Efficient Speed (ES): Adaptive DVFS and Clock Modulation for Energy Efficiency

Pietro Cicotti*† Ananta Tiwari*† Laura Carrington*†

*Performance Modeling and Characterization (PMaC) Laboratory,
San Diego Supercomputer Center

†EP Analytics, Inc.
{*pietro.cicotti, ananta.tiwari, laura.carrington*}*@epanalytics.com*

***Abstract*—Meeting the 20MW power envelope sought for exascale is one of the greatest challenges in designing those class of systems. Addressing this challenge requires over-provisioned and dynamically reconfigurable system with fine-grained control on power and speed of the individual cores. In this paper, we present EfficientSpeed (ES), a library that improves energy efficiency in scientific computing by carefully selecting the speed of the processor. The run-time component of ES adjusts the speed of the processor (via DVFS and clock modulation) dynamically while preserving the desired level of the performance. These adjustments are based on online performance and energy measurements, user-selected policies that dictate the aggressiveness of adjustments, and user-defined performance requirements. Our results quantify the best energy savings that can be achieved by controlling the speed of the processor, with today's technology, at the cost of negligible performance degradation. We then demonstrate that ES is effective in automatically calibrating the speed of execution in real applications, saving energy and meeting the desired performance goal. We evaluate ES on GAMESS, an ab initio quantum chemistry package. We show that ES respects the stipulated 5% performance loss bound and achieves 16% decrease in energy required to complete the execution while running with a power draw that is 18% lower.**

I. Introduction

A simple extrapolation based on today's Joules per FLOP ratio suggests that a 20 fold increase in energy efficiency is required to meet the 20MW power budget seeked for exascale [4]. In addition to the improvements in the design of hardware, software-based techniques to dynamically reconfigure hardware to *exactly* match (within the limits of the hardware) a given workload's demands are needed to bridge this massive gap in energy efficiency.

Computing systems are often inefficient because they are designed as a one-size fits all, and must perform on a wide variety of workloads. A more energy efficient mode of operation is to identify system subcomponents that are not heavily exercised and put them on lower power or speed states. While promising, this solution is challenging to implement because a given scientific application can have large number of computational phases that put vastly different levels of stress on different subcomponents. Care must also be taken with respect to the negative repercussions of changing hardware parameters – e.g., changing CPU clock frequency can save power but that change can also impact performance negatively when applied blindly. Decisions to reconfigure hardware, therefore, has to be dynamic and has to carefully consider the tradeoffs between multiple competing goals.

In this paper, we present EfficientSpeed (ES), a library that exposes a series of customizable energy optimization policies to improve energy efficiency in scientific computing while preserving most of the performance. The library-based approach allows for HPC programmers to directly indicate (via source code modifications) which computational phases or loops within their application will likely be good candidates for energy efficiency optimizations. The same library-based approach also allows for binary instrumentation toolkits to insert ES calls directly into the binary around phase entry and exit points, which is the method that we utilize in this paper. The programmers and application scientists have the option to express performance expectations (with respect to the base or default hardware configuration), which are strictly met by ES policies.

ES utilizes two methods to reconfigure hardware for better energy efficiency – Dynamic Voltage and Frequency Scaling (DVFS), which is a way to transition the CPU to a low power state (associated with lower frequency), and clock modulation, which is a way to control the clock duty cycle (without affecting the clock frequency). ES first establishes performance baselines for given computational phases by observing those phases' executions at base or default speed. ES then selects the best CPU speed to run various phases using various reconfiguration policies. ES takes advantage of the iterative behavior of most scientific codes by evaluating new hardware configurations for some iterations.

In this paper, we make the following contributions:

- We first demonstrate how application-driven hardware reconfiguration can save energy by performing a comprehensive study to show the full potential of this strategy. This study establishes an upper-bound on how much improvement in energy efficiency can be achieved using ES's optimization policies.
- We present ES library – library that comprises of a series customizable hardware reconfiguration recipes that improve energy efficiency while meeting the performance goals set by the users.
- We evaluate the policies exposed by ES in a large scale

chemistry code, GAMESS, and show 16% energy savings while only registering less than 5% loss in performance.

The paper is organized as follows. The next section describes related research, which is followed by the description of design and implementation of the ES library in Section III. Section IV describes our experimental test-bed and the evaluation results are presented in Section V. Section VI provides concluding remarks.

II. Related Work

Using DVFS and clock modulation as key techniques to improve energy efficiency is a well-studied topic. We divide the discussion of the related projects into two classes – intra-node and inter-node techniques. Inter-node DVFS schemes [9], [18] identify MPI inter-node load imbalance or the time spent blocked in MPI routines and use that information to lower the clock frequency of the hardware running the slacking or blocking MPI ranks, since overloaded ranks bottleneck the application progress. Sundriyal et. al propose runtime procedures that target MPI collective operations [21] and strategies that group several point-to-point communications and apply DVFS and clock modulation at per-group level aiming to reduce the overhead of repeated application of DVFS and clock modulation [22]. ES takes an intra-node approach and the approaches taken by the aforementioned projects are complementary to ours.

Intra-node technique looks for opportunities to scale down CPU frequency, and therefore power draw, for application phases which lack computational work (e.g., phases where CPU spends significant time in waiting for the data motion from memory to complete). Ge et al. [11] explore the opportunities to reduce energy consumption by running memory-bound applications either at statically selected CPU frequency for the entire execution or at subroutine-specific frequencies. ES's approach dynamically selects clock speed for various phases in a given application.

Laurenzano et al. [16] compare computational signatures (e.g., cache behavior and arithmetic intensity) of different phases in a given application against signatures collected for an extensive set of micro-benchmarks to guide the dynamic clock frequency selections. This method is prone to making incorrect decisions for computations with properties that are not specifically accounted for in the construction of micro-benchmarks. Green Queue [23] uses sophisticated binary analysis and tracing tools to demarcate computational phases in an application. Based on modeled power and measured performance, each phase is assigned a static frequency to run on. Application binary is then instrumented to adjust CPU clock frequency at phase entry and exit points. ES takes a dynamic approach to changing CPU clock frequency and can be used within Green Queue to drive the optimization process.

Freeh et al. [10] manually divide a given program into phases by analyzing program profile. Phase boundaries are based on a memory pressure metric – operations per cache miss. Each of the identified phases is then run on all of the available frequencies to determine the most energy efficient frequency for that phase. For an application with L loops, finding the optimal clock frequencies using this approach requires $O(L)$ runs of the application. Complex applications may have many phases rendering this method impractical, motivating ES's approach of optimizing all phases in a single execution.

A number of projects use power and energy models trained on hardware counter data [6], [8], [17], [20] to determine optimal frequency settings for an application. Often these approaches are time interval-based. Time interval-based approaches take observations about the application from previous intervals to estimate the time/power requirements and workload of upcoming intervals. These estimations are mostly based on hardware counters aggregated in the previous intervals — cache accesses counters [12], MIPS (Millions of Instructions per Second) [13] and CPU stall cycles [14]. Time interval-based approaches can run into suboptimal behavior when pre-defined time-intervals do not line up with changes in application behavior. ES's strategy of using user-specified or automatically discovered loops avoids this issue.

III. Design and Implementation

In this Section, we describe the design and implementation of ES. ES is a library that can be used to insert hooks around different computational phases of large-scale scientific applications and utilize highly customizable energy efficiency optimization policies to reduce the energy footprint of applications while respecting the performance requirements stipulated by the users. Energy efficiency optimization is done using hardware configuration mechanisms that control the speed of CPU (DVFS and clock modulation).

One of the key components of ES is its application programmer interface (API), which consists of a global state (GS) data structures, initialization and finalization functions, phase boundary (entering and exiting) demarcation functions, and core speed search policies. ES's internal state is not visible to the program. While initialization and finalization functions are required at the beginning and at the end of a program execution, the phase boundary functions delimit the execution phases to be controlled and optimized by ES and interleave the control flow between the ES and the application.

A program can be instrumented manually, by explicitly invoking the library functions, inserting pragma directives, or via binary instrumentation. In this paper we consider user-guided binary instrumentation as way to instrument a program with ES. Binary instrumentation is convenient because it does not require access to the source code, or recompiling. In addition, by integrating ES with a mature framework for binary instrumentation [15], application profiling and trace data analytics [5], we have simplified the process for identifying candidate loops that can be treated as larger computing phases. The description of the automatic instrumentation framework and the phase selection process are outside the scope of this paper.

The remainder of this Section is organized as follows – Section III-A describes the ES policies and parameters that control the search for the optimal speed, Section III-B describes the GS data structure, and Section III-C describes the functions of ES and the API.

A. Search Policies

ES provides its users fine-grained control over several features through the specification of environment variables. User can specify 1) performance targets (or bounds on performance loss due to the reduction in CPU speed) for each of the candidate phases, 2) optimization target (whether to optimize for power, energy or energy delay product), 3) policy to use in order to adjust the CPU speed and 4) a default or base speed for phases that are not controlled by ES.

1) Performance Target: When searching an optimal speed while meeting a given *performance target*, ES will select the lowest speed that meets that target. The speed selected, assuming that the power draw monotonically increases with speed, should give the lowest power draw within the desired performance range. In addition, if energy measurements are possible via the Running Average Power Limit (RAPL) counters [7], ES can use dynamic measurements to optimize for power, energy consumption, and energy delay product (EDP).

The performance target, expressed in Instructions Per Cycle (IPC), for each of the candidate phases can be provided by the user via a configuration file or discovered at run time. In the latter case, all phases are first executed at full CPU speed and the performance measured is used as the reference. Once the reference performance for a phase is defined, the speed for that phase is adjusted according to the selected policy. To tolerate performance variability, it is also possible to average the performance measurements collected over a user-selected number of iterations.

For performance comparisons, ES transforms the IPC metric into Instructions Per Second (IPS) metric. IPS captures performance independently of the clock speed, or the duration of the phase. IPS metric can, therefore, be used to compare performance across different speed settings and different executions of the same phase and users do not need to specify different performance targets for different instances of a phase or for different inputs.

IPS is obtained at runtime from IPC hardware counters and frequency (cycles per second). When entering a phase, ES starts instruction and cycle counters, and when leaving the phase the counters are read to compute IPC for the phase. IPC is then scaled according to the current speed and converted to IPS. As in discovering the performance target, the moving average of IPS can be used when comparing performance with the reference target.

2) Optimization Target: Online energy measurements via the RAPL counters make it possible to optimize for power energy, and energy delay product (EDP) while respecting the performance target specified by the users. To compare performance of a given phase against some specified target, energy and EDP must be transformed so that the values compared are independent of the instance of the phase. Therefore, energy is compared as energy per instruction, and EDP is compared as EDP per instruction, which is equivalent to energy divided by IPS.

The search for the optimal speed finds a local minimum in power, energy, or EDP. Ideally, the local minimum is also an absolute minimum, although in practice other events and the state of the system (e.g. temperature) affect the power draw and as a result, power, energy, and EDP are not always concave or monotonic. Nevertheless, ES adjusts the speed to descend the curve toward a minimum which, in most cases, is a close approximation of the absolute minimum (more in Section V).

3) Adjustment: ES exploits the iterative nature of scientific applications to adjust the speed. For a given phase, the measured performance from previous timestep or iteration is compared against the target performance and the CPU speed for the next timestep is adjusted accordingly towards the optimization goal, before entering the phase. This adjustment may vary depending on the selected policy. If it is determined that performance target cannot be met at lower speed settings, the phase is marked as such (or deactivated) and removed from further consideration. The granularity of adjustment for other phases can be as fine as single speed step (e.g., going from 2.6GHz to 2.5GHz CPU clock frequency on our test system), or it can be proportional to the how far the current performance of the phase in consideration is from the specified target.

4) Base Performance: When exiting a phase, there is the option of resetting the speed to a different level and ES offers three possible choices – do nothing, reset the speed to the level detected when entering the phase, or set the speed to a user-specified base level. The first option (do nothing) can help minimize the overhead by avoiding frequent speed changes, and the implicit assumption is that most of the running time is spent in executing phases that are controlled by ES. However, in some cases, a significant amount of time is spent outside of these phases and in such cases, it is better to either reset the speed to the value encountered when entering the phase, assuming a pre-existing speed setting, or to set the speed to a user-defined level that can represent an optimal value for most of the program.

B. The Global-State Data Structure

ES library maintains a global data structure where it stores the configuration and the state of the library, including phase-specific data. The configuration stored in the GS, for example, includes the selected policies for adjusting the CPU speed setting while searching the optimal speed. The configuration of such policies is established when ES is initialized and is stored in the global state data structure. In addition, the GS is used to store the current speed setting and the base speed setting. Other global settings include whether or not the library produces any output, and the verbosity of the output.

For each instrumented phase, the GS also incudes a dedicated structure. The dedicated structure is used to maintain per-phase state information, including whether the phase is active or not[1], the last performance measurements as are required to compute the moving average, the base performance, and the latest selection of the optimal speed setting. While most of the GS does not change after the initialization, phase specific state information is continually updated during execution, according to the search policy in place and the measurements.

[1] A phase can be deactivated if it is determined that it is best to run that phase at the max speed.

C. The Application Programmer Interface (API)

There are four functions to control and interact with ES– initialization and finalization functions to start and gracefully terminate the runtime system of ES, and phase boundary functions to signal the beginning and the end of a computing phase. Together, these functions control the speed of the application ensuring that the desired performance level is maintained while trying to reduce power draw and energy consumption.

The initialization function reads the configuration from environment variables and optionally from an input file, and then allocates and initializes the GS. The finalization function simply frees allocated resources and resets frequency and modulation to the maximum. If selected, the finalization function also outputs the final speed settings and statistics (i.e. steps to find the optimal speed).

When entering a computing phase, ES configures the speed of the core according to the state of the phase and the selected policy, and then starts the performance measurement. Upon exiting the computing phase, the achieved performance is recorded and fed to the policy controller that updates the state of the phase and adjusts the selection of the speed for the next occurrence of the phase. Finally, depending on the policy in place, ES may reset the speed to the base or to the speed detected when entering the phase.

IV. Experimental Setup

The testbed system is a dual-processor node with two 8-core Intel Xeon E5-2650v2 processors (Ivy Bridge). Each core has a 64KB L1 cache (32 KB Icache + 32KB Dcache), a 256KB combined L2 cache, and a 20MB shared LLC. The two processor are connected to 64GB of DDR3 DRAM. The system is configured with Hyper-Threading and Turbo Boost disabled for all the experiments. Each of the processors can be independently clocked at maximum 2.60GHz frequency and minimum 1.20GHz (at 100 MHz increments). Processor clock frequency is changed using the `cpufreq-utils` package [1] that is available with many popular Linux distributions. Clock modulation can be done at per-core level by writing to specific Model Specific Registers (MSR). For our testbed, we have 16 different modulation levels (i.e. 6.25% to 100% duty cycle, at 6.25% increments).

To measure the power draw of the system and evaluate energy efficiency we used the WattsUp? Pro power meter [2], which measures the AC power being consumed by the entire system. The device can produce power readings at one second intervals and the readings can be accessed via a USB connection.

We evaluate the effect of CPU speed variation several benchmarks. First we run the benchmarks, testing the performance variation and power draw at each speed, then we run the benchmarks under the control of ES. Finally, we evaluate ES on GAMESS, an ab initio quantum chemistry package. The tests are listed in Table I.

The set of micro-benchmarks includes the STREAM benchmarks (*copy, add, scale, triad*), a naive matrix-matrix multiplication, and the computation of the mean value of a vector.

Micro-Benchmarks	
Name	Description
Stream Copy	double precision **x=y**
Stream Add	double precision **z=x+y**
Stream Scale	double precision **z**=a**x**
Stream Triad	double precision **z**=a**x+y**
MatMul	double precision C=A×B
Mean	double precision a=avg(**x**)
NPB Benchmarks	
Name	Description
BT class B	Block Tri-diagonal Solver
CG class C	Conjugate Gradient
FT class C	3D Fast Fourier Transform
LU class C	LU Factorization
MG class C	Multi-Grid solver
SP class B	Scalar Penta-diagonal solver
Applications	
Name	Description
GAMESS	ab initio quantum chemistry

TABLE I: Benchmarks and Applications used for ES's evaluation

These micro-benchmarks range from a 2:0 ratio of memory operations to floating point operations, for *copy*, to a 1:2 ratio, for *mean* (each value read is scaled by $\frac{1}{n}$ directly).

We selected 6 of NPB benchmarks that are most relevant for fluid dynamics computations [3]: *BT, CG, FT, LU, MG,* and *SP*. *BT* solves a problem of multiple independent systems of non-diagonally dominant, block tridiagonal equations; *CG* uses a conjugate-gradient method to compute an approximation of the smallest eigenvalue of a sparse, symmetric positive definite matrix; *FT* solves partial differential equations using FFTs; *LU* solves a sparse system using Symmetric Successive Over Relaxation (SSOR); and the *MG* benchmark is a simplified multigrid calculation.

Finally we applied ES to GAMESS [19], an *ab initio* quantum chemistry package with a large set of capabilities for obtaining wave functions, applying correlation treatments, and computing derivatives. For our evaluation, we used an input that performs the MP2 energy and gradient calculations of substituted silatrane, the 1-trichloromethylsilatrane (TCMS) molecule (more details about the input can be found in [22]).

V. Results

This section presents our evaluation results. In Section V-A, we test and quantify the effect of varying the processor speed. Then, in Section V-B we show the results of running GAMESS under the control of ES.

A. DVFS and Clock Modulation

We first measure the impact of varying the processor speed on energy and performance. We establish a baseline that can be used to evaluate the accuracy of ES in selecting the optimal speed configuration. In these experiments the speed is set at the start of execution of the application, and the application runs without any ES instrumentation. This process is repeated for each speed setting, the results of which are shown in Figure 1.

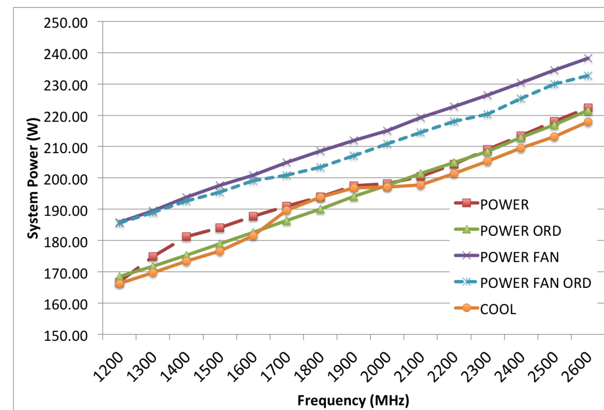
Figure 1 shows the average power draw of all the benchmarks, over a range of speed settings. Each series represents a different set of experiments – 1) running all the benchmarks in some pre-established order before increasing processor speed (labeled POWER in the figure), 2) running each benchmark for every speed (POWER ORD), and 3) running benchmarks with a 5 minutes delay between runs (this configuration, labeled COOL, is used for the experiments in the remainder of the paper). In addition, the first two sets are repeated by holding the system fans at maximum speed as opposed to letting the system control the fan speeds (POWER FAN and POWER FAN ORD respectively). From these results, we demonstrate variability in power due to factors other than the processor speed – temperature and other components of the system (e.g. fans) affect the overall power draw in unpredictable ways. For example, the fans can cause increase of the power draw by almost 20W. The order of execution, which affects the temperature and overall state of the system, also affects the power draw. As a result, letting the system rest between benchmarks reduces the power draw in most cases. Finally, we observe that DVFS and clock modulation are affected differently by the state of the system. When using DVFS, the average power draw grows regularly with respect to CPU speed, and selecting the lowest speed within a performance range is good approximation in finding the lowest power draw. When using clock modulation we observed more variability in the power draw, and dynamic readings of the power draw enable a more accurate selection when regulating the speed via clock modulation.

Figure 2 shows the power draw of each benchmark, normalized to the maximum speed, for a range of DVFS (Figure 2(a)) and clock modulation configurations (Figure 2(b)). On the individual benchmark plots, it becomes apparent that the power draw increases with great similarity between benchmarks, but it also reveals that between 1.9GHz and 2.2GHz, there is in most cases a dip. Similarly, there is a dip between 62.5% and 81.25% duty cycle, which overlaps with to the speed range roughly corresponding to the 1.9GHz and 2.2GHz range. We attribute this anomaly to the system and mostly likely to stepped changes in the fans activity.

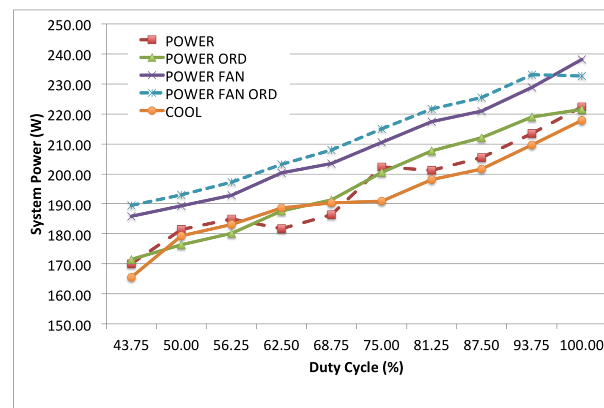
The other metric that determines the efficiency of the system is execution time. Figure 3 shows the slowdown suffered with respect to the maximum CPU speed. For reference, the figure also shows the ideal slowdown, which is linear with respect to the speed. As expected, since processors are not 100% utilized due to stalls of various nature, the performance loss observed is always less than the decrease in speed. The change in efficiency can be inferred from the slowdown. While the CPU speed configuration is changed in discrete and uniform steps, the actual performance does not change linearly and in general, the improvement in performance decreases as the speed increases. We also observe that the performance loss is monotonic, but does present significant variations on the slope, and those variations, combined with the irregularities in the power draw are reflected in energy consumption and EDP.

Figure 4 shows the energy consumption of each benchmark, normalized to the maximum CPU speed, for a range of DVFS (Figure 4(a)) and clock modulation configurations (Figure 4(b)). In almost all of the benchmarks, as the speed increases the energy consumption decreases to a minimum and then increases again. As long as the processor is not utilized efficiently (e.g. stalls on memory waits), lowering the speed (and therefore the power draw) results in lower energy consumption. When the processor is used efficiently by the code, lowering the speed does not improve the relative performance (e.g. IPC) and increases the running time (hence energy consumption); in addition, even if the processor power draw is reduced proportionally to the increase in running time, there is a constant component of the overall system power draw (e.g., idle or static power) that does not decrease. The combined effect of the irregularities observed in the power draw and slowdown is that in some cases there are multiple local minimums within the same benchmark; however, there is either little or no difference between minimums, or they are separated by a significant performance difference. Therefore, the search for an energy-optimal configuration at a given performance should always succeed.

Finally, to take into account both energy and performance, we consider EDP as the metric of efficiency. Intuitively, if a decrease (or increase) in energy consumption corresponds to a decrease (or increase)) in performance of equal amount, then EDP does not change; but if a decrease (or increase) in



(a) DVFS



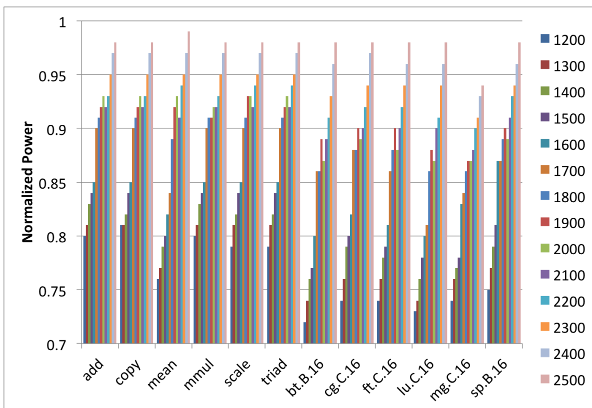
(b) Clock Modulation

Fig. 1: Average power draw of the benchmarks, over a range of CPU speed settings.

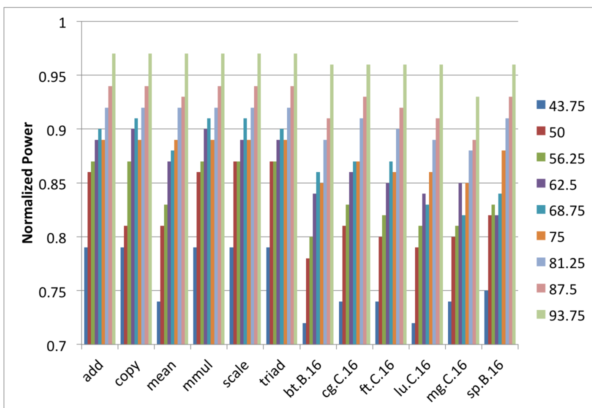
energy corresponds to a decrease (or increase) in performance of a smaller amount EDP increases indicating lower efficiency, and a decrease in EDP indicates higher efficiency. Similarly to energy, for the most part EDP exhibits a concave curve, as shown in Figure 5. In addition, since in EDP the contribution of time is squared (and therefore the performance curve which is more regular has more weight), the EDP curve is smoother and presents mostly absolute minimums.

Selecting the most energy efficient configuration has different outcomes depending on the optimization goal. We consider the following optimization goals: lowest power, energy, and EDP within a 5%, 10%, 20%, and 30% performance loss and overall (indicated as ∞ performance threshold). For each of those goals, Table II shows the average variation in performance, power draw, energy consumption, and EDP.

Tolerating a large slowdown has in practice little impact on the energy saved and EDP because in most cases, the biggest energy savings are gained at speeds close to the top speed. In fact, in most applications there is no change in the selection when tolerating larger than 5% or 10% slowdowns: because the additional speed configurations do not result in reduced energy consumption. Most of the energy reduction is achieved within a 5% performance degradation, in which case optimizing for EDP seems to be a better overall choice.

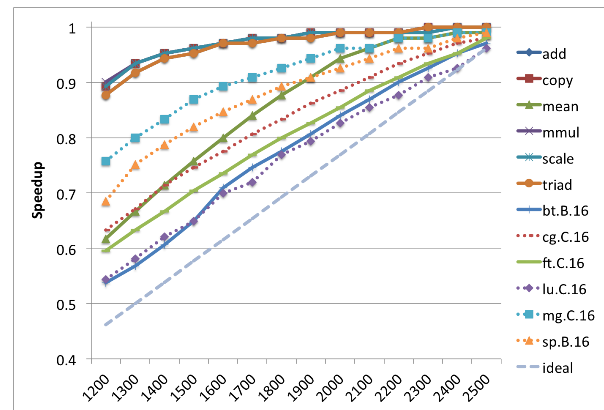


(a) DVFS

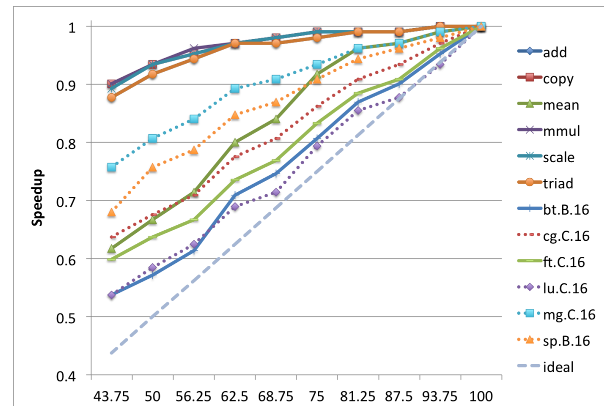


(b) Clock Modulation

Fig. 2: Power draw of each benchmark, normalized to the maximum CPU speed, for a range of DVFS and clock modulation configurations.



(a) DVFS



(b) Clock Modulation

Fig. 3: Performance loss with respect to the maximum CPU speed.

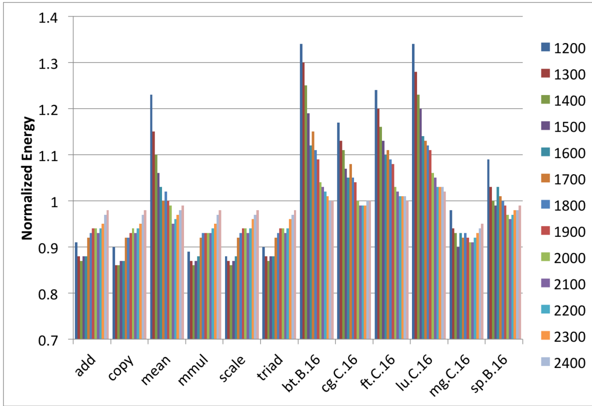
However, because of the differences between applications, the difference in energy saved by different optimization goals in some cases is larger than indicated by the average (e.g. in the mean and SP benchmarks, the lowest energy saving is 2% (out of 5%) and 1% (out of 3%) larger for optimal energy than for optimal EDP, within a 5% performance threshold). The difference in energy saving between the optimal energy and the optimal EDP configuration increases for all the benchmarks towards lower speeds. We also notice slight advantage in using DVFS over clock modulation due the finer granularity of DVFS.

B. *GAMESS*

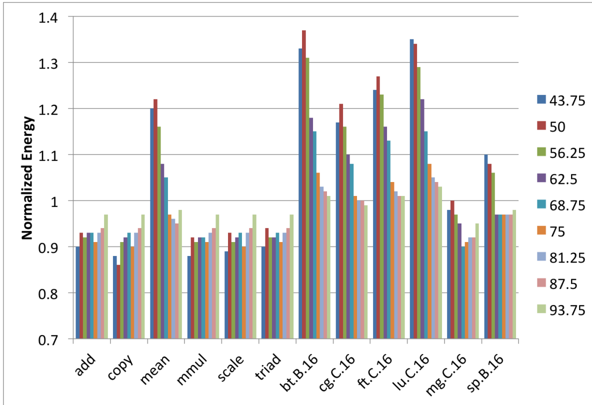
In this section we demonstrate the use of ES on GAMESS. To instrument GAMESS we used binary instrumentation to collect statistics on loops and functions, and identified four phases: the direct 4-index transformation (subroutine partran), the computation of the contributions to the 1-particle density matrices (subroutine `mkvvo`), the Z Vector solver (subroutine `zvectr`), the computation of the 2-particle gradient (subroutine `pjkdmp2`), and the DDI data server loop (subroutine `DDI_Server`). In GAMESS, for each process that computes chemistry calculations, there is an associated "data server" process that services data requests from the distributed arrays.

Opt. Metric	Performance Thresh.	Avg. Speedup DVFS/Mod	Avg. Power Variation(%) DVFS/Mod	Avg. Energy Variation(%) DVFS/Mod	Avg. EDP Variation(%) DVFS/Mod
Power	∞	0.74/0.73	-24/-24	7/7	57/56
Power	30%	0.84/0.84	-18/-18	-2/-2	18/17
Power	20%	0.86/0.87	-17/-16	-3/-4	13/11
Power	10%	0.93/0.93	-13/-11	-6/-4	1/3
Power	5%	0.96/0.97	-11/-8	-7/-5	-3/-2
Energy	∞	0.96/0.94	-11/-11	-7/-6	-3/0
Energy	30%	0.96/0.94	-11/-11	-7/-6	-3/0
Energy	20%	0.96/0.94	-11/-11	-7/-6	-3/0
Energy	10%	0.96/0.97	-10/-8	-7/-6	-4/-3
Energy	5%	0.97/0.98	-10/-7	-7/-5	-4/-4
EDP	∞	0.98/0.99	-8/-6	-6/-5	-5/-4
EDP	30%	0.98/0.99	-8/-6	-6/-5	-5/-4
EDP	20%	0.98/0.99	-8/-6	-6/-5	-5/-4
EDP	10%	0.98/0.99	-8/-6	-6/-5	-5/-4
EDP	5%	0.98/0.99	-8/-6	-6/-5	-5/-4

TABLE II

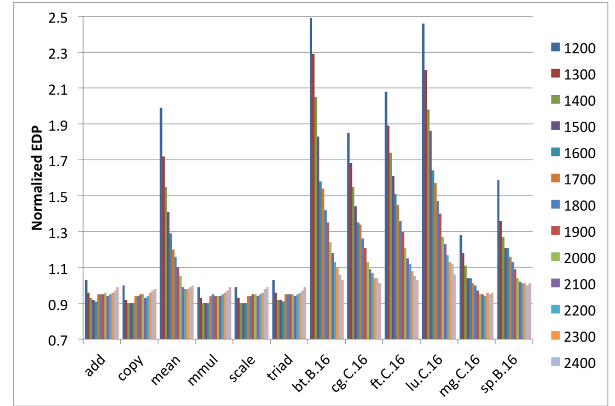


(a) DVFS

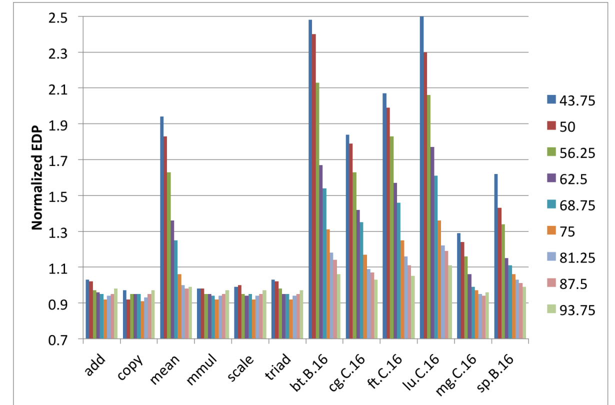


(b) Clock Modulation

Fig. 4: The energy consumption of each benchmark, normalized to the maximum speed, for a range of DVFS and clock modulation configurations.



(a) DVFS



(b) Clock Modulation

Fig. 5: EDP of each benchmark, normalized to the maximum speed, for a range of DVFS and clock modulation configurations.

A 16-core run of GAMESS will, therefore, have 8 compute processes and 8 data servers.

To establish the performance baseline, we first run GAMESS by manually setting the frequency on the cores. Table III shows the results on frequencies from 2.6GHz to 2.2GHz, below which we consider the slowdown unacceptable. In all cases, the odd cores run at the lowest frequency: data server processes of GAMESS are mapped to those cores and running at the lowest speed does not affect the overall performance. The results show that at 2.4GHz and 2.3GHz the performance penalty is within 5%, and at 2.2GHz, performance is just below 10%. If we were to tolerate a 5%

Manual - Performance Baseline Experiments				
Frequency (MHz)	Performance	Power	Energy	EDP
2600	100.00	100.00	100.00	100.00
2500	98.90	86.71	87.68	88.66
2400	95.46	85.45	89.51	93.77
2300	91.93	84.37	91.77	99.82
2200	88.37	83.11	94.04	106.42
Overhead Analysis: Settings from file				
Frequency (MHz)	Performance	Power	Energy	EDP
2600	100.00	100.00	100.00	100.00
2500	98.90	87.23	88.20	89.18
2400	96.14	85.82	89.27	92.85
2300	91.93	84.56	91.98	100.05
2200	88.43	83.67	94.62	107.00
ES Auto				
Performance thresh	Performance	Power	Energy	EDP
0.05	96.90	81.47	84.07	86.76
0.10	92.44	80.81	87.42	94.57
0.15	90.76	80.07	88.22	97.21
0.20	87.22	79.97	91.69	105.11

TABLE III: GAMESS Results

performance penalty, there is a potential power draw reduction of almost 15%, energy consumption saving of 12%, and an EDP improvement of 11%. Energy and EDP are however more sensitive to the performance variation and increase at any frequency lower than 2.4GHz.

ES also allows manual selection of CPU speed; users can provide the settings for different phases in a configuration file. We use this feature to evaluate whether ES and our approach of using binary instrumentation to insert ES API calls into the binary adds any noticeable overhead. The second set of results in Table III (labeled "Overhead Analysis: Settings from file") presents the outcome of this overhead analysis. We observe no noticeable overhead and the savings achieved are approximately the same as the performance baseline experiments that we described above.

We then allow ES to automatically search for and find the right CPU speed configuration (given different performance loss thresholds) for different phases; results are labeled as "ES Auto" in Table III. Depending on the slowdown tolerated, ES achieves higher savings than the manual solution because of its ability to adjust the speed per-phase. ES achieves a 20% power draw reduction within a 10% performance penalty, and an 18% power draw reduction within a 5% performance penalty. Similarly, ES improves on energy and EDP.

VI. Conclusion

We presented ES, a library that dynamically adjusts the processor speed to lower the power draw, the energy consumption, or the EDP of an application while preserving performance within the desired range.

On the benchmarks suite ES reduced the average power draw by 11%, the average energy consumption by 10%, and the average EDP by 8%, all within a 5% performance penalty. Tolerating higher penalties increases the savings further, though in general the return diminishes at larger penalties.

We applied ES to GAMESS, a widely used quantum chemistry package, and demonstrated how ES can dynamically adjust the speed, with little overhead, and lower the power draw obtaining a 18% reduction. More importantly, we demonstrated the effectiveness of optimizing computation phases independently, showing that ES improved the savings achieved by statically setting the speed for the whole computation.

The analysis to discover and select suitable computation phases in large production codes is difficult and labor intensive. Future work will focus on developing a framework to automate the discovery and instrumentation of application phases. Other extensions of the library will include controlling (or at least receiving feedback from) other system devices, such as memory, and to establish policies and manage the power budget assigned to an application.

Acknowledgments

This material is partly based upon work supported by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476.

References

[1] CPU Frequency Scaling. https://wiki.archlinux.org/index.php/Cpufrequtils.

[2] WattsUp? Meters. https://www.wattsupmeters.com/.

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

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. www.cse.nd.edu/Reports/2008TR-2008-13.pdf, 2008.

[5] L. Carrington, M. Laurenzano, and A. Tiwari. Characterizing large-scale hpc applications through trace extrapolation. *Parallel Processing Letters*, 23(04):1340008, 2013.

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

[7] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, Aug 2010.

[8] G. Dhiman and T. S. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proceedings of the 2007 international symposium on Low power electronics and design*, ISLPED '07, pages 207–212, New York, NY, USA, 2007. ACM.

[9] V. W. Freeh, N. Kappiah, D. K. Lowenthal, and T. K. Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *J. Parallel Distrib. Comput.*, 68(9):1175–1185, Sept. 2008.

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

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

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

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

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

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

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

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

[18] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, New York, NY, USA, 2009. ACM.

[19] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993.

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

[21] V. Sundriyal and M. Sosonkina. Per-call energy saving strategies in all-to-all communications. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 188–197. Springer Berlin Heidelberg, 2011.

[22] V. Sundriyal, M. Sosonkina, A. Gaenko, and Z. Zhang. Energy saving strategies for parallel applications with point-to-point communication phases. *Journal of Parallel and Distributed Computing*, 73(8):1157 – 1169, 2013.

[23] A. Tiwari, J. Peraza, M. Laurenzano, L. Carrington, and A. Snavely. Green queue: Customized large-scale clock frequency scaling. In *Proceedings of the Second International Conference on Cloud and Green Computing*, CGC '12, 2012.