2015 IEEE 35th International Conference on Distributed Computing Systems

Phase Detection with Hidden Markov Models for DVFS on Many-Core Processors

Joshua Dennis Booth
Sandia National Laboratories
jdbooth@sandia.gov

Jagadish Kotra
Penn State
jbk5155@cse.psu.edu

Hui Zhao
Penn State
hzz105@cse.psu.edu

Mahmut Kandemir
Penn State
kandemir@cse.psu.edu

Padma Raghavan
Penn State
raghavan@cse.psu.edu

***Abstract*—The energy concerns of many-core processors are increasing with the number of cores. We provide a new method that reduces energy consumption of an application on many-core processors by identifying unique segments to apply dynamic voltage and frequency scaling (DVFS). Our method, phase-based voltage and frequency scaling (PVFS), hinges on the identification of phases, i.e., segments of code with unique performance and power attributes, using Hidden Markov Models . In particular, we demonstrate the use of this method to target hardware components on many-core processors such as Network-on-Chip (NoC). PVFS uses these phases to construct a static power schedule that uses DVFS to reduce energy with minimal performance penalty. This general scheme can be used with a variety of performance and power metrics to match the needs of the system and application. More importantly, the flexibility in the general scheme allows for targeting of the unique hardware components of future many-core processors. We provide an in-depth analysis of PVFS applied to five threaded benchmark applications, and demonstrate the advantage of using PVFS for 4 to 32 cores in a single socket. Empirical results of PVFS show a reduction of up to 10.1% of total energy while only impacting total time by at most 2.7% across all core counts. Furthermore, PVFS outperforms standard coarse-grain time-driven DVFS, while scaling better in terms of energy savings with increasing core counts.**

I. Introduction

Emerging many-core architectures are the building blocks of future distributed computing systems. Compute resources in these many-core processors must contend with limited energy [1], [2], [3]. Many techniques exist to reduce the energy of an application, but no technique is more popular than dynamic voltage/frequency scaling (DVFS) [4]. The use of DVFS depends on correctly identifying locations in the application and hardware component, such as computational cores, to scale voltage and/or frequency. The choice of which components will become of importance as many-core processors get more complex. These complexities include heterogeneous hardware and the ability to scale voltage/frequency for individual hardware components [1], [5], [6], [7]. However, coordinating DVFS for multiple hardware components can be even more difficult [5], [7]. In this work, we provide a general scheme to identify locations and hardware components to apply DVFS using phase based models created with Hidden Markov Models .

DVFS techniques reduce the voltage and/or frequency of particular system components in order to reduce power. Currently, these components include computational cores and memory, but may include per-core and on-chip networks in the future [2], [5]. When voltage and/or frequency is decreased, that particular hardware component operates at a reduced speed; operations depending on the component will be slower. Therefore, DVFS must be applied to the correct hardware component at the correct time in an application in order to reduce energy, since energy is the integral of both time and power. In order to be successful, DVFS must be applied when the hardware component is underutilized, yet using a large amount of power. Also, applying DVFS frequently, such as many time-driven DVFS methods, can result in significant overhead [8].

In this work, we provide an alternative general framework that uses Hidden Markov Models (HMMs)to detect where in an application and which hardware components to apply DVFS. We will refer to this framework as phase-based voltage and frequency scaling (PVFS). The Hidden Markov Models are constructed based on samples of hardware counters taken at strategic location using tags. Hardware counters are chosen based on availability of which components DVFS can be applied. Additionally, the location of these tags can be automatically placed and added iteratively based on analysis of the HMM . Because the HMM is produced from this automatic method using hardware counters, this method can be applied to any distributed system even if the components vary greatly, such as System on Chip (SoC) and heterogeneous many-cores. Accurate HMMs produced as a result of tags and sampling will contain hidden states that correspond to particular sections of code with defined performance and power characteristics. We call these hidden state and sections with defined performance and power characteristics *phases*. After, we construct a static DVFS schedule based on tag location.

We evaluate our PVFS method on a test suite of benchmark applications. In order to test the flexibility of our method, we use a cycle-accurate simulator of a Network on-Chip (NoC) based many-core processors. This evaluation allows us to demonstrate the flexibility of performing DVFS on an set of hardware components, such as per-core, network components, or memory. Additionally, NoC many-core processors have shown promise as a future many-core processor architecture, and many evaluates have been done on DVFS for networks. In particular, NoC many-core processors can have networks that account for as much as 60 Watts [1], [2].

II. Background and Related Work

A. DVFS

Schemes for DVFS have been proposed in numerous other work [4], [9], [10]. These schemes reduce total energy and/or thermal hot-spots on many-core processors. For both

1063-6927/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDCS.2015.27

objectives, past work has considered applying DVFS to various mixtures of many-core processor components, such as cores [9], NoC [11], and memory [12]. These DVFS schemes have been applied using compilers [3], operating systems [13], and software [10].

Time-Driven DVFS. One common DVFS scheme uses time-driven sampling of performance and power [7], [8], [12]. In this scheme, a sample is taken at regular time intervals, i.e., epochs, and the efficiency of the sample is compared to the efficiency of the last time interval. DFVS is greedily applied based on how performance and energy efficiency has changed between samples. Due to the popularity, we compare PVFS to a time-driven DVFS implementation.

Compiler-Driven DVFS. Kandemir and Ozturk [3] propose a compiler directed DVFS for CPU/links in NoC based many-core processors. In their approach, the compiler assigns the voltage/frequency domains to CPUs/links after analyzing the program phases involving critical path(s) execution of each phase. Integer Linear Programming is used to solve for the critical path(s) and slack information to determine the voltage/frequency domains for CPU/links.

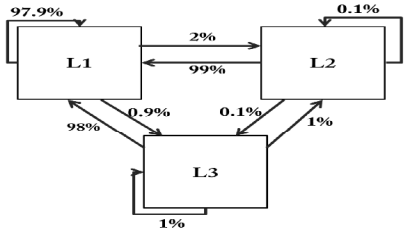
Statistical Application Analysis. Statistical models have been used successfully to model the interactions of applications and hardware [14], [15]. In work by Chen and Aamodt [15], very large Markov Chains are constructed to model the execution of an application. In a similar way, Su et al. [14], build neural network models to represent the system. These neural network models provide more flexibility than Markov Chains, however provide less understanding of interactions of application and hardware. Markov Chains have also been used for energy in distributed senor networks [16].

Relationship to Past Work. Our HMM PVFS is significantly different from previous works in several ways. First, other DVFS schemes have not used explicit statistical models, and this work is the first to use Hidden Markov Models for DVFS. The goal of using these statistical models is to examine the average behavior of small sections in the application to find phases. By using average behavior, phases are found despite rare anomalies such as page faults. Second, our models are different from those of the work of Chen and Aamodt [15] and Su et al. [14]. The statistical models in previous work are large, complex, and intensively trained. Our proposed Hidden Markov Models are smaller, cheaper to construct, and able to detect phases. These three properties make them ideal to schedule DVFS.

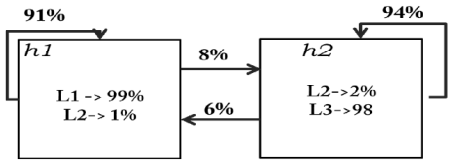
B. Hidden Markov Models

A state space model attempts to capture the *stochastic process*, which is a sequence of outcomes, as defined by states with a set of attributes from a parameter space [17]. In our models, these states can be thought of as phases. Though many statistical modeling techniques exist for stochastic processes, Dynamic Bayesian networks are popular in speech recognition, bioinformatics, and robotics because they attempt to model the state at time t from information at t-1 [17], [18], [19]. The popularity has produced multiple different techniques such as Hidden Markov Models.

Discrete Markov Chains. The most basic form of a Hidden Markov Model is a Discrete Markov Chain. In a Markov Chain, there exist only *observable states*, which are states that can visibly be seen or measured. At each time step t, there exists a probability between 0 to 1 of transition to some other state or back to the current state. On a computer system, these observables could be power states, instructions issued, or hits to a cache level. Consider the example in Figure 1a. In this example, the model has three observable states, namely a hit in L1, L2, or L3 cache. Based on the data reuse pattern, the probability of transitions between states can be found. However, a Markov Chain is limited to observable states, and a code segment has many unobservable transitions.



(a) Markov Chain for cache hits to L1, L2, and L3.
The arrows express the probability of transitioning between states.



(b) Hidden Markov Model of an unknown code fragment that has one phase of high computation and one of low computation.

Fig. 1: Examples of two common Bayesian networks used for modeling application and system interactions.

Discrete Hidden Markov Models. The natural extension to Markov Chain is Hidden Markov Models (HMMs). In a HMM, hidden states exist and represent *unobservable information*, such as execution phases or combinations of unknown observables. Transitions from time t to t+1 are done between the hidden states, and each hidden state may have an emission from a set of observable states. As an example, consider a kernel that has two execution phases, which it cycles cyclically. The first phase is very dense and computationally intensive resulting in high L1 cache hit rates. The second phase is a write-intensive phase that does little computation and accesses a lot of shared data in L3. The resulting model is given in Figure 1b. In this model, $h1$ and $h2$ represent the hidden states which are phases in a trained HMM. Inside each hidden state, a list is given of observable state probabilities. Between the hidden states are arrows marked with the hidden state's transition probabilities.

In order to train a HMM, a discrete sequence of observable states must be collected. The states that compose this sequence represent observations in time. These observation periods must be small enough that measurements in the period do not vary much. Using this sequence, multiple models are trained using various starting transitional probabilities and number of hidden states. The goal is to find the simplest model that accurately mimics the application.

Once an accurate model is found, HMMs provide an array of utilities that characterize the application. The analysis can take three forms. First, the Viterbi Path (VP) [18] can be found that provides the most likely path and long-term probability of the hidden states. The long-term probability provides an estimate of the probability of being in a hidden state and can be found in complexity $O(T\,S^2)$ where T is the number of time steps and S is the number of states [18]. Second, the probability of observing a state can be examined for each of the hidden state, which represents phases. Lastly, the transitions between phases can be analyzed in order to understand the execution of the application.

III. Our Approach

This section introduces our phase-based DVFS that uses Hidden Markov Models (HMMs). We first provide an overview of how to implement our method using three steps. These steps are parametrization, training, and applying. After, we provide an in-depth discussion about model choices, complexity, and ability to apply to various hardware.

A. Steps to Apply PVFS

Parametrization. The first step in applying HMMs for PVFS is to define relevant system metrics, levels of discretization, and free parameter sets of the system in order to reduce the total energy of an application. A system metric is a simple measurement that normally derived from one or more hardware counter, such as instructions per cycles (IPC). A free parameter set is a collection of system metrics that represent the performance and power of a component to apply DVFS, and can contain any number of system metrics. For example, a free parameter set for DVFS on a core may contain the system metric of time per float-point operation, core dynamic power, and number of memory loads. The selection of which system metrics will depend on available hardware counters and objective. If we decide to consider applying DVFS to k different components, we will need k free parameter sets. However, the measurements between free parameter sets may not be unique, e.g., using core dynamic power for both caches and computational cores.

Additionally, our HMMs require discrete measurements, and will require us to discretize our system metrics into levels. The choice of how many levels to classify a system metric into will depend on both the metric and computer system. The number of levels must be large enough to account for variance in the metric and small enough to not over complicate the model. We find that a good selection for the number of levels is between 3 and 5 for most of our system metrics. If m_i represents the number of discrete levels chosen for system metric m, then the total number of direct system metrics in the model will be $\prod_{i=1}^{n} m_i - R$, where n is the number of system metrics in all k free parameter sets and R are repeated system metrics.

Training. In training, iterative tag placement inside the code drives the training of an accurate HMM. A tag is a function call that records the needed hardware counters along with an unique location identifier (id). Additional identifiers could be added to the tag to indicate things such as hardware components the code is executed on. These additional identifies could be used if the distributed system contains different subdevices, such as SIMD or FFT units. At each iteration, data collected from these tags is used to build and analyze a HMM. The iterative placement of tags is done in two steps: initialization and updating. The initialization step places tags at key locations in the code, such as the beginning and end of function calls. The updating step places additional tags inside the code based off the HMM analysis. Updating is done until we achieve an accurate HMM.

Algorithm 1 Initial placement of tags in C code.

```
for all Source Files do
    Find beginning of function based on regex
    Place tag at location
    Scan through function for LOOP using regex
    Scan for end of LOOP by matching "{" and "}"
    if LOOP length is greater than 4 then
        Place tag at beginning location of LOOP
        Place tag at end location of LOOP
    end if
    Scan for end of function by matching "{" and "}"
end for
```

Both the initialization and updating steps can be automated for large application. The initial tag placement should be at key locations such as the beginning and end of function calls and of long loops. A minimal number of tags should be used to ensure that we do not over fit the derived HMM. A general outline for the initial placement of tags is given in Algorithm 1. This algorithm is very similar to parsing syntax into an abstract syntax tree used by compiler.

After tags are placed, a sample of the application is run and models are constructed. This sample should mimic normal code coverage, i.e., the normal execution path of the application. There is no maximum length. The tags will produce a trace of system metrics organized by tag id. Translation of system metrics into discrete levels can either be done at the tag or by processing after the sample run. The sequence of tags can than be used to train a new HMM where each combination of system metrics is an observable. If multiple subdevices are use, multiple HMMs are trained from sequence identified by a device id in the tag.

Algorithm 2 Update tags in C code.

```
Let T be the set of critical tags
Add tags to T from hidden states with probability < τ and number of tags > γ
for all Tags (t) in the HMM do
    for all Hidden State (h) do
        if 100/|H| − ϵ ≤ Probability(t) ∈ h ≤ 100/|H| + ϵ then
            Add t to T
        end if
    end for
end for
for all t in T do
    Scan for function call, loop, or control statement from t
    Add tag at found location
end for
```

Next, the VP of the trained HMM is used to update tag locations. The HMM is searched to find tag ids where additional tags should be added. The search considers two criteria. The first criteria is when a hidden state has a probability lower than some threshold and contains multiple tag ids with diverse

levels of performance. This would represent a hidden state that is not well defined, and tags should be added near other tags within this hidden state. The second criteria is when a tag id is semi-uniformly distributed between hidden states. A tag id that is distributed semi-uniform between hidden state represent a tag that does not represent a single performance window. We note that other distributions can be used if you desire a more aggressive training process. Once these tags are identified, the algorithm scans from the found tag until the next key feature, e.g., a function call, loop, or set of flow control statements (if/else/switch), and a new tag is placed. The generalization of this algorithm is given in Algorithm 2. The algorithm used in this paper is slightly more complicated by considering bi-modal distributions and small transition probabilities.

The iterative process is stopped when an accurate HMM is found. The accuracy of a model is evaluated in two ways. The first way is based on the probability the sequence from the VP matches a sample trace. The second is when we do not need to add additional tags to code based on the above criteria.

Applying. Once an accurate model is found, the hidden states represent phases that define specific performance and power characteristics. Phases where performance and power are disproportional and have a high probability are selected for DVFS. Tags found in phases where power is much higher than performance will be used to scheduled DVFS. Additionally, transition probabilities between this hidden state and others may be used to decide if DVFS is reasonable. Considering these transitions prevents applying DVFS that might harm a different high performing phase. The choice of selecting tags based on local probability and transitional probability also determines DVFS aggressiveness.

B. Discussion

The use of Hidden Markov Models for DVFS is a general framework that can be used on a variety of components and has many choices in how it is implemented. In the result section of this paper, we focus on a very narrow case of a NoC many-core processors with X86-64 compuational cores. However, the general framework may be used on more interesting hardware, such as on heterogeneous processors, using different system metrics, or even using online HMMs.

Hardware. Distributed systems are comprised of a variety of hardware. The variation includes traditional x86-64, NoC, SoC, and more. In particular, this paper looks at applying DVFS to all cores and networks on-chip uniformly. Though no current devices offer per-core and NoC DVFS, applying DVFS with controllers is slowly progressing to lower levels and clusters of hardware components. For example, the work done by Mishra et al. [6] examines how to apply DVFS to a chip-multiprocessor that has levels of controller. In that work, there exist islands, i.e., clusters of cores, each with their own controller for voltage/frequency. Our method could be extended to both this type of islands of cores and per-core DVFS by allowing the creation of an HMM for each controller by marking which controller. A second type of future architecture is heterogeneous processors. Here we define heterogeneous processors as those with multiple unique computational units contained in the package. This includes processors, such as AMD's APU, that mix traditional computational cores with graphics processing units [20]. These systems may have separate knobs to apply DVFS to each computational unit. Our general method can be adapted for these types of hardware by adding an identification tag that indicates which device is being used. Based on this identification, separate HMMs can be trained for each unique unit.

Complexity. Here, we present the most general version of our method. However, this can be made much more complex to fit the needs of the user. Multiple components can be considered simultaneously adding more combinations of system metrics. If the traditional Baum-Welsh algorithm is used to create the HMM, the complexity to train is $O(TS^2)$ where T is the number of time step, i.e., tags in a trace, and S is the number of hidden states [18]. This would be multiplied by how many HMMs are needed for the particular system setup. Therefore, codes that are longer and have more variation will increase training time, but only in a linear sense. The upper bound on S equals the number of combinations of system metrics. However, a HMM with that number of S would not make sense, since most combinations are invalid. For example, you could not have low core power with high instructions per cycle. Hence, S will correspond to the number of unique phase in the application and will not be directly the number of combinations of system metrics. This leads to the HMM cost being nonlinearly with the number of unique phases that exists. The number of unique phases depends on the code, system, and more importantly the number of discrete levels chosen for system metrics. However, we show in the results section that the number of unique phases is relatively small for our benchmark application ($S < 6$). In a number of experiments on Intel Xeon system with larger application, we have a similar observation.

Limitations. Like all methods, PVFS with HMM has several limitations. These limitations include being a static scheduling method and requiring an application with an iterative nature. The first limitation is based on the need to add/move tags in the training process. This training need could be removed if tags where initially placed densely through out the application, and the training was done using regularized optimization to choose a sparse subset of tags. We have not ruled this method out, however the method would present many new implementation issues, such as dealing with large amounts of trace data. The second limitation is that our method works better on codes with an iterative nature. Each time a tag is encountered, it becomes an observation of a HMM. The more a tag is encountered, the more the tag will weigh in the model. In codes that only call all functions once, all segments would be weighted the same even though their run-time length are different. The use of additional tag weights could be added to counter act this limitation, but has not been fully tested in this paper.

IV. Experimental Setup

A. Modeling using HMM

We construct our HMM using MATLAB [21], and consider DVFS of two hardware components. The first component we consider DVFS for is compuational cores. We consider the system metrics of instructions per cycle (IPC) and power on each core for this component's free parameter set. For the system metrics of IPC and core power, we use three distinct

discrete levels. The choice of the number of levels will depend on the metric and system. We found that these metrics are either uniformly or bimodal distributed on our system, and therefore, three bins capture the behavior. Our three levels represent low (L), medium (M), and high (H) values.

The second component we consider DVFS for is NoC. NoC power energy reduction can be more difficult because of diversity in routing. We consider the system metrics of maximum buffer occupancy and network power for the component's free parameter set. Maximum buffer occupancy (BFM) is chosen as it was shown by Das et al. [1] to perform as well or better than many other metrics for modeling network performance. For both these metrics, we use four discrete levels, since there are more variation. We note that other system metrics could be used depending on the underlining system.

B. Benchmarks

We consider in-depth how PVFS affects a test suite of eight benchmarks. These benchmarks are chosen for their diversity in terms of domain, algorithm type, computational intensity, language, and scalability. These benchmarks are given in Table I, and all codes contain multiple phases of computational intensity. Each benchmark has their input data adjusted so their execution time after initialization to completion is similar on 4 cores. We now provide a detailed description of each of these benchmarks in order to better understand the margin of energy improvement that exists.

TABLE I: Test suite benchmarks used to evaluate PVFS.

Benchmark	Suite	Domain
`Freqmine`	PARSEC	Frequent itemset mining
`Dedup`	PARSEC	Data deduplication compression
`Mgrid`	SPECOMP	Multi-grid solver
`Wupwise`	SPECOMP	Quantum chromodynamics
`XSBench(MC)`	CORAL	Monte-Carlo transport
`LCALS`	CORAL	Collection of loops
`CoMD`	MANTEVO	Molecular dynamics
`HPCCG`	MANTEVO	Irregular conjugate gradients

FreqMine. In this benchmark, both tree-based and sorting algorithms are present. These two algorithms have different performance, scalability, and power usage on increasing core counts. The diversity of these two algorithms help demonstrate PVFS's ability to identify different phases. `Freqmine` benchmark applies an array-based version of frequent pattern-growth method (FP-growth) for frequent itemset mining [22]. Frequent itemset methods are important to discover item relationships among large sets. FP-growth is done in two passes over a dataset in order to create a tree of relationships. The `Freqmine` benchmark uses OpenMP to parallelize for-loops in all passes.

Dedup. The benchmark `Dedup` is a compression algorithm that uses pthreads. It uses both local and global compress. Parallelism is exploit in both the pipeline of tasks and data [22]. Many regular data-accesses are made.

MGrid. The benchmark `Mgrid` uses a multi-grid method in order to solve a linear system [23]. A multi-grid method depends on a series of 'V'-cycles. In each'V'-cycle, a large problem is coarsened through a number of projections into a smaller problem, solved, then expanded back into the starting larger problem using interpolation. Each function in the code depends on multiple levels of for-loops parallelized using OpenMP. Many of these individual steps are computationally intensive, since the computation is done only with neighboring array accesses. However, between these computations an update needs to be made so that all threads have a uniform view of the data in memory that could potentially reduce energy efficiency.

Wupwise. `Wupwise` is a quantum chromodynamics code of a Wuppertal Wilson Fermion Solver. The main time consuming section of the code is a computation of quark propagators. This computation is done on a lattice breaking it into sections, and applying solved iteratively [23].

XSBenchmark (MC). `XSBench` (MC) is a mini-application designed to represent the key aspect of Monte Carlo neutronics. Monte Carlo neutron transport codes require a high number of memory lookups. This behavior makes the application ideal for evaluating the performance of the memory system of a shared memory node. However, this same behavior makes the concurrency of the application limited by the amount the processor's memory controllers can simultaneously serve. This behavior is in contrast to many of the loops in LCALS and parts of CoMD that scale well with additional cores. In such an application, dynamic power due to cache and functional units use should remain low, but power will be high for the memory controllers and DRAM. With this benchmark, we wish to investigate if PVFS is able to capture long loads from memory despite not being weighted by iterations.

LCALS. The Livermore Compiler Analysis Suite (`LCALS`) benchmark contains a large collection of loop kernels from the historical Livermore Loops [24] and loops taken from applications. This suite was created to emphasize floating-point operation performance in the performance of simulation codes. The code enables the study of single instruction multiple data (SIMD) vectorization and OpenMP thread level parallelism. Due to `LCALS`'s size, only the standard OpenMP loops will be considered in this work. These include loops taken from partial-in-cell computations, pressure calculations, energy calculations, and multiple level loops. While some loops execute at peak performance, others make irregular accesses that execute at only a fraction of this peak performance.

CoMD. The `CoMD` benchmark is a proxy-app for typical molecular dynamics applications. This benchmark uses a regular grid data structure to store elements. Each step in time is an iteration that must calculate potential energy, force, and displacement of each element. For force calculations, the forces between each element must be calculated within these blocks. Data accesses for these calculations are done in array order creating locality during an inner loop resulting in good performance. The effects of elements between blocks are handled through a halo that limits the need to only access neighboring blocks. However, elements may have to move to different blocks after displacement creating irregular accesses.

HPCCG. `HPCCG` is a sparse conjugate gradient on a 3D chimney. It has recently gain popularity and cited by HPCWire [25], since it has been promoted as a benchmark on usable float-point operations on HPC systems. The original goal of the benchmark is to be used as a companion to Linpack benchmark to understand how communication affects peak perfor-

mance [25].

C. Simulator Setup

In order to evaluate HMM based PVFS, we will use a cycle accurate simulator. The choice of using a simulator is two fold. First, we wish to test how well our method works on a system with multiple components, such as an NoC based system, and currently we do not have an NoC based many-core on which we can collect execution statistics through performance counters. Most current work foresees future many-cores being NoC [3]. Second, we would like the ability to adjust the frequency and voltage at each on-chip network router in unison. The ability to adjust these values at this granularity does not yet exist in current market processors. As our simulator, we use Sniper [26]. Sinper is a multi-core parallel cycle-accurate x86 simulator that uses McPAT [27], i.e., an integrated power model framework, to measure the power and energy of the simulated processor. Furthermore, Sniper uses a system call emulation mode where all the system calls are emulated by the simulator making it fast for the user-level code experiments. DSENT [28] is used in order to model the power of NoC, and has been shown to be accurate for both electric mesh and optical networks. We model traditional 4 and 8 core processors, along with 16 and 32 core NoC processors. Each core is an out-of-order Ivybridge like microarchitecture core. Each core has a private L1 and L2, and all cores share an L3. Additional details of our setup are found in Table II. Sniper magic instructions are used to process PVFS tags and DVFS. When a tag is encountered during training, the needed hardware counters are written to form traces of the tags. During evaluation, the tags act as point to apply PVFS 's static DVFS, and the tags indicate if DVFS should be applied. In our experimentation, we use the voltage and frequency levels in Table III when applying DVFS.

TABLE II: Simulated chip description.

Core	22nm Ivybridge
L1 cache	32KB, 8-way associative, Access Latency: 4 cycles
L2 Cache	256KB, 8-way associative, Access Latency: 8 cycles
Coherence	MESI
L3 Cache	2MB/core, Shared, 16-way associative, Access Latency: 30 cycles
NoC	2 cycles latency; 64 bits/cycle 16 cores (4x4); 32 cores (8x4)
DVFS Latency	2 usecs (Both core and NoC)
DRAM	45 nsecs Latency; DDR3-1600; per-controller b/w: 7.6 GB/sec; 4-8 cores use 1 controller; 16-32 cores use 4 controllers

TABLE III: Voltage and frequency levels used by DVFS.

Frequency (MHZ)	3000	2750	2500	2250	2000
Voltage (V)	1.06	0.95356	0.928281	0.866188	0.80653

D. Baseline DVFS

In our experiments, we compare our method in terms of dynamic power (P), time (T), and energy (E). Measurements are parametrized by application (app), number of cores (p), and method (m). We consider the four methods of $baseline$, PVFS, EL, and EDP. $Baseline$ refers to measurements with no DVFS. PVFS is our HMM DVFS method, and the other two are detailed below.

Energy Limit (EL). Energy Limit (EL) is a lower bound estimate on energy that an application could use. This method allows us to understand what theoretically could be the maximum savings for our application or pseudo-optimal. We define this as the product of the minimal power observed during actual computation and the time for execution of the $baseline$ case, i.e.,

$$EL(app, p) = T(app, p, baseline) * min(P(app, p, baseline)).$$

Energy Delay Product (EDP). We additionally compare to the energy delay product (EDP) method of DVFS. This is a time-driven DVFS scheme by Swaminathan et al. [8], and is popular on many systems. However, EDP may not scale well with increasing core counts as the time between samples needs to be decreased with increasing cores. This method uses the energy delay product (EDP_t) and instructions committed (I_t) in every epoch (Δ_t) as a measure to choose when DVFS should be applied. Additionally, a tolerance $\gamma \in (0, 1)$ is selected to be used by all application to prevent frequent switching and overhead. We pick γ to be conservative to limit the penalty time to be similar to our method. For our baseline, we fix our epoch to be 1msec as this provides some of the best result across benchmarks. DVFS is applied in the following manner using these measurements: if EDP_{t+1} and I_{t+1} is greater than γEDP_{t+1} and γI_t, then DVFS level does not change; if EDP_{t+1} and I_{t+1} is less than γEDP_{t+1} and γI_t, then voltage and frequency reduces one level; if only I_{t+1} is greater than γI_t, then voltage and frequency increase.

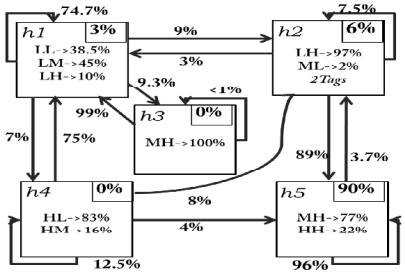
V. Empirical Analysis of HMM for DVFS

In this section, we examine the result of applying our HMM PVFS on our test suite. We choose to provide a detailed analysis over a small set of diverse applications in order to demonstrate how and why HMMs are a flexible method for PVFS. This section is divided into subsections based on application while focusing on the 32 core case. Furthermore, we observe that HMMs for 4 and 8 core systems are relatively similar, and the HMMs for 16 and 32 cores core systems are relatively similar. However, a measurable difference can be found between non-NoC and NoC based systems. Our method's overall impact on energy and time across different core counts is provided in the last subsection.

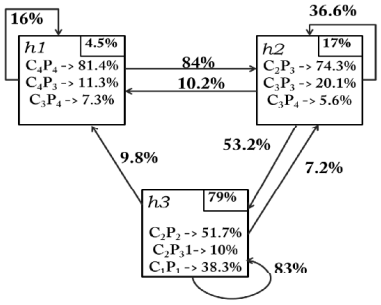
A. HMM of Individual Applications

`Freqmine.` `Freqmine` has two main computational sections, namely tree-based and sorting algorithms. Eight tags where initially placed automatically at the beginning and end of functions called by main. We train models considering both core and NoC performance/power.

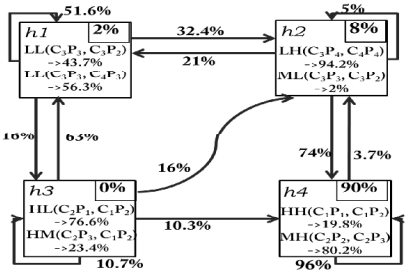
In Figure 2a, the HMM is given for just cores, which results in using 14 tags. Most of these additional tags fall into the tree-based sections and reduction phases between threads. In each block, the long-term probability of being in that hidden state is given in the upper right-hand corner. This value is found by examining the Vitberi Path of the model. Furthermore, a



(a) `Freqmine` Core HMM.



(b) `Freqmine` NoC HMM.



(c) `Freqmine` Core/NoC HMM.

Fig. 2: Hidden Markov Models for `Freqmine`.

list of all probabilities is given for the system metrics based observable states emitted by that hidden state. The first letter represents performance and the second power in each pair. In particular, the HMM for `Freqmine` has 5 hidden states with the long-term probability of each as 3%, 6%, 0%, 0%, and 90% .

In order to apply DVFS, we *identify phases with low performance and high dynamic power*. Ideally, we find phases where the power level is higher than performance, such as LM (low IPC/ medium power) and LH (low IPC/ high power). However, not all tags with this difference are ideal candidates for DVFS. For example, performance transition states exist. In these performance transition states, power must be increased before performance can be increased, and applying DVFS to these tags would harm the performance of the application. Therefore, we find phases emitting low performance, high power observable states with high long-term probability and with limited transition probability to phases with high performance. Several phases meet this criteria to varying degree, namely $h1$ and $h2$. Two tags in phase $h2$ are chosen as they do not have any tags that overlap with $h5$, and $h1$ is mostly dominated by low and medium dynamic power levels. These 2 phase correspond to sections of the application that make irregular data access while building the tree and thread barriers for reductions. A schedule is constructed that applies DVFS when encountering the 2 tags in the $h2$ and applies the original voltage/frequency to all other tags. This schedule results in a 6.9% relative energy savings by reducing the voltage to as low as .8v, and each step down for 21msec.

In Figure 2b, the HMM for just NoC is given using 6 tags. Here, there exists only three phases. The discrete levels of system metrics are represented with C_i for network contention and P_i for network power, where $i = 1$ is the smallest level and $i = 4$ is the largest. Hidden state one ($h1$) corresponds to a phase where network contention and power is low. However, $h3$ is a phase with high network contention and power. We do not apply DVFS to only NoC as NoC power is only a small fraction of total power. The largest observed percentage of NoC power to core power is for `MC`, and is10.3%. For `Freqmine`, this percentage is 4.2%. The more important instance is when applying DVFS to both cores and NoC. This case has the opportunity to reduce total energy the most. However, this instance may not even succeed in achieving the energy performance of only applying DVFS to each core if not handled correctly. For example, based on the HMM in Figure 2b, DVFS would be applied to several tags ($h2$). One such tag corresponds to an initialization and sort function in the application. In this function, NoC contention is low, but this tag is in a transition period, leading to medium performance that may be harmed by waiting for network accesses.

In order to take advantage of the full opportunity, both core and NoC must be considered together. In Figure 2c, a HMM that considers core perform, core power, network contention, and network power is constructed using 12 tags. This results in a very different four phase model. In HMMs that consider all system metrics, we introduce a notation that states the core performance and power pair followed by a set of top network contention and power pairs, i.e., $LL(C_1P_1, C_1P_2)$. Here we apply DVFS to 3 tags. Two tags we apply DVFS to both cores and NoC. The third we only apply DVFS to NoC. We save 8.78% energy by moving down to .8V in the core while moving up to 1.1V at the first two tags selected. Here we can see a true benefit of using PVFS to save energy in many-core processor systems where individual hardware components allow to DVFS.

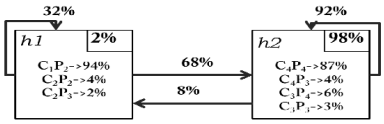
`Dedup`. Initially, 10 tags are used to train the HMM. The model for this application trains significantly faster than other applications in the test suite. The main reason for this is the ability to find spawning of pthreads quickly. Modeling that only considers cores contains 4 hidden states, and 4 tags where selected to apply DVFS. This resulted in approximately 10% energy reduction, even though the tags were short compared to those of `Freqmine`. However, when NoC is considered, the HMM does not change much, but the individual tag location in the HMM changes greatly. Based on this newer model 3 tags were used for core DVFS and 2 for NoC DVFS. Only slightly more energy was saved. This small increase was due to `Dedup` relatively small NoC traffic during different phases of the application.

`MC`. Unlike `Freqmine` that is all computational after initial-

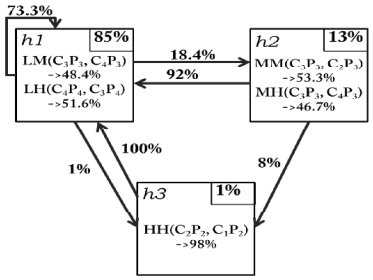
ization, MC spends much of its time making memory requests. Due to this behavior, core network plays a greater deal more important than other applications in this suite. In particular, MC has an injection rate, i.e. flits/node/cycle, similar to the synthetic or server workloads used in many NoC DFVS work [1]. However, MC lacks the iterative nature of many other applications. Because of this nature, only 5 original tags are automatically located.

A HMM considering core performance and power results from finding 13 tags and 4 phases. Three phases are dominated by low or medium performance, and only one of these has high power. However, this phase is small, and applying DVFS to the 3 tags in this phase results in a savings of 1.9%. Additionally, static DVFS is applied uniformly in steps at each iteration of a tag, and the lack of iterative nature takes DVFS up to 34msec to reach a voltage of .8V. However, a better method that bases HMM weights may be better than applying uniformly in such cases. In Figure 3a, the HMM for NoC contention and power results in a 2 phase model constructed from 10 tags. From this model, phase $h2$ stands out as having high contention and power, while $h1$ has relatively high power for having low network contention.

By considering both core and NoC, the HMM is formed in Figure 3b. This HMM is constructed using 12 tags and contains 3 phases. We observe a clear phase ($h1$) where DVFS can be applied to decrease the frequency and voltage on core while increasing the frequency and voltage for NoC. Here, 3 tags are selected, and result in 4.59% energy savings.



(a) MC NoC HMM.



(b) MC Core/NoC HMM.

Fig. 3: Hidden Markov Models for XSBench (MC).

Mgrid. Mgrid is a multigrid application that follows a 'V'-cycle. This application has a number of communication steps separating each level of coarsening and interpolation. This partitions Mgrid into very distinct phases. We initially tag Mgrid with 5 tags based on calls in the main function.

In Figure 4, the HMM is provided for a model of core performance/power using 10 tags. We can observe 3 distinct phases in this model. In this model, phase $h1$ acts as a center, and all paths between phases have to pass through it. The tags in this phase correspond to 2 key areas. The first area is the communication phase that is done between the projection and interpolation steps. The second area corresponds to the function used to solve with pseudoinverse. In communication phases, voltage can be reduced to .86V and only one step of DVFS is applied to the pseudoinverse. This leads to 9.1% energy savings when using DVFS on 3 tags.

The HMM for NoC contains 2 phases when built with 10 tags. These phases are clearly divided into one phase with high network contention and power and one phase with low network contention and power. However, each phase has many free parameter sets with low to medium probability making it difficult to select tags for DVFS.

The HMM that considers the combination of both core and NoC changes little. Again, three distinct phases are found that are similar to those in the HMM of just core performance/power. In this case, considering on-chip networks did not provide extra room for improvement, however it is impossible to see this without first creating HMM. Using similar three tags to apply DVFS results in a decrease of 10.1% in energy.

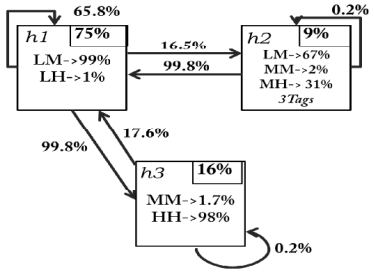
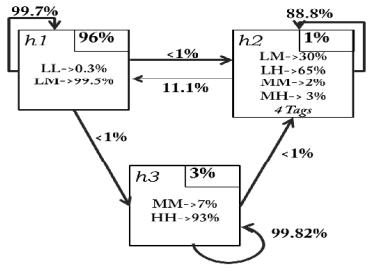


Fig. 4: Hidden Markov Model (Core) for Mgrid.

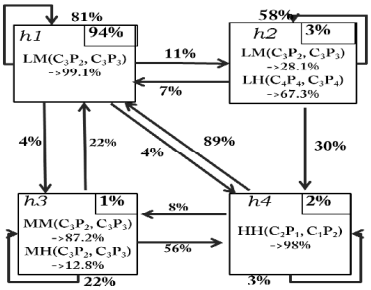
Wupwise. This application is dominated by a section of high computation, which leaves little room to apply DVFS. The HMM considering only core results in 12 tags being used to provide 3 hidden states. Using 3 tags, the energy savings is almost 7%, but only reduces to .9V. Because dominated by computation, considering NoC does not provide much additional savings.

LCALS. The LCALS benchmark is a collection of multiple loops. These loops have very different data access patterns, but can be relatively short. As a result, the HMM for core is provided in Figure 5a. This application has 15 tags in order to find a stable model. In phase $h2$, there exists a high probability of having low performance while using a high amount of dynamic power. The other two phases have relatively good ratios of performance and dynamic power, and we will only apply DVFS to the four primary tags found in $h2$. DVFS can be applied faster than 10msec, and varies greatly depending on the tag location. This leads to an energy savings of 7.1%.

In contrast, the HMM for NoC with 4 distinct level of network contention only has one phase. This makes the model useless for the DVFS, since we cannot find a unique phase. The reason for this failure is that NoC contention stays relatively similar throughout the entire application. However, this small variation in network contention is enough to show up when



(a) LCALS Core HMM.



(b) LCALS Core/NoC HMM.

Fig. 5: Hidden Markov Models for LCALS.

mixed with core performance. In Figure 5b, the HMM for combined core and NoC performance and power that uses 16 tags. Though this model has 3 phases, the phase of interest, i.e., phase $h2$, has slightly different tags than HMM using only core performance. This difference leads to 2 different tags being used for DVFS. These tags lead to a reduction of 7.9%.

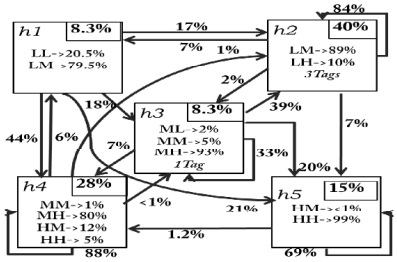


Fig. 6: Hidden Markov model (Core) for CoMD.

CoMD. This classical molecular dynamics application leads to one of the most complicated HMMs for all cases. Initial tags were placed in the main function, but no tags were placed inside the time step. However, the HMM built for core results in having 12 tags throughout time step and its function calls. This HMM is found in Figure 6, and we observe the complex transition between its 5 phases. Despite being complicated, the HMM is able to identify two phases $h2$ and $h3$ that have significantly higher dynamic power than performance. Applying DVFS results in a 7.1% energy reduction.

The HMM for NoC leads to a model with only 3 phases. These phases are similar to those in Figure 2b except with different long-term and transition probabilities. Because the HMMs for core and NoC are so different, it would be difficult to find a subset of tags that intersects the two models for DVFS.

However, the free parameter sets of core and NoC results in 14 tags. This model has 5 phases. These phases slightly differ from those in HMM for only cores, as the transitional phase $h3$ becomes more pronounce. We apply DVFS to only the core in these transitional phases as power seems to match the network contention. DFSV for both core and network is applied in phase $h2$, and DFVS to only NoC in phase $h5$. Using this scheme, the energy reduction is improved to 9.8%. This major increase comes from being able to independently adjust voltage and frequency of the core and NoC, while knowing how the two are related.

HPCCG. HPCCG is an application designed to benchmark the obtainable performance of an sparse scientific application. It is relatively simple, but requires 11 tags to create an accurate HMM. This HMM was very close to Wupwise, but results in slightly more energy reduction. We note that the modeling is very different from the single node OMP version presented here and the message passaging implementation that is present on large HPC machines. When considering NoC in addition to core, the HMM is similar but new tags are added to the phase with low performance and high power. These tags correspond to locations of switching sparse kernel calls in the inner CG iteration. Though short, these tags allows the DVFS to obtain a level much closer to EL.

B. PVFS Across Cores

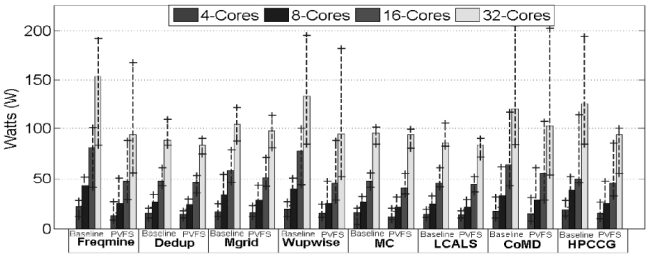
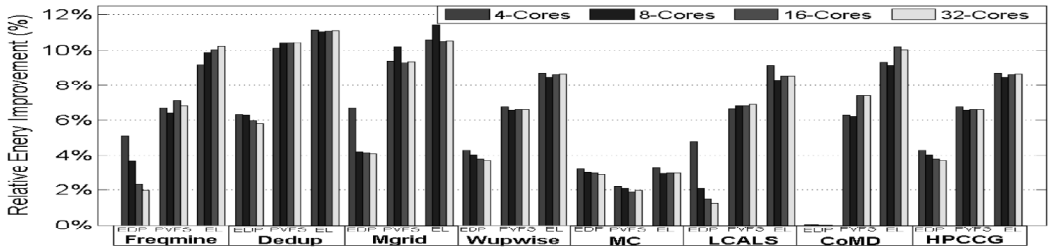


Fig. 7: Dynamic Power without and with PVFS. Average power is given with bars, while the maximum and minimum marked with "+".

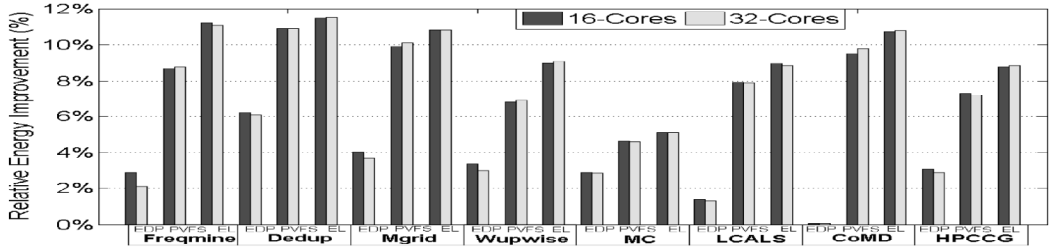
We first consider the change in dynamic power when applying PVFS for cores. In Figure 7, we provide a bar chart of the average dynamic core power of each application without DVFS and with PVFS. Here, we present only the dynamic power as the leakage will be the same for a fixed number of cores, but will increase proportionally with the number of cores. In each grouping of application, a bar is given for each core count considered. Additionally, the observed maximum and minimal power is marked with an "+". Even with leakage power ignored, the minimum, average, and maximum power of the application increases with the number of cores. However, we notice that with PVFS the minimum, average, and maximum power of all applications decrease. This trait is most noticeable in the float-point computations of Wupwise and HPCCG.

Next, we examine the improvement in terms of energy. We define the relative energy improvement ($REI(app, p, m)$) as

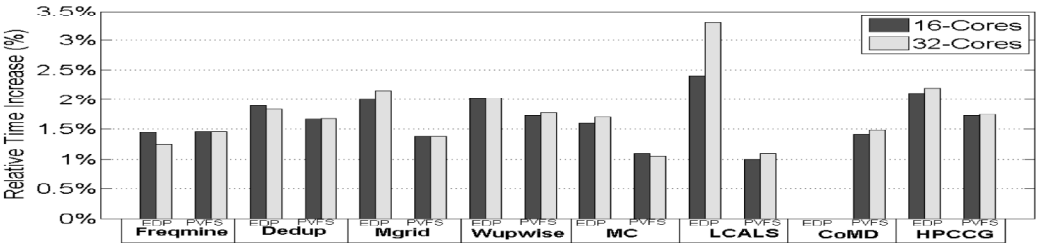
$$REI(app, p, m) = \frac{E(app, p, baseline) - E(app, p, m)}{E(app, p, baseline)}.$$



(a) Relative energy reduction ($REI(app, p, m)$) for EDP, PVFS, and EL where larger is better for core.



(b) Relative energy reduction ($REI(app, p, m)$) for EDP, PVFS, and EL for core and on-chip network.



(c) Relative time increase $RTI(app, p, m)$ for EDP and PVFS for core and on-chip network.

Fig. 8: The impact of applying EDP and PVFS to the benchmark suite in terms of relative changes of energy and time.

In Figure 8a, we consider $REI(app, p, m)$ of EL, EDP, and PVFS for only DVFS on core. We first notice that PVFS reduces energy more than EDP in all cases except on `MC`. On 32 cores, PVFS is able to reduce energy of four benchmarks over 6% while EDP can only achieve an energy reduction above 6% in one case that uses 4 cores. Additionally, PVFS is able to achieve a similar energy reduction on all core counts for most applications, however EDP energy reduction can vary as much as 3.2% with additional cores. More importantly, EL provides a guideline of best possible energy use of the application. PVFS in the worst case (`MC`) achieves 64% of EL and up to 94% of EL for `Dedup` for only cores.

In Figure 8b, we consider $REI(app, p, m)$ of EDP and PVFS on our test suite of applications when combining core and NoC. EDP is done for both core and NoC. Results are only reported for 16 and 32 core systems because these are the only NoC systems simulated. Across all applications and core counts, we observe at least slightly better energy savings. In some cases, such as `MC`, this better energy savings can substantial. When we compare against EL, PVFS in the worst case (`Wupwise`) achieves 75% of EL and up to 95% of EL for `Dedup` when considering both core and NoC. These ratios are better than using just core, and the ability to improve energy impact with multiple components has been shown to be a much harder problem [7]. In particular, Deng et al. [7] show with CoScale that uncoordinated DVFS is not reasonable and that coordinated DVFS can be difficult.

Additionally, all DVFS methods come with some time penalty, which is also commonly known as performance degradation. Here we define our time penalty as the relative time increase ($RTI(app, p, m)$), i.e.,

$$RTI(app, p, m) = \frac{T(app, p, m) - T(app, p, baseline)}{T(app, p, baseline)}.$$

In Figure 8c, we present the RTI while using both core and NoC. We do not present RTI with only cores because the results are similar. Here we see that EDP can have as high as a 3.2% penalty, while PVFS has only 2.4%. Overall, we see that PVFS has better penalty to EDP.

In work on CoScale [7], Deng et al. show they can use an EDP method to apply DVFS to cores and memory. In that work, they show an energy savings of around 20%, however, they allow for up to a 10% RTI. Our selection of tags is more conservative in impacting time, though a more aggressive selection could be made. In comparison, they show that if they bound RTI to 1% and 5% they get an average energy savings of 4% and 9%. This demonstrates that we are able to achieve similar energy reduction as the current state of the art while still allowing for more flexibility to use multiple components.

VI. Conclusions and Future Work

This paper introduces a generic framework using Hidden Markov Models to detect phases in applications based on hardware component measurements, and we use these found phases to construct a static schedule for DVFS. We call this scheme phase-based DVFS (PVFS). The flexibility of this scheme is that the user can choose any number of measurements to model the interaction between application and system. In order to explore the potential of PVFS, we analyze the use of PVFS on eight very different benchmarks. These benchmark applications vary in language, core algorithms, data access pattern, and read/write-intensity. For each benchmark, we use our framework for DVFS on core and combination of core and on-chip network. Additionally, we believe that an online dynamic framework can be derived in the future. Our current analysis includes explaining the HMM found for each benchmark, our selection of tags for DVFS using the HMM, and the comparing the resulting HMM after PVFS is applied. As a result of this PVFS, we observe that we have an energy reduction as high 10.1% while only impacting time by at most 2.7% while applying DVFS only to core. When considering both core and on-chip network, we can save up to 10% on a higher number of cores and have less impact on time. PVFS substantially reduces the energy more than an epoch method (EDP) for all but one benchmark. More importantly, PVFS is able to have continued energy improvement on 4 to 32 cores, while EDP's ability to reduce energy reduces with the number of cores.

Acknowledgement

We acknowledge supprot from NSF grants 1213052, 1205618, 1439021, 0963839, and 1017882, along with grant support from Intel.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL8500.

References

[1] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 320–331.

[2] A. Mishra, R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C. Das, "A case for dynamic frequency tuning in on-chip networks," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 292–303.

[3] M. Kandemir and O. Ozturk, "Software-directed combined cpu/link voltage scaling fornoc-based cmps," in *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '08, 2008, pp. 359–370.

[4] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey, "A voltage reduction technique for digital systems," in *Solid-State Circuits Conference, 1990. Digest of Technical Papers. 37th ISSCC., 1990 IEEE International*, Feb 1990, pp. 238–239.

[5] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani, "Crank it up or dial it down: Coordinated multiprocessor frequency and folding control," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 210–221.

[6] A. Mishra, S. Srikantaiah, M. Kandemir, and C. Das, "CPM in CMPs: Coordinated power management in chip-multiprocessors," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–12.

[7] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating cpu and memory system dvfs in server systems." in *MICRO*. IEEE Computer Society, 2012, pp. 143–154.

[8] K. Swaminathan, E. Kultursay, V. Saripalli, V. Narayanan, M. Kandemir, and S. Datta, "Improving energy efficiency of multi-threaded applications using heterogeneous cmos-tfet multicores," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, Aug 2011, pp. 247–252.

[9] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *HPCA*. IEEE Computer Society, 2008, pp. 123–134.

[10] M. Kandemir, T. Yemliha, and E. Kultursay, "A helper thread based dynamic cache partitioning scheme for multithreaded applications," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11, 2011, pp. 954–959.

[11] A. K. Mishra, R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C. R. Das, "A case for dynamic frequency tuning in on-chip networks," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 292–303.

[12] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "MemScale: Active low-power modes for main memory," *SIGPLAN Not.*, vol. 47, no. 4, pp. 225–238, Mar. 2011.

[13] Intel, *Mobile Intel Pentium III Processor-M Design Document*, 2003.

[14] C.-Y. Su, D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. de Supinski, and E. A. Leon, "Model-based, memory-centric performance and power optimization on NUMA multiprocessors," in *Proceedings of the 2012 IEEE International Symposium on Workload Characterization*, San Diego, CA, nov 2012.

[15] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model," in *International Symposium on High-Performance Computer Architecture*, 2009, pp. 329–340.

[16] W. Zhao and W. K. Ng, "Predict energy consumption of trigger-driven sensor network by markov chains," in *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on*, June 2011, pp. 350–356.

[17] D. Minh, *Applied Probability Models*. CA: Duxbury, 2001.

[18] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," in *Proceedings of the IEEE*, 1989.

[19] Q. Diao and J. Song, "Prediction of cpu idle-busy activity pattern," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, Feb 2008, pp. 27–36.

[20] AMD, "Compute cores, white paper," https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, accessed: 2014-12-01.

[21] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[22] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[23] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPEComp: A new benchmark suite for measuring parallel computer performance," in *Proceedings of the International Workshop on OpenMP Applications and Tools*, ser. WOMPAT '01. London, UK: Springer-Verlag, 2001, pp. 1–10.

[24] F. McMahon, "The livermore fortran kernels: A computer test of the numerical performance range," LLNL, Tech. Rep., 1986.

[25] HPCWire, "Podcast: Update on the hpcg benchmark with michael heroux," http://www.hpcwire.com/podcast/update-hpcg-benchmark/, accessed 2015-03-12.

[26] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 52:1–52:12.

[27] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 469–480.

[28] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, May 2012, pp. 201–210.