Title
Green queue: a framework for selecting energy optimal DVFS configurations in large scale MPI applications

Permalink
https://escholarship.org/uc/item/3pd00857

Author
Peraza, Joshua

Publication Date
2012-01-01

Peer reviewed|Thesis/dissertation
Green Queue: A Framework for Selecting Energy Optimal DVFS Configurations in Large Scale MPI Applications

A thesis submitted in partial satisfaction of the requirements for the degree
Master of Science

in

Computer Science

by

Joshua Peraza

Committee in charge:

Professor Allan Snavely, Chair
Professor Tajana Rosing
Professor Steven Swanson

2012
The thesis of Joshua Peraza is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

________________________________________

________________________________________

________________________________________

Chair

University of California, San Diego

2012
# TABLE OF CONTENTS

Signature Page .......................................................... iii

Table of Contents ........................................................ iv

List of Figures ........................................................... vi

List of Tables ............................................................ vii

Acknowledgements ......................................................... viii

Abstract of the Thesis ................................................... ix

Chapter 1 Introduction .................................................. 1

Chapter 2 Background and Related Work .............................. 3

Chapter 3 Green Queue .................................................. 8
  3.1 Trace Collection .................................................. 8
    3.1.1 PSiNSTracer ............................................... 10
    3.1.2 PEBIL ..................................................... 10
  3.2 Trace Analysis .................................................. 11
    3.2.1 Trace Database ........................................... 11
    3.2.2 Interprocedural Loop Analysis ......................... 11
    3.2.3 Phase Detection ......................................... 14
  3.3 Deployment ..................................................... 15
    3.3.1 DVFS Instrumentation .................................. 16
    3.3.2 SecureScaler ............................................ 16
  3.4 Intertask Frequency Selection .................................. 16
  3.5 Application Characterization .................................. 17
    3.5.1 pcubed .................................................. 17
    3.5.2 Kernels ................................................ 18
  3.6 Intratask Frequency Selection ................................ 18
    3.6.1 Geometric Distance ................................... 18
    3.6.2 Power and Performance Modeling ....................... 19
    3.6.3 Power Modeling and Performance Measurement ....... 20

Chapter 4 Evaluation .................................................. 22
  4.1 Intertask DVFS ................................................ 22
  4.2 Intratask DVFS ................................................ 24
    4.2.1 Varying Phase Granularity ......................... 24
    4.2.2 Disabling Frequency Restoration .................. 25
    4.2.3 Comparison With Application Unaware DVFS ...... 26
LIST OF FIGURES

Figure 3.1: The Green Queue Framework ........................................... 9
Figure 4.1: WRF energy consumption with various penalty settings ....... 23
Figure 4.2: POP energy consumption with various penalty settings ....... 24
Figure 4.3: Green Queue Energy Savings with Various Minimum Phase Sizes 25
Figure 4.4: Comparison of Blind Scaling with Green Queue for CG ...... 27
Figure 4.5: Comparison of Blind Scaling with Green Queue for FT ...... 27
Figure 4.6: Comparison of Blind Scaling with Green Queue for Sweep3D .. 28
LIST OF TABLES

Table 4.1: Intertask DVFS Energy Savings . . . . . . . . . . . . . . . . . . 23
Table 4.2: Intratask DVFS Energy Savings . . . . . . . . . . . . . . . . . . 25
ACKNOWLEDGEMENTS

I would like to thank Ananta Tiwari, Michael Laurenzano, and Laura Carrington for their guidance in the completion of this thesis.
ABSTRACT OF THE THESIS

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

by

Joshua Peraza

Master of Science in Computer Science

University of California, San Diego, 2012

Professor Allan Snavely, Chair

This thesis 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. Green Queue makes use of both intertask and intratask DVFS techniques. The intertask technique targets applications where workload is imbalanced, reducing frequency for ranks with smaller 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 performance response to frequency changes. Green Queue is evaluated at 1024 cores on Gordon, the newest supercomputer at the San Diego Supercom-
puter Center, utilizing Intel Sandybridge CPUs. Green Queue demonstrates up to 21% and 32% energy savings for intratask and intertask DVFS strategies, respectively.
Chapter 1

Introduction

As the size of supercomputers increases, so too does their thirst for power. With the move towards exascale computing, the number of processor cores in a system is ever increasing and each puts additional power requirements on the system both for operation and for cooling. The cost of this additional power has become substantial. The K computer, currently heading the Top500, consumes 12.7 megawatts of power[27]. At 10¢ per kWh, the cost of electricity could reach up to $11 million annually. Over the lifetime of the system, the cost of energy can exceed the initial hardware cost of the system[29]. This produces an incentive to reduce the energy consumption of these numerous, massive systems.

Using energy efficiently requires being able to control how much energy is committed to a task and understanding what benefit can be expected from its use. This grants the opportunity to make informed trade-offs in performance and energy consumption. In this thesis, Dynamic Voltage-Frequency Scaling (DVFS) is the mechanism for controlling this trade-off. Many researchers have used DVFS to either reduce total energy costs or to cap dynamic power consumption. DVFS allows a user to throttle the speed of a CPU in exchange for reduced power consumption. The key challenge to using DVFS effectively is to understand what the effect on both power and performance will be for a change in CPU frequency. Application unaware DVFS will 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[13].
This thesis explores techniques for determining optimal frequency configurations for pure MPI applications. In this thesis, optimality refers to the configuration that minimizes total energy consumption but other metrics derived from delay and power, such as energy-delay product, can be minimized similarly. This thesis culminates in the development of Green Queue, a framework supporting the analysis and instrumentation of MPI programs for DVFS. Green Queue was evaluated with several 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 20% with intratask DVFS and up to 32% with intertask DVFS.
Chapter 2

Background and Related Work

DVFS has been explored by several researchers as a mechanism for reducing energy consumption. DVFS strategies have fallen into two broad categories: intertask and intratask. Green Queue utilizes both intertask and intratask strategies for DVFS.

Intertask DVFS capitalizes on the idea that an application is only as fast as its critical path. Any computation not on the critical path can be scaled down without impacting performance. Rountree et al. computed slack time per task at runtime in order to only slow down code not on the critical path [22]. Freeh et al. used the relative compute times of different MPI ranks to scale down the frequency on the least busy ranks[14]. For simplicity, Green Queue utilizes a strategy similar to this for intertask DVFS. Efforts have instead been focused on performing intratask DVFS in applications with balanced workloads.

Intratask DVFS attempts to reduce energy consumption by varying the CPU frequency within a single task as it moves between program phases. With intratask DVFS, frequency reductions are never free, but they may be cheap. If a region of code is heavily tasking the memory system, 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 results in sub-linear performance loss, allowing for net energy savings.

For much of the past decade of DVFS research, this has not been necessary to minimize energy. When CPU power dominated total system power, a
decrease in frequency resulted in a net energy savings, despite a 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[11]. Other research has followed in the same vein [12][16][30].

There are two high level questions any strategy must answer to perform intratask DVFS: When should frequency be adjusted and to what frequency? The research discussed so far had similar solutions to the second question. They develop analytical models for the balance of on-chip vs off-chip workloads and fill the parameters with the model. Then the frequency is scaled to match some target performance loss. Strategies for solving the first question are somewhat more variable.

The first question is the problem of phase identification. A phase is a window in the execution of a program where relevant characteristics are homogeneous. In the case of DVFS, the homogeneous property is optimal frequency. The brevity and number of phases in real programs precludes determining the optimal frequency for all phases experimentally.

Simple strategies 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 [11][12]. 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 will continue into the next. If phases are short lived then the frequency selected at each time slice will no longer be optimal.

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[16]. In this strategy, phases are discretized into ranges 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 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 function or loop granularity[30]. The intuition behind this strategy is that programs are structured and phases will tend to be attached 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 dynamic CPU power dominates the total 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, even for 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.

This assumption no longer holds. As CPU voltage and feature size shrink, so too does the range of dynamic power and opportunity for power savings via DVFS. Leakage power 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[13][19].

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 future systems requires that total system power is considered and that we have more accurate performance models in order to take advantage of less than linear performance loss whenever possible.

Freeh et al use a strategy that accounts for total system power[15]. First a program is divided into phases using traces which report operations per cache miss for blocks of code. Phases are formed by merging blocks that have similar
operations per cache miss. As with previous strategies, the model assumes that operations per cache miss correlates strongly with optimal frequency. However, Freeh et al. do not attempt to estimate the power or performance response to changes in frequency. Instead, total energy consumption for several frequency configurations is measured and the best configuration is used. In order to avoid a number of application runs exponential in the number of phases, phases are prioritized according to the operations per miss metric. If running a phase at a lowered frequency does not reduce energy consumption, then it is assumed that DVFS will not improve energy usage for any phases with more operations per cache miss.

A primary weakness of this strategy is that it still requires a $O(num\_phases)$ runs to select a frequency configuration. A modeling approach is attractive because real applications may have many phases. Modeling allows predictions to be made about an applications power and performance response to DVFS without having to run the application numerous times, diminishing the net energy savings that will actually be achieved.

Snowdon et al. use hardware counters to model both system power consumption and performance response to frequency scaling\[24\][25]. In the model, a training set of hand picked applications are used to correlate hardware counter measurements with power and performance response to frequency scaling. The highest correlated counters are then used at runtime to estimate both power and performance response to potential frequency changes. The model selects optimal frequency based on a policy chosen by the user.

This model, however, is not portable. Different CPU models provide different counters which can force the model to change. 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[21], but performance 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[18]. These signatures are compared against signatures
collected from a training suite. The training suite is artificially generated to cover the characterization space.

There are several advantages to this approach, but they come with their own hurdles. First, because the model is in software, it is not restricted by the availability of hardware counters. The domain of possibly relevant software properties is boundless. This allows modeling behaviors difficult or impossible to capture with hardware counters, but identifying what these properties are is an arduous task. Second, an artificial benchmark suite provides control over the training space, but it is prone to systematic behavior. A researcher can decisively explore the application space, but the model loses variability in properties that aren’t specifically accounted for. This leads to overtraining on these invisible properties which have more variability in real applications.
Chapter 3

Green Queue

The Green Queue project will culminate in a production quality framework of tracing, database, and analysis tools to transparently run user applications with reduced energy consumption using a variety of techniques. This thesis will focus on the intertask and intratask DVFS aspects of the Green Queue. Figure 3.1 gives an overview of the Green Queue framework.

The first time an application is submitted to Green Queue, it is examined by several analysis tools. Traces are gathered and analyzed. Meanwhile, the submitted application is run without modification or DVFS. The next time the application is submitted to the Green Queue with a similarly sized data set, the analysis from its previous submission is retrieved from a database. The analysis is used to instrument the application with DVFS controls and then run at a reduced energy cost.

3.1 Trace Collection

The first step in an applications journey through the Green Queue is trace collection. At this stage, the tracing framework performs test runs of the application using binary instrumentation tools to measure application behavior.
Figure 3.1: The Green Queue Framework
3.1.1 PSiNSTracer

PSiNSTracer is a tool that provides instrumented wrappers for MPI calls[26]. LD_PRELOAD can be used to insert calls instrumented with timers into the application. The timers sum the total time spent inside various MPI calls. The results are written to a file and can be used to determine amounts of communication vs CPU time as well as imbalances of CPU time between MPI ranks. When imbalances beyond a threshold are measured, the application is identified as a candidate for intertask DVFS.

3.1.2 PEBIL

PEBIL[17] is a binary instrumentation tool for x86-Linux platforms that provides a host of tools used by Green Queue. First, PEBIL provides a basic block counter. This tool generates a summary of all basic blocks in an application, including number and types of instructions. PEBIL 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. When an application instrumented with the basic block counter is executed, the result is a count of the number of times each block was visited during the execution of the application as well as counts for the number of times each loop head was entered.

The results of the basic block counter can be used to identify the hot code in the application. PEBIL then uses a second tool to instrument the hot blocks for cache simulation. PEBIL’s cache simulator is capable of simultaneously simulating multiple cache structures with up to three levels. Each cache level supports configurable total size, associativity, line size. The tool supports both inclusive and exclusive cache policies.
3.2 Trace Analysis

Once trace collection is complete, Green Queue moves on to analysis. In this stage, Green Queue uses trace information to reconstruct the application, create characterizing signatures for each phase, select an optimal frequency configuration and refine the result for efficient practical implementation.

3.2.1 Trace Database

The trace database is at the center of any analysis module in the Green Queue toolkit. The trace database is a clean interface to the trace data gathered for an application. On the backend, data is stored in an SQL database to allow for flexible development of queries. A java class frontend allows for easy re-use of many common high level queries across the several analysis modules in Green Queue and other projects. The database allows querying and aggregating data by function, loop, or basic block and selection of data particular to specific ranks or simulated cache systems.

3.2.2 Interprocedural Loop Analysis

An interprocedural analysis is critical to the 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 are not considered in the analysis, this characterization can be arbitrarily bad. Consider the loop below:

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

If functions are not considered, this may be characterized as a very memory intensive loop and the analysis would select a low frequency. The actual behavior depends on what the function \( f \) does. If \( f \) works the CPU more heavily than memory and \( f \) has a large number of instructions, then the loop would be best run at a higher frequency.
A second motivation for interprocedural analysis is the detection of phases in modular programs. To ensure that DVFS is used effectively, only phases larger than a threshold number of dynamic instructions are considered. Now consider this loop:

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

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, this loop may actually form a substantial phase.

The structural analyzer is a module developed to support interprocedural analysis and to provide a mechanism for high level navigation of the application traces. The module 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. Indiscriminate function inlining can cause an explosion in memory usage. Each time a function is inlined, the data structures accounting for its trace data would be duplicated at each point it is inlined. For many small, insignificant functions, inlining does not actually improve the accuracy of phase characterization. The number of dynamic instructions that they contribute is too small to affect the code that calls them. So, the inlining algorithm simply prunes functions whose dynamic instruction count is below a threshold.

A second concern when performing inlining is handling cycles in the call graph resulting from direct or indirect 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. This simplification has not been a problem in any experiments so far.

The algorithm used to actually perform inlining 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
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

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 wholly into the caller function, the caller function’s statistics may become inflated. In the absence of sufficient information in the trace data to guarantee accurate attribution of information, an assumption is made that the portion of instructions
attributable to any caller of a function is proportional the number of times that
 caller called the function. This information is easily available from the basic block
counter trace data. The total number of times the function is called from any
block is summed and compared to the number of times this particular basic block
made the function call.

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 because it does not effect the
analysis or it is a root function (e.g. main). The analysis may fail to determine
the call sites for a function if it was called via a function pointer, rather than by
name. This is expected to be rare and has so far not effected any analyses.

3.2.3 Phase Detection

In prior work, phases have often been identified by sudden changes in ap-
lication characterizations. The approach presented here differs markedly in that
no assumption is made about what properties demark a phase. 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. Consequently, frequency selection occurs midway through phase detection.

To bootstrap the phase detection algorithm, an assumption is made that
an inner most loop is homogeneous. 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 phases and if a phase is too short, the cost
of a phase switch may surpass the benefit of running at the optimal 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 number of loop head entries. 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 instrumentation code to be placed higher in the loop hierarchy and produce less overhead.

Once phase entry points have been moved to sufficiently sized loops, the frequency selection analysis is performed to mark each phase with an optimal frequency. More on this process is described in section 3.6.

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.

Currently, the trace data does not have sufficient information to determine in what order sibling loops are executed. Consequently, there may still be neighboring phases with the same frequency. This is addressed with dynamic phase merging, discussed further in section 3.3.1.

### 3.3 Deployment

The result of phase detection and frequency selection is a frequency configuration file. More infrastructure is needed to employ DVFS in an application on a production system. First, the application must be modified to enact frequency changes on phase entries. However, in a production system, DVFS is a privileged operation. A layer of indirection is needed to securely mediate frequency requests.
3.3.1 DVFS Instrumentation

PEBIL is used to instrument an application for DVFS. At instrumentation time, only phase entry points are programmed into the application. PEBIL uses a list of loop heads to identify the phase entries and inserts calls into an instrumentation library. At runtime, the library loads a frequency configuration mapping phases to frequencies. This allows frequency settings to be modified without re-instrumentation of the application. The instrumentation can track statistics about the performance of the DVFS configuration such as the number of frequency switches and also performs dynamic phase merging. The instrumentation tracks the current frequency. Upon entering a phase, the instrumentation checks if the new frequency being requested is the same as the current frequency. If the frequency is the same, control is immediately passed back to the application to minimize the overhead of instrumentation.

3.3.2 SecureScaler

The second component of providing DVFS on a production machine is a mediator for frequency requests. Green Queue uses SecureScaler to mediate DVFS requests. SecureScaler is a daemon that makes frequency selections on behalf of a client process and allows system administrators to insert security policy between user applications and the kernel such as only allowing certain users access to DVFS. SecureScaler uses Unix domain sockets to accept frequency requests from client applications.

3.4 Intertask Frequency Selection

The first step in analyzing application traces is to determine whether the workload is balanced across cores. If the workload is balanced, the application is analyzed further for intratask DVFS. If it is imbalanced, it is a candidate for intertask DVFS.

The intertask DVFS strategy used by Green Queue is to scale frequency for a core in proportion to its computation workload compared to the busiest core.
However, DVFS is only configurable per socket, not per core, so instead Green Queue scales frequency to the maximum frequency chosen for any core on the socket. Computation workload is estimated using timing information gathered by PSiNSTracer to be the amount of time the code spends outside of MPI calls. Then, a penalty is applied to the ratio to avoid scaling frequency so low that the workload on the scaled cores overtakes the critical path in compute time. The formula to compute a target frequency $F_i$ for core $i$ is shown.

\[
F_i = \min\left( F_{\text{MAX}}, F_{\text{MAX}} \cdot (1 + p) \cdot \frac{\text{Workload}_i}{\text{Workload}_{\text{max}}} \right)
\]

The most aggressive strategy would set the penalty $p$ to 0. As $p$ is increased cores are scaled less aggressively until all cores are at the maximum frequency.

### 3.5 Application Characterization

During frequency selection, models are used to make predictions about power or performance of code at different frequency settings. These models are learned from characterization profiles collected from sets of training applications. While learning a model, training applications are analyzed similarly to client applications, but quit the analysis once a characterization profile is created for them. The time and power response to different frequency settings is directly measured for the training application and associated with the characterization profile. A characterization profile in Green Queue 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 def-use distances for integer and floating point operations. Two sources of training applications are used.

#### 3.5.1 pcubed

The first training set used is pcubed [18]. 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. pcubed allows controlled exploration
of the characterization space, however, while \texttt{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 \texttt{pcubed} alone can accurately predict the entire parameter space of \texttt{pcubed} benchmarks with a comparatively small number of data points while still mispredicting for real applications.

### 3.5.2 Kernels

To supplement \texttt{pcubed} a suite of 31 HPC kernels[8] are added to the training set. These kernels add variety and complexity to the training set to help prevent models from becoming overtrained to \texttt{pcubed}'s particular loop idioms.

### 3.6 Intratask Frequency Selection

The most challenging problem addressed by this thesis is selection of an optimal frequency for a phase. The DVFS analyzer is responsible for implementing Green Queue's frequency selection policy. It takes a code profile, generated from a function or loop summary as input and outputs the optimal frequency for the code region based on the model. Several approaches to frequency selection were explored.

#### 3.6.1 Geometric Distance

The first strategy considered, taken from Laurenzano et al.[18], is to make frequency selections based on what the optimal frequency is for the nearest benchmark in the training set. The training set for this strategy includes only \texttt{pcubed} benchmarks. The distance metric between two code profiles is a weighted geometric distance on the properties of the characterization vector and the weights are hand selected based on the range of values a property can take and the estimated importance of the property in determining frequency. The distance between the input application and every benchmark is measured and the benchmark with the
smallest distance is chosen. The power and performance behavior for the benchmark has been measured at all available frequencies so an optimal frequency can be chosen. The strength of this method is that it can avoid discontinuities in the power and performance curves that may be difficult to model. However, this method requires accurate power and performance measurements or predictions for many points throughout the characterization space. Otherwise, the chosen point may have very different characteristics than the application under consideration. A key failure of this strategy is that it does not naturally reflect the difference in relevance between properties in the characterization vector. A small difference in one property may have a much greater effect than a large difference in another. Weights can be assigned to each property in the vector according to their relevance, but it is not straightforward how these weights are to be chosen.

3.6.2 Power and Performance Modeling

The next strategy considered is to use a loop’s characterization profile to model power and performance. The modeling technique chosen for Green Queue is the gradient boosting method[3]. R is used to train the model on the profile, power, and time data for all of the benchmarks and can be saved to a file for later reuse. When Green Queue needs to select a frequency for an application loop, the models are loaded and fed the loop profile. The model returns predictions for the loop’s power and performance at each available frequency which can then be used to estimate the optimal frequency. This strategy represents a significant improvement over the first method if power and performance are modeled accurately. This method avoids the need to hand tune weights and can make use of both pcubed and kernel training loops. However, this method does not currently capture the behavior of all applications. There has been success in prior research for modeling power on modern systems[21]. However, runtime performance is more difficult to predict because it depends on application properties not fully captured by the characterization vector.
3.6.3 Power Modeling and Performance Measurement

Freeh et al. select frequency configurations by direct measurement\cite{15}. However, they note that they make use of a heuristic to avoid a large number of application runs. If there are \( l \) phases and \( f \) available frequencies, then an exhaustive search requires \( f \) runs per phase for a total of \( f \times l \) runs. The frequency of only a single phase can be modified at a time because power is measured at large granularity. A single phase may be too short to measure accurately. The heuristic used prioritizes phases by their estimated propensity for minimal performance loss with DVFS reducing the number of runs to \( O(l) \). Real applications may have many phases making a linear number of runs an expensive investment. Furthermore, performance response to frequency has been found to be difficult to predict generally with any simple metric, resulting in possibly significant missed energy savings.

The final approach to frequency selection observes that while power is modeled well and performance is not, performance can be measured at a fine granularity where power cannot. The result is an approach that combines the two. PEBIL is used to build a loop timer tool. The tool uses \texttt{rdtsc} to place timers at the entry and exit of selected loops. The tool sums the time spent in each loop throughout the application run. To minimize any overhead from the instrumentation, loops are chosen with a minimum size using the same method described in section 3.2.3. At a minimum loop size of 50000 instructions per loop entry, no significant overhead was measured. To measure performance response to frequency, the application is run once at each available frequency. The loops that are most amenable to DVFS will show less degradation in performance at lower frequency.

A couple of optimizations can be made to reduce the cost of additional application runs. First, many HPC applications are iterative; timing information will be similar for most of the iterations so timing information can be gathered at a reduced number of iterations. Second, HPC applications will be run on hundreds or thousands of cores. While gathering loop timing information, a weak scaled data set can be used to measure loop times on a single node, drastically reducing the energy investment of measuring loop times at each frequency.
After loop timing information for each frequency has been measured, Green Queue can combine the information with power predicted from R gbm models to estimate energy consumption at each frequency and select the optimal configuration.
Chapter 4

Evaluation

Experiments were run on a single rack of Gordon at SDSC[23]. The 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 and 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 and 85 GB/s memory bandwidth. There are 15 frequency settings available, ranging from 1.2 GHz to 2.6 GHz, in addition to a Turbo Mode setting. Turbo Mode was not used in these experiments.

Power was measured using an SNMP interface provided by two APC PDU’s supplying power to the rack.

Green Queue was evaluated for 12 different HPC applications and benchmarks: MILC [6], GTC [20], Sweep3D [2], PSCYEE [9], LBMHD [28], CG [10], FT [10], MG [10], LAMMPS [5], HYCOM [4], WRF [1], and POP [7].

4.1 Intertask DVFS

Frequency selection for unbalanced workloads is controlled by a penalty factor $p$ in the formula:

$$F_i = \min \left( F_{MAX}, F_{MAX} \cdot (1 + p) \cdot \frac{Workload_i}{Workload_{max}} \right)$$
To determine a good value for $p$, energy consumption is measured for two applications for several values of $p$, shown in Figures 4.1 and 4.2. POP demonstrates that a small penalty is useful to avoid overscaling. As the size of the penalty increases, less scaling is done and the energy consumption approaches the baseline. Using these results, a penalty of $p = 0.05$ is chosen to use for experiments with LAMMPS and HYCOM. The results in Table 4.1 show the energy reduction, performance loss and relative energy-delay product of applications measured with Green Queue compared to the application run with no instrumentation at the highest frequency.

**Table 4.1: Intertask DVFS Energy Savings**

<table>
<thead>
<tr>
<th>Application</th>
<th>Energy Reduction</th>
<th>Delay Added</th>
<th>Relative Energy-Delay Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMMPS</td>
<td>31.7%</td>
<td>2.3%</td>
<td>-30.1 %</td>
</tr>
<tr>
<td>WRF</td>
<td>16%</td>
<td>27.2%</td>
<td>6.8 %</td>
</tr>
<tr>
<td>HYCOM</td>
<td>0.9%</td>
<td>7%</td>
<td>6.1 %</td>
</tr>
<tr>
<td>POP</td>
<td>3.1%</td>
<td>0.8%</td>
<td>-2.3 %</td>
</tr>
</tbody>
</table>
4.2 Intratask DVFS

Several aspects of Green Queue’s intratask DVFS strategy were evaluated. Experiments were run to measure the effect of varying the minimum phase granularity, disabling restoration of CPU frequency at the end of a phase and to compare Green Queue with application unaware DVFS.

4.2.1 Varying Phase Granularity

The first experiment explores 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 4.3.

The results show that optimal phase size is different for different applications. This is because the energy savings that can be achieved depend 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 at the most aggressive phase granularities.

The phase granularity that on average performs best (5000000 instructions) is used to create frequency configurations for the remaining applications. The results are shown in Table 4.2.
Figure 4.3: Green Queue Energy Savings with Various Minimum Phase Sizes

Table 4.2: Intratask DVFS Energy Savings

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

4.2.2 Disabling Frequency Restoration

When instrumenting a phase with DVFS, there is a choice of whether or not to restore the CPU frequency to a default value (the maximum) when exiting a known phase. The argument to restore CPU frequency is a conservative one. If a region of code is not characterized by the analysis, the highest frequency is preferred to avoid a performance penalty without known energy savings. Disabling frequency restoration upon exiting a phase is optimistic about the coverage of the 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 initial experiments with phase granularity were run with frequency restoration disabled. These are the solid bars in Figure 4.3. The hashed bars labeled "Best+Exit" in Figure
4.3 show the energy savings achieved when frequency is restored to the maximum frequency at phase exit points. The frequency configuration used for these runs is the best configuration taken from the various phase size experiments. 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 optimization. The number of frequency transitions can be reduced by at least half with frequency restoration disabled and possibly more if it enables additional merging of consecutive phases.

4.2.3 Comparison With Application Unaware DVFS

A simple approach to DVFS might scale frequency for a whole application according to some acceptable performance loss. If a 50% performance loss was acceptable, then frequency might be cut by 50% with the expectation that energy or power will be reduced. However, while energy used by the CPU alone is certainly reduced, energy consumed by the whole system may increase. To compare blind frequency scaling with Green Queue’s application aware scaling experiments are run that measure the energy consumed by an application when the whole application is scaled down to a single frequency. Every available frequency is measured for three applications: CG, FT, and Sweep3D. Results are shown in figures 4.4, 4.5, and 4.6.

The results for CG and FT show that application aware DVFS is a significant improvement over blind scaling because of its identification of phases. In both of these cases, 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
Figure 4.4: Comparison of Blind Scaling with Green Queue for CG

Figure 4.5: Comparison of Blind Scaling with Green Queue for FT
is a 18% energy reduction and it suffers a 16% performance penalty.

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 has 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 4.6 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 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 significant performance.
Chapter 5

Conclusion

Energy bills have become a significant cost to high performance computing data centers. This thesis investigated Dynamic Voltage-Frequency Scaling (DVFS) as a technique for reducing total energy consumption of MPI applications for high performance computing and has shown that there is significant potential for energy and dollar savings. This thesis identified applications with up to 21% and 32% energy savings for intratask and intertask DVFS, respectively. With energy bills for HPC centers in the millions to tens of millions of dollars a year and rising, there is a clear need to reduce consumption with informed tradeoffs.

5.1 Contributions

The contributions of this work include:

A technique for 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, one at a time. This thesis proposed a method of combining power modeling and performance measurements to predict energy consumption for frequency configurations with one application run per frequency setting.

Development of a production quality tracing and analysis framework, Green Queue. Green Queue provides a robust framework for future work in analyzing application traces. Green Queue also provides SecureScaler, a fre-
frequency throttling daemon, allowing security policies to be put in place between users and the kernel on production systems.

**An evaluation of Green Queue on a full rack of Gordon.** Gordon is the newest supercomputer at the San Diego Supercomputer Center. With 1024 cores per rack, this thesis demonstrates that Green Queue’s DVFS strategies can achieve significant energy savings at scale on modern hardware.

**Experiments demonstrating the value of application aware DVFS over blind DVFS.** This thesis demonstrated that application aware DVFS can significantly outperform techniques that are not customized to the application. Green Queue shows increased energy savings coupled with a decreased performance loss.

### 5.2 Future Work

Future work will refine the idea of optimal phase size. This thesis has shown that the optimal phase size differs between application. This is because there is a tradeoff between investing time and energy to make a frequency switch and saving energy by running a phase at its optimal frequency setting. Increased energy savings can be achieved by analyzing this tradeoff for each phase, taking into account the frequency of neighboring phases. If two neighboring phases have slightly different 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 into larger phases.

A third avenue for future work is to combine intertask and intratask DVFS
strategies. This work will need to address the complexities of modeling per-core contributions to power and making DVFS decisions when all ranks on a socket are not synchronized or have completely different workloads.
Bibliography


[27] Top500. K computer, sparc64 viiifx 2.0 ghz, tofu interconnect, November 2011.

