Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling

Michael A. Laurenzano[1], Mitesh Meswani[1], Laura Carrington[1], Allan Snavely[1], Mustafa M. Tikir[2,⋆], and Stephen Poole[3]

[1] San Diego Supercomputer Center, La Jolla, CA, United States of America
{michaell,mitesh,lcarring,allans}@sdsc.edu
[2] Google, Inc, Mountain View, CA, United States of America
mustafa.m.tikir@gmail.com
[3] Oak Ridge National Laboratory, Oak Ridge, TN, United States of America
spoole@ornl.gov

Abstract. Over the life of a modern supercomputer, the energy cost of running the system can exceed the cost of the original hardware purchase. This has driven the community to attempt to understand and minimize energy costs wherever possible. Towards these ends, we present an automated, fine-grained approach to selecting per-loop processor clock frequencies. The clock frequency selection criteria is established through a combination of lightweight static analysis and runtime tracing that automatically acquires *application signatures* - characterizations of the patterns of execution of each loop in an application. This application characterization is matched with one of a series of benchmark loops, which have been run on the target system and probe it in various ways. These benchmarks form a covering set, a *machine characterization* of the expected power consumption and performance traits of the machine over the space of execution patterns and clock frequencies. The frequency that confers the optimal behavior in terms of power-delay product for the benchmark that most closely resembles each application loop is the one chosen for that loop. The set of tools that implement this scheme is fully automated, built on top of freely available open source software, and uses an inexpensive power measurement apparatus. We use these tools to show a measured, system-wide energy savings of up to 7.6% on an 8-core Intel Xeon E5530 and 10.6% on a 32-core AMD Opteron 8380 (a Sun X4600 Node) across a range of workloads.

Keywords: High Performance Computing, Dynamic Voltage Frequency Scaling, Benchmarking, Memory Latency, Energy Optimization.

1 Introduction

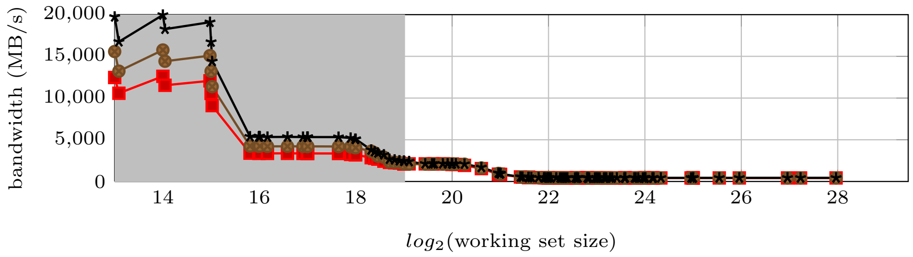
Energy costs have become a significant portion of the costs involved in the operational lifetime of largescale systems. These costs have impacts that manifest

⋆ This work was done as an active employee of the San Diego Supercomputer Center.

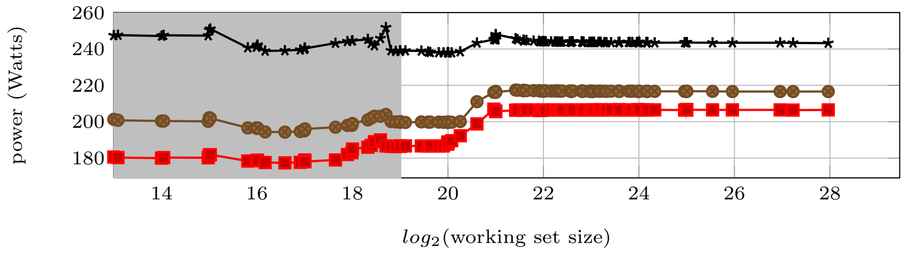
themselves in economic, social and environmental terms. It is therefore prudent to understand and minimize these costs where possible. With that goal in mind, in this work we introduce a methodology that facilitates dynamic voltage frequency scaling (DVFS) based on the expected impact that operating at some frequency will have on HPC application performance and power consumption. This methodology is then leveraged in order to choose fine-grained clock frequency settings, potentially a different frequency for each loop, for the application that minimizes system-wide energy use. Along with this methodology, we present a set of open source tools that automates the entire process.

Certain classes of scientific problems and subproblems exhibit memory bound behavior, i.e., the time to solution for the problem is decided primarily by the proximity, size and speed of available memory. Historically, the amount of memory available to computer hardware has increased at an exponential rate[1]. Nevertheless, many applications can, and will continue to, use all of the memory available to them. This means that it is important to consider the impact of physically distant data on performance and power consumption. To facilitate processor frequency scaling as a means to reducing power consumption, many modern processors have been designed to operate at a different clock frequency than certain parts of the memory subsystem[2]. This observation, along with the notion that some applications spend much of their time waiting on data that is physically distant, implies that the execution of such applications may suffer only small or acceptable performance losses when operating at lower clock frequencies, which in turn yields lower power consumption rates. Clock frequency management policies that are in use today, however, generally do not take full advantage of this opportunity. They tend to rely on very broad and coarse-grained measures to determine when it is prudent to lower clock frequency based on perceived inactivity[3][4][5]. Our methodology seeks a more refined clock management policy that can exploit the opportunity to down-clock the processor in cases where overall system activity is high, but where the processor is stalled on high-latency memory events.

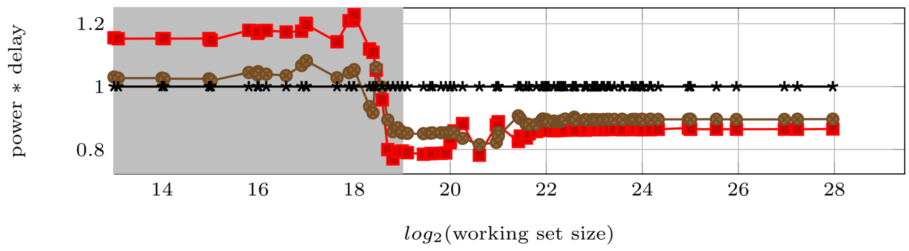
The opportunity to decrease power consumption by down-clocking the processor as it waits for physically distant data is demonstrated in Figure 1, which shows the performance (Figure 1(a)), power consumption (Figure 1(b)), and power-delay product (Figure 1(c)) for a series of Stream-derived[6] stride 8 memory load tests being run at different working set sizes and clock frequencies on an 8-core Intel Xeon E5530. The results in Figure 1(a), which shows the measured memory bandwidth for this series of tests, suggest that performance is independent of processor clock frequency when the working set size is larger than 512KB. This size coincides with the size of the L2 cache, or equivalently, when the working set size is large enough that the data resides in a memory level farther than L2 cache. Figure 1(b) shows the average power consumption levels during these same memory load tests. It is important to note that power consumption is dependent on clock frequency even for working set sizes where performance is not. Taken together, we can view the results of Figure 1 as an opportunity to reduce power consumption while minimally impacting performance



(a) Performance, expressed as memory bandwidth (MB/s).



(b) Power consumption (Watts) of the entire system (y-axis minimum is 170 Watts).



(c) Power-delay product compared against the maximum frequency, 2.4GHz.

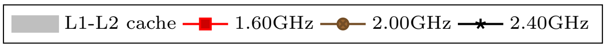


Fig. 1. Performance, power, and power-delay product of a series of Stream-derived stride 8 load tests for several clock frequencies on an Intel Xeon E5530

for certain working set sizes. The power-delay product for the Stream tests is given in Figure 1(c), which shows that power-delay product can be significantly reduced for certain working set sizes by lowering the processor clock frequency. Power-delay product is a metric that combines power usage and performance, and is simply the product of delay and normalized power usage of an application run with some clock frequency management policy when compared to the baseline clock frequency management policy. Note that power-delay product is equivalent to energy usage normalized to the baseline clock frequency mode, so these terms can be used interchangeably.

Though useful as a proof of concept, it is rarely the case that application behavior is as simple as the tests shown in Figure 1. Unlike with the simple Stream benchmark, the processing unit usually has some amount of computation that can be performed while it is stalled on memory accesses, leading to varying degrees of performance degradation when the processor is down-clocked. As such, it is necessary to understand the complex effects that memory, computational behavior and clock frequency have on performance, power and their interesting combinations (such as energy). Our approach to advancing this understanding uses a benchmark to cover the space of some possible behavioral parameters (memory size, memory access pattern, computation amount, computation type, ILP, clock frequency) to measure the effect that these factors have on performance, power and energy. For applications, we can then measure the parameters over which we have limited control (memory and computation related parameters) in order to make informed decisions about the parameter we can control (clock frequency) in order to control the power, performance, and energy characteristics of the application. When applied to selecting for energy-optimal clock frequency, our experiments show that this approach yielded measured, system-wide energy reductions of up to 10.6%.

2 Methodology

To measure the power consumption of a system we employ a WattsUp? power meter[7] to act as an intermediary between the power source and the system power supply. In order to automate the insertion of power measurement interface and clock frequency management calls, we implemented a binary instrumentation tool and library based on the PEBIL instrumentation toolkit[8]. The clock frequency management mechanism is built on top of the `cpufreq-utils` package[1][3] that is available with many Linux distributions. This instrumentation tool and library provide a powerful and low-overhead way of automatically providing a frequency management strategy to the application without requiring any build-time steps or system support. The power measurement apparatus, at the time of this writing, costs less than $150. Since a data center the size of SDSC[9] has a 2 million dollar annual electricity bill, using this kind of tool within a large data center could save a lot of money without a lot of effort.

2.1 Benchmarking for Power and Performance

In an effort to better understand how a system behaves in the presence of certain types of computational and memory demands, we have developed a benchmarking framework called `pcubed` (**P**MaC's **P**erformance and **P**ower benchmark) that allows us to generate a series of loops while retaining control over the working set size and memory address stream behavior, floating point (FP) operation counts, and data dependence features of each. The first two parameters relate to the

[1] The `cpufreq-utils` frequency switching mechanism currently requires superuser privileges, but we plan to implement a userspace tool that supports our methodology.

behavior of the memory subsystem, while the latter two are related to how effectively the processor can hide memory access latency by performing other useful operations. `pcubed` generates a series of loops, each composed of a series of strided memory references and double-precision FP operations performed on an array. The individual test permutations vary on working set size (*arrsize*), stride length (*stride*), number of memory operations (*memops*), number of FP operations (*fltops*), and number of independent FP operation sequences (*parops*).

Running a set of tests encompassing wide ranges and combinations of these parameters at all available clock frequencies for a target system yields a set of results that describes how that system behaves with respect to performance and power consumption in the presence of a wide range of demands for its computational and memory resources at every clock frequency. The results can then be used as the foundation for forming hypotheses about how an application with a certain set of features in common with the benchmark instances will operate in terms of both performance and power usage at every clock frequency on the target system.

2.2 Application Characterization

In order to determine how an application's characteristics relate to the sets of `pcubed` test characteristics, it is necessary to recognize those features in the application. These collected features are a set of observable characteristics that are related to the input parameters that can be supplied to `pcubed`. These observables are the level 1, 2 and 3 cache hit rates (derived from *arrsize* and *stride*), the ratio of the number of FP operations to the number of memory operations (derived from *fltops* and *memops*) and the lookahead distances for FP and integer computation respectively (derived from *parops* and the loop structure derived from static analysis on the binary), expressed as the average lookahead distance divided by the number of instructions in the loop.

Feature characterization is done at the loop level since loops are the vehicle through which most computation is performed in HPC applications, though it would also be possible to do this at the function level. Every loop and inner loop is examined in order to quantify certain features about its memory behavior, FP intensity and data dependency information. This step consists of a static analysis pass and a runtime trace of memory and control flow behavior that is performed by a binary instrumentation tool implemented with the PEBIL toolkit.

In order to make determinations about the expected behavior of an application loop, we first map it to one of the `pcubed` test loops that is collected as part of the system characterization. For this, we use the geometrically nearest loop in the 6-dimensional space whose members are the set of observable characteristics derived from the `pcubed` input parameters: level 1, 2 and 3 cache hit rates, the ratio of FP operations to memory operations and the average lookahead distance for FP and integer computation. As we will show later, the use of geometric distance between loop feature sets as a basis for comparison seems to work well in practice, but understanding whether geometric distance is the best measure of similarity for two loops is an open research problem.

3 Experimental Results

The technique we have proposed in this work can be used to evaluate an application and its matching pcubed loops on any metric involving performance and power. Here we evaluate only power-delay product (energy = E), though metrics such as energy-delay product ($E * D$) or metrics that further emphasize performance such as $E * D^2$ could be evaluated. In order to develop a DVFS strategy for an application whose purpose is to minimize energy usage, we use the set of results gathered from the pcubed loop that is geometrically closest to each of the application's loops. We choose the clock frequency for each pcubed loop which minimized energy to be the frequency at which we will run the matching application loop. We used two systems and a series of benchmark applications to evaluate this frequency selection scheme.

The first of these systems is an Intel Xeon E5530[10] workstation. The E5530 has 2 quad-core processors. Each core has its own 32KB L1 cache and 256KB L2 cache. Each of the quad-core processors has a shared 8MB L3 cache (for a total of 16MB of L3 for the 8 cores). Each of the 8 cores can be independently clocked at 1.60GHz, 1.73GHz, 1.86GHz, 2.00GHz, 2.13GHz, 2.26GHz, 2.39GHz or 2.40GHz. The second system is a Sun X4600[11] node that is a part of the Triton Resource[12] at the San Diego Supercomputer Center. This Sun X4600 node contains 8 quad-core AMD Opteron 8380[13] processors. Each core has its own 64KB L1 cache and 512KB L2 cache, and each processor shares 6MB of L3 cache (for a total of 48MB of L3 for the 32 cores). Each of the 32 cores can be independently clocked at 800MHz, 1.30GHz, 1.80GHz or 2.5GHz.

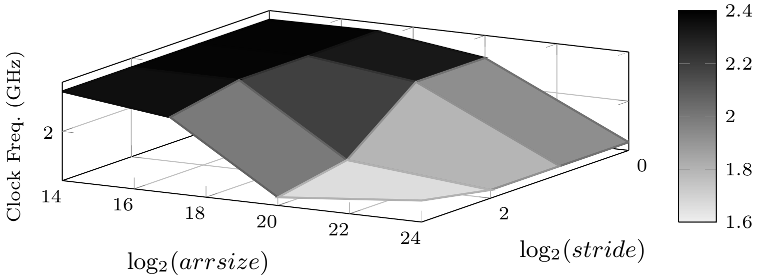
Both of these systems were probed for every available clock frequency by running pcubed on a set of 2320 benchmark instances[2] covering a wide range of loop characteristics for every clock frequency exposed by each system, which is 8 frequencies for the Intel Xeon E5530 and 4 frequencies for the AMD Opteron 8380. The runtime of pcubed is roughly 6 hours per clock frequency, though this depends on the actual set of tests being run and the clock frequencies involved.

3.1 Drawing Conclusions about System Behavior

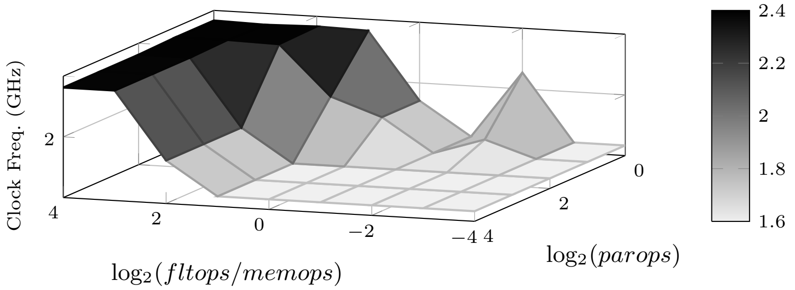
Running pcubed on a target system can allow us to draw some conclusions about that system. For instance, if it were found that the energy-optimal frequency for a large number of tests was at the lower end of the available frequencies, it would be possible to argue that lowering the range of available clock frequencies could result in a more energy-efficient system. As a slight variation on this, if a system were being planned that had a very narrow workload which demonstrated similarities to a class of pcubed loops that was most energy efficient at lower frequencies, this would suggest that a similar, cheaper architecture that offered a lower maximum frequency would be sufficient for that workload. Similarly

[2] The number of combinations and the exact parameters required depends on the features of the underlying architecture such as number of cache levels, cache line sizes, number of computational units, and number of registers.

if most tests were found to be energy-optimal at the higher clock frequencies, this could indicate that raising the range of available frequencies might have an impact on a system's energy efficiency. Neither of these phenomena were found for either the Intel Xeon E5530 or the AMD Opteron 8380, but it remains to be seen whether such systems exist. By examining the `pcubed` results alone, we can also get an understanding of what feature thresholds delineate energy-optimal frequency domains for the target system. For example, Figures 2(a) and 2(b) show maps of which clock frequency is the most energy efficient for `pcubed` tests as a function of memory behavior and computational behavior respectively. The data in these maps meets our expectations in that the energy-optimal clock frequency generally declines as the amount of time spent stalled on memory increases or as the availability of computation to the processor decreases.



(a) Energy-optimal clock frequency as a function of memory behavior. These tests have fixed *memops* = 1, *fltops* = 2 and *parops*=1.



(b) Energy-optimal clock frequency as a function of the availability of computation. These tests have fixed *arrsize* = 16MB and *stride* = 1.

Fig. 2. `pcubed`-measured energy-optimal clock frequencies on an Intel Xeon E5530

3.2 Energy-Optimal Clock Frequency Selection

For each application benchmark, we make an instrumentation pass over the executable, run the instrumented executable, then run a post-processing step on the results in order to extract the features described in Section 2.2. The overhead of the runtime application analysis depends on the application and its behavior, but for all of the benchmarks studied the maximum overhead was a

13x slowdown (average of 4.0x slowdown) on application runtime, but this step only needs to be run once per application. This post-processing step combines the static and dynamic application analysis, locates the geometrically closely `pcubed` benchmark loop, uses the results from that `pcubed` loop to make a determination about which clock frequency will result in energy-minimal execution for the application's loop, then makes a second instrumentation pass on the executable in order to embed the DVFS strategy into the application.

The set of applications used for the Intel Xeon E5530 is the NAS Parallel benchmarks[14], compiled with both the pgi and gnu compiler, as well as GUPS[15], SSCA#2[16], S3D[17] and HYCOM[18] compiled with the pgi compiler. BT and SP of the NAS Parallel Benchmarks were run on all 4 cores of a single socket. All other benchmarks were run on all of the 8 available cores. The power-delay product (or energy) for each of these benchmarks run with our DVFS scheme is computed against a benchmark run without our scheme, which is to say that it is computed against the default or peak clock rate of the system. Figure 3 shows these power-delay products. The average amount of energy saved for this set of benchmarks is 2.6%, but is as high as 7.6% for CG compiled with the gnu compiler. The set of applications used on the AMD Opteron 8380 is the NAS Parallel benchmarks, GUPS, SSCA#2 and HYCOM, all compiled with the pgi compiler. BT and SP of the NAS Parallel Benchmarks were run on 16 cores on 4 of the 8 sockets available. All other benchmarks were run on all of the 32 available cores. Shown in Figure 3, the average energy saved on the Opteron is 2.1% with a maximum savings of 10.6% on CG.

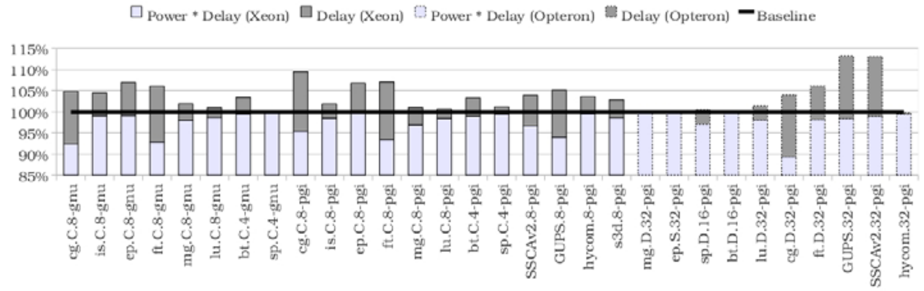


Fig. 3. Application energy usage (Power∗Delay) and slowdown (Delay) when run with a DVFS management scheme, normalized to the default frequency management scheme

In addition to power-delay product, Figure 3 also shows the raw delay to give an account of the slowdown incurred by the tests shown there. The delay is nontrivial and averages 3.8% for both the Intel Xeon E5530 and the AMD Opteron 8380. This highlights the idea that if performance is of enough importance, it is unwise to optimize purely in terms of energy. Rather, in that case it would be prudent to use a higher order function such as energy-delay product that further emphasizes performance. With modifications to the few (less than 10) lines of source code that currently perform the evaluation of the `pcubed` tests based on energy, one could easily perform evaluations based on energy-delay or any other function of performance and power.

3.3 Technique Validation

It would be time consuming to run every loop of an application at every clock frequency to determine which of those clock frequencies resulted in energy-optimal execution. A simple approach that used this strategy would require a number of runs that is on the order of the product of the number of loops and the number of available clock frequencies. Alternatively, our approach uses a set of benchmark runs (that only have to be run once in the lifetime of a system) in addition to a single instrumented application run in order to gather a heuristic to this effect. But how good is this heuristic? In order to begin to answer this we exhaustively verified that the selected clock frequency on the Intel Xeon E5530 were energy-optimal for a pair of benchmarks codes that have the property that their runtime is dominated by a single loop and therefore an exhaustive search on this loop requires relatively little effort.

For CG, the loop that is responsible for 95% of the dynamic instruction count was found to be geometrically closest, using the metrics described in Section 2.2, to the `pcubed` loop that has $arrsize$ = 1MB, $stride$ = 1, $fltops$ = 8, $memops$ = 16 and $parops$ = 1 (meaning that every FP operation is dependent on the result of its predecessor), which was found by our technique to be energy-optimal when run at 2.13GHz. By subjecting the dominant loop in CG to every available clock frequency and measuring the energy required to complete each run we found that 2.13GHz is the *actual* energy-optimal operating frequency for this loop, confirming that our selection is correct. A similar methodology was applied to the dominant loop in GUPS, which was found by our technique to be energy optimal at 1.60GHz. The `pcubed` instance found to be geometrically closest to the dominant GUPS loop has $arrsize$ = 16MB, $stride$ = 8, $fltops$ = 4, $memops$ = 64 and $parops$ = 4 (meaning that the FP operations carry only an inter-iteration dependence onto themselves). This loop was also found to run energy-optimally at 1.60GHz. This does not serve as conclusive proof that the frequencies selected by our methodology are energy-optimal in all cases nor does it indicate that every interesting aspect of program behavior is encapsulated by the space of possible loops that can be generated by `pcubed`. It does, however, serve to provide some validation for a scenario where exhaustively validating the frequency quality selection would be extremely time consuming.

4 Related Work

Dynamic voltage frequency scaling is a well known technique that has been used to reduce power and energy usage in the context of various application domains [19][20][21][22][23]. The DVFS research in HPC tends to follow one of two approaches. The first approach is to identify and exploit MPI inter-task load imbalance. The work done in [24] and [25] focuses on locating these imbalances and applying reduced frequencies to computation regions that are not on the application's critical path. By reducing the energy used on a non-critical path, overall energy can be reduced since power consumption is decreased with negligible performance loss. Since these approaches seek to exploit inter-task imbalance for energy gains instead of intra-task imbalance, they are complementary to ours.

The second approach, which our work falls into, seeks to find a way to exploit performance-clock independence that occurs within a task as a result of memory access stalls. Ge et. al. show in [26] that it is possible to reduce energy or energy-delay by running some memory bound applications either at a fixed frequency for the entire run or by hand-selecting the dynamic frequency settings for the application. Our technique goes further and demonstrates how to automatically select and use a set of such frequency settings.

In [27], the application is run to collect profiling information, then is divided by hand into phases that consist of regions that are either of similar memory behavior or are split by MPI calls. The application is then augmented to give it the capability to perform frequency scaling at its phase boundaries, and then sets of phase/frequency combinations are run in order to determine how particular frequency selections affect power and performance. This work differs from ours in two major ways. First, their methodology differs from ours in terms of the how the application is broken down for analysis. Our methodology currently looks at loop boundaries as the only possible scaling locations; theirs incorporates other possible frequency scaling points. The other major difference between their methodology and ours is that they *search* for the best frequency for the phases in the application by running it with different frequency scaling strategies, while our approach attempts to probe for the capabilities of the system then determines the frequency for the application's loops analytically.

5 Future Work

Going forward, we plan to develop the frequency selection tools as a user-level package so that it does not require root privileges. We also intend to further develop the tools and ideas in order to determine whether they extend to more architectures and compilers to determine the applicability of our methodology to other architectures. Instead of limiting the application analysis to loops only, we also would like to expand the scope of the analysis to include functions.

The extent to which `pcubed` and the parameters used to motivate its development (cache hit/miss behavior, amount of computational work available, and ILP) cover a sufficiently large portion of HPC workloads is unclear. It is likely that more thorough treatment of certain aspects of program behavior is needed. Cache coherence behavior, memory access pattern type, type and width of FP operations, and high latency off-chip events such as I/O or MPI Communications events are obvious candidates for further exploration along these lines. Another way of approaching the same problem would be to leverage some of the existing body of work relating hardware counter-supplied information to processor clock frequency selection. This will facilitate a better understanding of how application-level performance relevant features such as memory access pattern translate to the underlying conditions, observed from hardware counters, that have an effect on clock frequency selection. Doing this effectively should minimize the set of benchmarks needed to form a covering set of the behaviors that can be expected for HPC applications, which may widen the applicability of the techniques proposed in this work. It could change or narrow the scope of the information that must be gathered from the application in order to perform a mapping of application regions to benchmark instances.

6 Conclusions

This work has shown a benchmark-based approach to selecting processor clock frequency in a way that takes advantage of unnecessarily high clock rates that are maintained during memory bound computations. This methodology is implemented on top of open source software and uses a system-specific performance and power characterization that is automatically derived from the results of a set of benchmark loops, generated by the `pcubed` benchmarking framework, that are run at each clock frequency on the system. A set of tools that capture static and runtime information for an application executable is then used to analyze the application's loops in order to find the benchmark loops whose features match them most closely. From this matching, we are able to select a DVFS strategy for the application that is expected to result in minimizing energy usage during execution. `pcubed` was run, and DVFS strategies were employed on a series of benchmarks on both an Intel Xeon E5530 and an AMD Opteron 8380, where we realized energy savings of up to 7.6% and 10.6% respectively.

Acknowledgements. This work was funded in part by the Department of Defense and used elements at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory and funded by the Department of Defense. This work also used resources from Dash and from the Triton Resource at the San Diego Supercomputer Center. Special thanks to Phil Papadopoulos, Jim Hayes and Jeffrey Filliez at SDSC for their help related to gathering measurements on the Triton Resource.

References

1. Moore, G.E.: Cramming More Components onto Integrated Circuits. Electronics Magazine (1965)
2. Pallipadi, V.: Enhanced Intel SpeedStep Technology and Demand-Based Switching on Linux, `http://software.intel.com/en-us/articles/enhanced-intel-speedstepr-technology-and-demand-based-switching-on-linux/`
3. CPU Frequency Scaling, `https://wiki.archlinux.org/index.php/Cpufrequtils`
4. CPUSpeed, `http://www.carlthompson.net/Software/CPUSpeed`
5. AMD PowerNow! Technology, `http://support.amd.com/us/Embedded_TechDocs/24404a.pdf`

6. McCalpin, J.D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical Report, University of Virginia (2000)
7. WattsUp? Meters, `https://www.wattsupmeters.com/secure/products.php?pn=0`
8. Laurenzano, M.A., et al.: PEBIL: Efficient Static Binary Instrumentation for Linux. In: International Symposium on Performance Analysis of Systems and Software (2010)
9. SDSC San Diego Supercomputer Center, `http://www.sdsc.edu/`
10. Intel Xeon Processor E5530, `http://ark.intel.com/Product.aspx?id=37103&processor=E5530&spec-codes=SLBF7`
11. Sun Fire X4600 M2 Server Architecture, `http://www.sun.com/servers/x64/x4600/arch-wp.pdf`
12. Triton Resource, `http://tritonresource.sdsc.edu/pdaf.php`
13. Keltcher, C.N., et al.: The AMD Opteron Processor for Multiprocessor Servers. In: International Symposium on Microarchitecture (2003)
14. Bailey, D.H., et al.: The NAS Parallel Benchmarks. International Journal of High Performance Computing Applications (1991)
15. Luszczek, P., et al.: Introduction to the HPC Challenge Benchmark Suite. In: International Conference on Supercomputing (2005)
16. Bader, D.A., et al.: Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems. Cyberinfrastructure Technology Watch (2006)
17. Hawkes, E.R., et al.: Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights Towards Predictive Models. Journal of Physics: Conference Series (2005)
18. Halliwell, G.R.: Evaluation of Vertical Coordinate and Vertical Mixing Algorithms in the HYbrid-Coordinate Ocean Model (HYCOM). Ocean Modelling (2004)
19. Poellabauer, C., et al. Feedback-based Dynamic Voltage and Frequency Scaling for Memory-bound Real-time Applications. In: Real Time and Embedded Technology and Applications Symposium (2005)
20. Choi, K., et al.: Dynamic Voltage and Frequency Scaling Based on Workload Decomposition. In: International Symposium on Low Power Electronics and Design (2004)
21. Rajamani, K., et al.: Application-aware Power Management. In: Symposium on Workload Characterization (2007)
22. Isci, C., et al.: An Analysis of Efficient Multi-core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In: International Symposium on Microarchitecture (2006)
23. Lorch, J.R., Smith, A.J.: Operating System Modifications for Task-Based Speed and Voltage. In: International Conference on Mobile Systems, Applications and Services (2003)
24. Freeh, V.W., et al.: Just-in-time Dynamic Voltage Scaling: Exploiting Inter-node Slack to Save Energy in MPI Programs. Journal of Parallel and Distributed Computing (2008)
25. Kimura, H., et al.: Emprical Study on Reducing Energy of Parallel Programs using Slack Reclamation by DVFS in a Power-scalable High Performance Cluster. In: International Conference on Cluster Computing (2007)
26. Ge, R., et al.: Improvement of Power-performance Efficiency for High-end Computing. In: Parallel and Distributed Processing Symposium (2005)
27. Freeh, V.W., Lowenthal, D.K.: Using Multiple Energy Gears in MPI programs on a Power-scalable Cluster. In: Symposium on Principles and Practice of Parallel Programming (2005)