gathered by measuring loop runtimes within the application for every frequency. This is facilitated by a PEBIL-based binary instrumentation tool which inserts timers around loop entry and exit points within the application. The tool sums the time spent in each loop throughout the application run. To minimize the overhead of timing these runs and because

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

Prepared using cpeauth.cls
DOI: 10.1002/cpe

such loops cannot be phases (see Section 3.2.3), loops smaller than certain number of instructions executed per entry into the loop are excluded from timing. Using this exclusion rule at a threshold of 50000 instructions per loop entry, no significant overhead was measured in any of our tests.

Several optimizations are available which reduce the cost of additional application runs. First, many HPC applications are iterative; timing information can be expected to remain constant across iterations, implying that timing information can be gathered at a reduced number of iterations. Second, for HPC applications that are run on hundreds or thousands of cores, a weakly scaled data set can be used to measure loop runtimes on a single node or less, drastically reducing the resource investment of measuring loop times at each frequency.

After loop timing information for each frequency has been measured, Green Queue can combine measured timing information with the gbm-based power predictions discussed in Section 3.5.2 in order to estimate application energy consumption for each phase at each frequency and then select the frequency which optimizes energy for that phase.

4. EVALUATION

To determine the effectiveness of the intratask and intertask DVFS techniques used within Green Queue, experiments were run on a single rack of Gordon at the San Diego Supercomputer Center[27]. Each rack contains 64 nodes networked by QDR Infiniband arranged in a 3D torus. Each compute node contains two 8-core Intel Xeon E5-2670 processors for a total of 1024 cores. Each core has 32 KB L1 instruction cache and 32 KB L1 data caches as well as a 256 KB combined L2 cache. The 8 cores on a chip share a 20 MB L3 cache. Each node has 64 GB of memory. There are 15 frequency settings available, ranging from 1.2 GHz to 2.6 GHz, in addition to a Turbo Boost setting[28]. Turbo Boost was not enabled in these experiments because it causes unpredictable fluctuations in frequency, which in turn makes desgining reproducible experiments difficult.

Power was measured using an SNMP interface to two APC PDU's supplying power to the rack. A single rack is used in the experiments because of a limitation in our power consumption methodology, not because of a limitation in the techniques presented for phase detection and DVFS selection. The techniques presented can target any core count, but power, in this case, can only be measured at the rack level.

Green Queue was evaluated for 9 HPC applications and 3 benchmarks. These applications include several important scientific problems which use large amounts of dedicated node-hours on leadership class machines. Subject applications along with brief descriptions are provided in Table I. Where not otherwise noted, the input specifications/parameters for each of these applications for a 1024-core run were derived either via direct interaction with developers or experts on each application, through an application's web-portal, or from 1024-core benchmarking runs provided by NERSC [29]. From the NAS parallel benchmark suite, we use FT, CG, and MG at the class E problem size.

4.1. Intertask DVFS Experimental Results

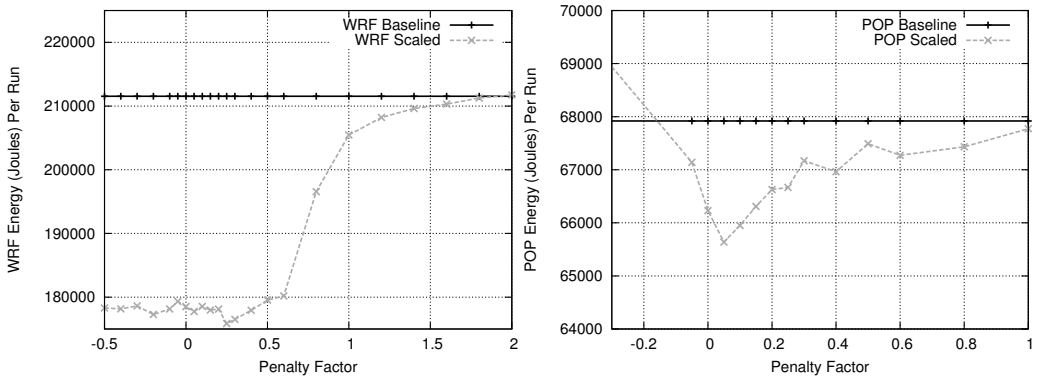
In examining our inter-task frequency scaling scheme, we begin by presenting some empirical results relating to the selection of the value of the penalty factor p from Equation (3) in Section 3.5.1. p is used within a strategy which attempts to exploit the imbalance of MPI applications to gain an energy 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 two full applications, POP and WRF, then use the results of those evaluations to select a single value for p that results in a generally 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.

DOI: 10.1002/cpe

Table I. Subject applications for Green Queue DVFS experiments.

Application	Description
Intratask Scaling	
MILC [30]	Code to study quantum chromodynamics (QCD), the theory of strong interactions of subatomic physics.
GTC [31]	Particle-in-cell application developed to study turbulent transport in magnetic fusion.
SWEEP3D [4]	Radiation transport code.
PSCYEE [32]	Parallel three dimensional finite-difference time-domain (FDTD) code for the Maxwell equations.
LBMHD [33]	Code that simulates homogeneous isotropic turbulence in dissipative magnetohydrodynamics.
NPBs [5]	NAS Parallel Benchmarks.
Intertask Scaling	
LAMMPS [34]	Molecular dynamics code.
HYCOM[‡] [35]	Hybrid isopycnal-sigma-pressure (generalized) coordinate ocean model code.
WRF [36]	Next-generation mesoscale numerical weather prediction system.
POP[§] [37]	3D ocean circulation model designed to study ocean climate system.

Figures 4a and 4b show our evaluations for a variety of choices for p. Both results are consistent with the conclusion that the choice of p as a small, positive value results in a nearly energy-optimal strategy. Green Queue therefore uses $p = 0.05$. With $p = 0.05$, we apply the intertask scaling methodology presented in Section 3.5.1 to two other applications – HYCOM and LAMMPS – shown in Table II. The results in table II show the energy reduction, performance loss and relative energy-delay product of applications measured with Green Queue compared to the application run in its normal fashion at the highest frequency. These applications show a range of imbalance properties from severe (LAMMPS[¶]) to moderate (WRF) to slight (POP and HYCOM).



(a) WRF intertask clock frequency scaling based on varying levels of agression in the frequency selection strategy.

(b) POP intertask clock frequency scaling based on varying levels of agression in the frequency selection strategy.

Figure 4

Table II. Intertask DVFS Energy Savings

Application	Energy Reduction	Delay Added	Relative Energy-Delay Product
LAMMPS	31.7%	2.3%	-30.1 %
WRF	16%	27.2%	6.8 %
HYCOM	0.9%	7%	6.1 %
POP	3.1%	0.8%	-2.3 %

4.2. *Intratask DVFS Experimental Results*

Several aspects of Green Queue's intratask DVFS strategy were evaluated. Experiments were run to measure the effect of varying the minimum phase size (see Section 3.2.2, disabling restoration

[‡]Run on 1001 cores, the closest valid configuration to 1024. MPI tasks are packed 16 per node onto the first 62 nodes, then 8 and 1 task per node respectively onto the two sockets of the 63rd node. The 64th node is left empty for all runs.
[§]Run two simultaneous cases of 480 cores on 64 nodes. MPI tasks are packed 16 cores per node onto the first 30 nodes of each half of the rack, leaving 4 empty nodes during all runs.
[¶]These imbalance properties are sensitive to the dataset used. That is, some datasets show severe imbalances while others do not.

DOI: 10.1002/cpe

of CPU frequency at the end of a phase (see Section 3.3.1) and to compare Green Queue with application unaware DVFS.

4.2.1. Varying Phase Granularity — We first explore the impact of varying the minimum allowable phase size. During phase selection, phases that are too small are eliminated as noise and a larger phase is assumed to contain them. The experiment is run for four applications: CG, MG, FT, and MILC. Results are shown in Figure 5.

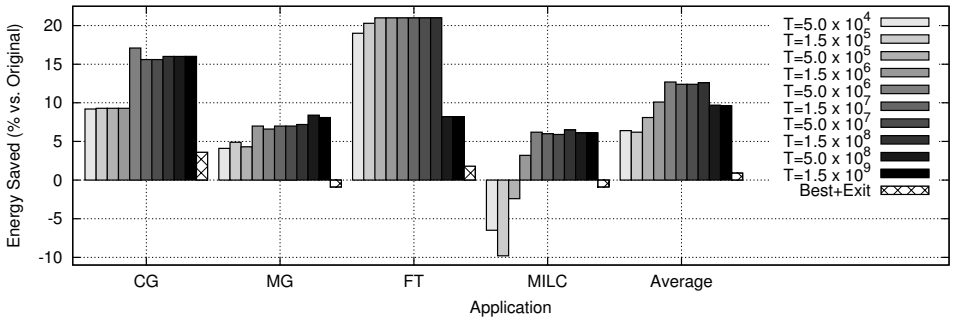


Figure 5. Green Queue Energy Savings with Various Minimum Phase Sizes

The results show that optimal phase size is different for different applications. This is because the energy savings that can be achieved in total depends on how much energy can be saved by scaling the next phase minus the overhead of making a frequency change. This tradeoff can result in a negative net energy savings as seen with MILC in Figure 5 at the most agressive phase granularities. The phase granularity that on average performs best (5 million instructions) is used to create frequency configurations for the remaining applications. The results are shown in Table III.

Table III. Intratask DVFS Energy Savings

Application	Energy Reduction	Delay Added	Relative Energy-Delay Product
CG	17.1%	9.5%	-9.2%
FT	21.0%	2.4%	-19.1%
MG	8.4%	8.7%	-0.4%
MILC	6.5%	13.0%	5.7%
PSCYEE	19.0%	8.5%	-12.1%
LBMHD	5.3%	7.1%	1.4%
GTC	4.8%	1.3%	-3.6 %
Sweep3D	0%	0%	0 %

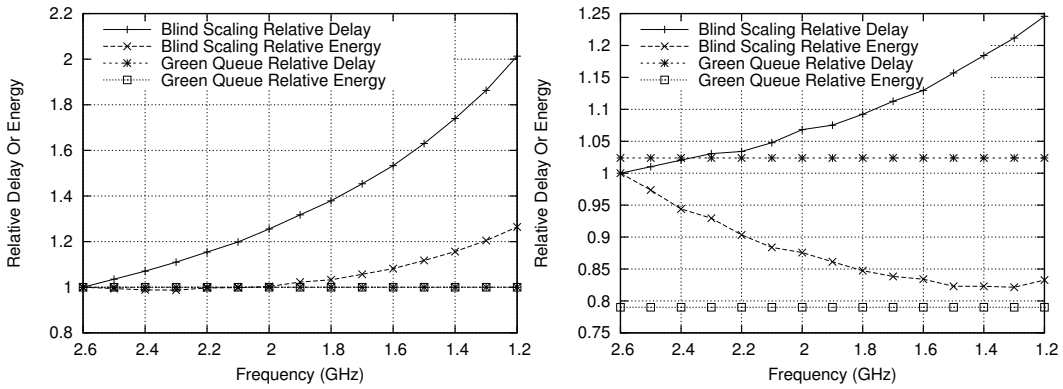
4.2.2. Disabling Frequency Restoration — When instrumenting a phase for runtime DVFS, in implementation there is a choice of whether or not to restore the CPU frequency to a default value (the maximum) when exiting a known phase. Restoring CPU frequency is to its maximum value upon exiting a known phase is the conservative route. If a region of code is not characterized by the analysis, the highest available frequency should preferred since this will avoid a performance penalty without known energy savings. Skipping a switch back to the maximum frequency on phase exit is to be optimistic about the coverage of the phase detection analysis. If coverage is complete, then the next phase will set its own frequency and an intermediate switch to the default frequency is avoided.

The the solid bars in Figure 5 are for experiments which skip setting the frequency upon phase exit. The hashed bars labeled ”Best+Exit” show the energy savings achieved using the ideal minimum phase size (5 million instructions) and when frequency is restored to the maximum frequency at phase exit points. In every case, disabling frequency restoration saves more energy than restoring frequency on phase exit. For MILC and MG, the overhead of restoring frequency at the end of each phase is enough to result in *increased* energy usage over the uninstrumented application. This indicates that coverage of the phase detection analysis is complete enough that having uncovered code run at possibly non-optimal frequencies is better than having to make additional frequency transitions at the end of each phase and in each case examined, was a valuable

DOI: 10.1002/cpe

optimization. The number of frequency transitions can be reduced by at least a factor of two with frequency restoration disabled and likely more since it will allow dynamic phase merging to occur.

4.2.3. Comparison With Blind Frequency Scaling — This work was partly motivated by experiments that demonstrated the possiblity that a frequency reduction could increase the total energy consumed by an application. This section compares those results with Green Queue to validate it's technique by demonstrating that Green Queue can achieve greater energy reductions than any single static frequency selection. The results are shown in Figures 6a and 6b.



(a) Comparison of Blind Scaling with Green Queue for Sweep3D (b) Comparison of Blind Scaling with Green Queue for FT

Figure 6

Sweep3D is useful to demonstrate that energy is not always reduced with a frequency reduction. Sweep3D consists of only a single identifiable phase. This is because Sweep3D hass several short inner loops contained in a stack of outer loops. Each inner loop is too short to effectively instrument as its own phase. Figure 6a shows that scaling the frequency below 2.0 GHz resulted in significant increases in total energy consumption. Green Queue selected 2.6 GHz as the optimal frequency for Sweep3D. The blind scaling experiments show that 2.4 and 2.3 GHz frequency configurations save 1.1% and 1.3% energy over Green Queue. This margin of error is acceptable. When energy differences between frequencies are very small, the performance penalty is likely to make the tradeoff less attractive, as evidenced by the 7% and 11% performance penalties shown by Sweep3D. Green Queue performs best when it can identify phases that can be scaled *without* losing signifcant performance.

The results for FT show that application aware DVFS is a significant improvement over blind scaling because of its identification of phases. In this case, not only does Green Queue save more energy than could be achieved running the application at any single frequency, it does so with a much smaller performance loss than would be needed to achieve similar results with blind scaling. With Green Queue, FT was able to achieve a 21% energy reduction with only a 2.4% penalty to performance. The best case that blind scaling can achieve is a 18% energy reduction and it suffers a 16% performance penalty.

4.3. Discussion and Future Work

Tables II and III summarizes the intratask and intertask results for all the subject applications. Figure 7 shows the power draw for both the baseline run and the Green Queue run for a subset of the applications. We present this graph as a visual aid show the dramatic nature of the effects on system power draw that can be accomplished with the Green Queue. Of all the applications that we considered for our intratask application-aware DVFS, FT saves the most energy, followed by PSCYEE. Both FT and PSCYEE are have parts that are memory-bound. Since Green Queue's phase discovery and DVFS selection methods utilize memory behavior as an important factor in detecting when a CPU is likely to have little computation to perform, we should expect Green

DOI: 10.1002/cpe

Queue to fare well on these applications. Energy saved for GTC and MILC, codes which use a substantial number of dedicated allocation hours on many leadership class machines, save 4.8% and 6.2% energy respectively.

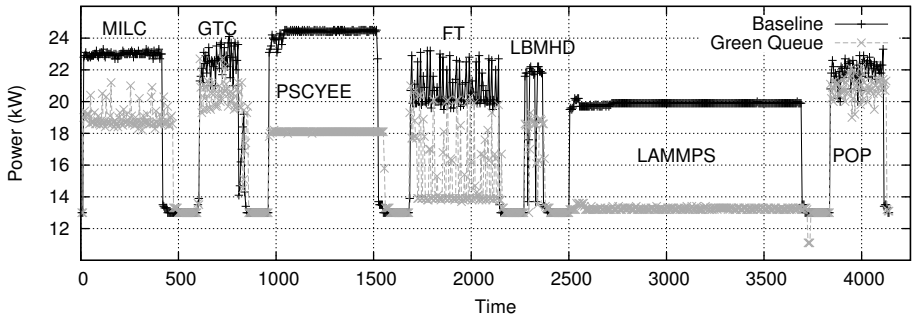


Figure 7. Baseline vs. Green Queue – power monitoring during application runs.

When we consider all the applications together, the average energy savings that we achieve with Green Queue is 12.1%. This improvement in overall energy savings comes at the expense of average performance loss of 7.9%. The maximum energy savings that we achieve with the intratask strategy is 21% for FT and this comes at the performance loss of 2.4%. The maximum energy savings from the internode strategy is 31.7% for LAMMPS, which comes with a performance penalty of 2.3%. Overall these results are encouraging. 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. The literature also points to what is known as the *cascade effect* [38], 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. Finally, using lower voltage-frequency states also improves the reliability and lifespan of processor hardware [39] and doing so judiciously helps both energy efficiency and the reliability of the system.

The proposed methods of analysis within Green Queue are tied to the application *binary* rather than the source code. In many ways this is an advantage, since solutions tied to source code would require rebuilding the application if the in cases where the source code is even available. However is also presents a limitation; modifications to the binary resulting even from small updates, or recompilation with a different compiler or flags results in a different binary that must be analyzed. In general, recomplilation with a different compiler or flags can result in drastically different power-performance behavior, validating the need for re-analyzing the binary. However, in some cases, much of the code may remain unchanged. Future work might address this issue by detecting which portions of the binary have changed and assessing the need to reanalyze the application.

This work has shown that the optimal phase size differs between applications. This is a result of the tradeoff between investing time (and therefore energy) to make a frequency switch and saving energy by running a phase at its optimal frequency setting. Green Queue is currently limited in its ability to analyze this tradeoff. Future work can achieve greater energy savings by calculating the projected energy savings of a frequency change and comparing it to the energy overhead of making the change. If two neighboring phases have only slightly different optimal frequencies, it may be best to run both phases at one frequency or the other to avoid the cost of a frequency transition.

In some applications, such as Sweep3D, there are several short phases in a loop. Each phase is too short to effectively instrument for DVFS on its own, so Green Queue merges them together, blending their behaviors. If there are significant differences between phases in a loop, increased energy savings may be possible if loop re-ordering is performed so that each phase is longer. A simple solution could inform users what sections of code have similar characteristics and if grouped together could be amenable to DVFS. A more advanced solution could be built into a complier to automatically re-order code so that larger phases can be achieved.

Other future work involves combining the techniques presented here with other energy saving strategies. The intertask and intratask techniques presented in this article could be combined

DOI: 10.1002/cpe

if the per-core contributions to power were accurately modeled and if the interactions between unsychronized or hetergenous tasks were understood in the context of DVFS. Current research at SDSC includes an investigation of job striping[40] which may also prove to be a good candidate for integration into Green Queue.

5. CONCLUSION

Energy bills have become a significant cost to high performance computing and data centers, upwards of the millions and tens of millions of dollars per year and rising. There is a clear need to mitigate this by understanding the tradeoffs involved between reducing energy usage and delaying time-to-solution. In this work we (1) presented Green Queue, a practical implementation of a system which facilitates understanding these tradeoffs, (2) investigated Dynamic Voltage-Frequency Scaling (DVFS) as a technique for reducing total energy consumption of MPI applications for high performance computing and showed that there is significant potential for energy and dollar savings, and (3) identified opportunities to reduce the energy consumption of particular HPC applications by up to 21% and 32% energy savings via intratask and intertask DVFS, respectively.

ACKNOWLEDGEMENTS

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 National Science Foundation grant: OCI #0910847, Gordon: A Data Intensive Supercomputer. 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).

REFERENCES

1. Top500. K computer, sparc64 viiifx 2.0 ghz, tofu interconnect November 2011. URL http://i.top500.org/system/177232.
2. Wire H. Mission possible – greening the hpc data center 2009.
3. Etinski M, Corbalan J, Labarta J, Valero M. Understanding the future of energy-performance trade-off via dvfs in hpc environments. *J. Parallel Distrib. Comput.* Apr 2012; **72**(4):579–590, doi:10.1016/j.jpdc.2012.01.006. URL http://dx.doi.org/10.1016/j.jpdc.2012.01.006.
4. ASCI Sweep3D. http://wwwc3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html.
5. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, *et al.*. The nas parallel benchmarks - summary and preliminary results. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, ACM: New York, NY, USA, 1991; 158–165, doi:10.1145/125826.125925. URL http://doi.acm.org/10.1145/125826.125925.
6. Freeh VW, Kappiah N, Lowenthal DK, Bletsch TK. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *J. Parallel Distrib. Comput.* Sep 2008; **68**(9):1175–1185, doi:10.1016/j.jpdc.2008.04.007. URL http://dx.doi.org/10.1016/j.jpdc.2008.04.007.
7. Rountree B, Lownenthal DK, de Supinski BR, Schulz M, Freeh VW, Bletsch T. Adagio: making dvs practical for complex hpc applications. *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, ACM: New York, NY, USA, 2009; 460–469, doi:10.1145/1542275.1542340. URL http://doi.acm.org/10.1145/1542275.1542340.
8. Choi K, Soma R, Pedram M. Dynamic voltage and frequency scaling based on workload decomposition. *Proceedings of the 2004 international symposium on Low power electronics and design*, ISLPED '04, ACM: New York, NY, USA, 2004; 174–179, doi:10.1145/1013235.1013282. URL http://doi.acm.org/10.1145/1013235.1013282.
9. Dhiman G, Rosing TS. Dynamic voltage frequency scaling for multi-tasking systems using online learning. *Proceedings of the 2007 international symposium on Low power electronics and design*, ISLPED '07, ACM: New York, NY, USA, 2007; 207–212, doi:10.1145/1283780.1283825. URL http://doi.acm.org/10.1145/1283780.1283825.
10. Isci C, Contreras G, Martonosi M. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, IEEE Computer Society: Washington, DC, USA, 2006; 359–370, doi:10.1109/MICRO.2006.30. URL http://dx.doi.org/10.1109/MICRO.2006.30.
11. Wu Q, Martonosi M, Clark DW, Reddi VJ, Connors D, Wu Y, Lee J, Brooks D. A dynamic compilation framework for controlling microprocessor energy and performance. *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, IEEE Computer Society: Washington, DC, USA, 2005; 271–282, doi:10.1109/MICRO.2005.7. URL http://dx.doi.org/10.1109/MICRO.2005.7.

12. Le Sueur E, Heiser G. Dynamic voltage and frequency scaling: the laws of diminishing returns. *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, USENIX Association: Berkeley, CA, USA, 2010; 1–8. URL http://dl.acm.org/citation.cfm?id=1924920.1924921.
13. Freeh VW, Lowenthal DK. Using multiple energy gears in mpi programs on a power-scalable cluster. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, ACM: New York, NY, USA, 2005; 164–173, doi:10.1145/1065944.1065967. URL http://doi.acm.org/10.1145/1065944.1065967.
14. Snowdon DC, Le Sueur E, Petters SM, Heiser G. Koala: a platform for os-level power management. *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, ACM: New York, NY, USA, 2009; 289–302, doi:10.1145/1519065.1519097. URL http://doi.acm.org/10.1145/1519065.1519097.
15. Snowdon DC, Petters SM, Heiser G. Accurate on-line prediction of processor and memoryenergy usage under voltage scaling. *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, ACM: New York, NY, USA, 2007; 84–93, doi:10.1145/1289927.1289945. URL http://doi.acm.org/10.1145/1289927.1289945.
16. McCullough JC, Agarwal Y, Chandrashekar J, Kuppuswamy S, Snoeren AC, Gupta RK. Evaluating the effectiveness of model-based power characterization. *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, USENIX Association: Berkeley, CA, USA, 2011; 12–12. URL http://dl.acm.org/citation.cfm?id=2002181.2002193.
17. Laurenzano MA, Meswani M, Carrington L, Snavely A, Tikir MM, Poole S. Reducing energy usage with memory and computation-aware dynamic frequency scaling. *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, Springer-Verlag: Berlin, Heidelberg, 2011; 79–90. URL http://dl.acm.org/citation.cfm?id=2033345.2033356.
18. Tikir M, Laurenzano M, Carrington L, Snavely A. Psins: An open source event tracer and execution simulator for mpi applications. *Euro-Par 2009 Parallel Processing* 2009; :135–148.
19. Laurenzano M, Tikir M, Carrington L, Snavely A. Pebil: Efficient static binary instrumentation for linux. *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, 2010; 175 –183, doi:10.1109/ISPASS.2010.5452024.
20. Hartono A, Norris B, Sadayappan P. Annotation-based empirical performance tuning using orio. *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009; 1 –11, doi:10.1109/IPDPS.2009.5161004.
21. Balaprakash P, Wild S, Norris B. SPAPT: Search problems in automatic performance tuning 2011; (ANL/MCS-P1872-0411). URL http://www.mcs.anl.gov/uploads/cels/papers/P1872.pdf, 5.
22. Tiwari A, Chen C, Chame J, Hall M, Hollingsworth J. A Scalable Auto-Tuning Framework for Compiler Optimization. *IPDPS'09*, Rome, Italy, 2009.
23. Polybench. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.
24. WattsUp? Meters. https://www.wattsupmeters.com/.
25. Bedard D, Lim MY, Fowler R, Porterfield A. PowerMon: Fine-grained and integrated power monitoring for commodity computer systems. *IEEE SoutheastCon 2010*, 2010; 479 –484, doi:10.1109/SECON.2010.5453824.
26. Generalized Boosted Regression Models. http://cran.r-project.org/web/packages/gbm/.
27. SDSC. Gordon user guide: Technical summary March 2012. URL http://www.sdsc.edu/us/resources/gordon/.
28. Intel. Intel turbo boost technology – on-demand processor performance August 2012. URL http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.
29. The National Energy Research Scientific Computing Center. http://www.nersc.gov.
30. MIMD Lattice Computation Collaboration. http://www.physics.indiana.edu/~sg/milc.html.
31. Lin Z, Rewoldt G, Ethier S, Hahm TS, Lee WW, Lewandowski JLV, Nishimura Y, Wang WX. Particle-in-cell simulations of electron transport from plasma turbulence: recent progress in gyrokinetic particle simulations of turbulent plasmas. *Journal of Physics: Conference Series* 2005; **16**(1):16. URL http://stacks.iop.org/1742-6596/16/i=1/a=002.
32. Andersson U. Parallelization of a 3d fd-td code for the maxwell equations using mpi. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, *Lecture Notes in Computer Science*, vol. 1541, Kgstrm B, Dongarra J, Elmroth E, Wasniewski J (eds.). Springer Berlin / Heidelberg, 1998; 12–19. URL http://dx.doi.org/10.1007/BFb0095313, 10.1007/BFb0095313.
33. Williams S, Carter J, Oliker L, Shalf J, Yelick K. Optimization of a lattice boltzmann computation on state-of-the-art multicore platforms. *J. Parallel Distrib. Comput.* Sep 2009; **69**(9):762–777, doi:10.1016/j.jpdc.2009.04.002. URL http://dx.doi.org/10.1016/j.jpdc.2009.04.002.
34. Large-scale Atomic/Molecular Massively Parallel Simulator. http://lammps.sandia.gov/.
35. HYbrid Coordinate Ocean Model or HYCOM). http://www.hycom.org.
36. Weather Research and Forecasting (WRF) Model. http://www.wrf-model.org/.
37. Parallel Ocean Program. http://climate.lanl.gov/Models/POP/.
38. 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.
39. Nightingale EB, Douceur JR, Orgovan V. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer pcs. *Proceedings of the sixth conference on Computer systems*, EuroSys '11, ACM: New York, NY, USA, 2011; 343–356, doi:10.1145/1966445.1966477. URL http://doi.acm.org/10.1145/1966445.1966477.
40. Breslow A, Porter L, Tiwari A, Laurenzano M, Carrington L, Tullsen D, Snavely A. Job Striping: The Right Way to Schedule a Supercomputer. *In Submission*.