Applying Statistical Machine Learning to Multicore Voltage & Frequency Scaling

Michael Moeng
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
moeng@cs.pitt.edu

Rami Melhem
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
melhem@cs.pitt.edu

ABSTRACT

Dynamic Voltage/Frequency Scaling (DVFS) is a useful tool for improving system energy efficiency, especially in multicore chips where energy is more of a limiting factor. Per-core DVFS, where cores can independently scale their voltages and frequencies, is particularly effective. We present a DVFS policy using machine learning, which learns the best frequency choices for a machine as a decision tree.

Machine learning is used to predict the frequency which will minimize the expected *energy per user-instruction* (*epui*) or *energy per (user-instruction)*^{2} (*epui*2). While each core independently sets its frequency and voltage, a core is sensitive to other cores' frequency settings. Also, we examine the viability of using only partial training to train our policy, rather than full profiling for each program.

We evaluate our policy on a 16-core machine running multiprogrammed, multithreaded benchmarks from the PARSEC benchmark suite against a baseline fixed frequency as well as a recently-proposed greedy policy. For 1ms DVFS intervals, our technique improves system *epui*2 by 14.4% over the baseline no-DVFS policy and 11.3% on average over the greedy policy.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); I.2.6 [**Learning**]: General

General Terms

Performance, Management

Keywords

Power Management, Multicore, Decision Tree

CF'10, May 17–19, 2010, Bertinoro, Italy.

1. INTRODUCTION

Dynamic Voltage/Frequency Scaling (DVFS) is an effective tool for improving energy efficiency for a chip multiprocessor (CMP). It is important for a policy to adapt depending on the workload because workloads exhibit different behaviors and have different optimal frequency settings, as shown in Fig. 1. Machines like the AMD Opteron [5] support per-core DVFS, which can be especially effective when threads with varying behaviors are running simultaneously.

As processor core counts increase, effective DVFS becomes both more important since the additional cores consume more energy; and more difficult to achieve due to the growing search space of possible frequency settings. In addition, complex core interactions with shared resources—such as cache or on-chip network—further complicate multicore DVFS policies.

We recognize the complexity of finding an optimal frequency setting for all workloads and turn to machine learning. We train the system on workloads with random voltage/frequency settings and find the best expected frequency given measurements from hardware counters. This data is then compressed into a decision tree, which can be efficiently accessed at run time.

There are many choices we face when utilizing machine learning for DVFS. We first choose good metrics to draw from hardware counters. These consist of measurements used to characterize the core's environment, and are influenced by our choice of optimization metric to represent energy efficiency. The machine learning package we use, WEKA [12], also has several parameters which we analyze.

In our setting, program execution is broken up into time intervals. The machine's Operating System (OS) changes the frequencies based on input from hardware counters. The OS's DVFS controller makes some brief calculations to obtain some *measurements* (common hardware counter measurements include cache miss rate and instructions per cycle, IPC). The DVFS controller then uses these measurements to choose a new *frequency* and *voltage*; these settings should minimize the energy consumed per (user-instruction)^2 (*epui*2), our primary goal metric which combines energy efficiency and throughput. In our work, we assume that frequency and voltage are scaled together and are related linearly to one another, which is shown to be a fair assumption in [10]. With voltage related linearly to frequency, the problem is reduced to selecting only frequency. We show the effect of our approach for various DVFS time intervals: 1ms, 250μs and 50μs. Because choosing the next frequency should be

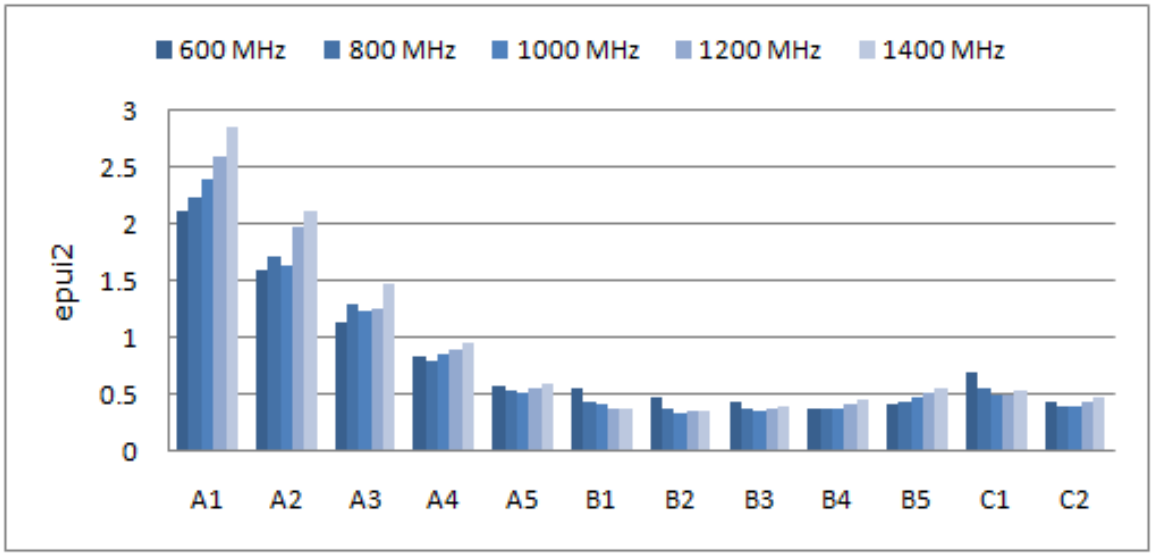


Figure 1: *Energy per (user-instruction)*^{2} for twelve different workloads at homogeneous, static frequencies. Workloads and simulation environment are explained in Sect. 4.

a quick process, especially for shorter DVFS intervals, the OS's DVFS controller uses hardware support in the decision-making process for faster calculations.

Our DVFS policy is simulated on a 16-core machine running multiprogrammed, multithreaded benchmarks. We compare against a fixed, baseline frequency as well as a recently proposed greedy policy [10]. Our results show an improvement in *epui*2 of 14.5% over the baseline fixed frequency and 11.3% on average over the greedy policy.

This paper is organized as follows: we cover related background work in Sect. 2. We then introduce our learning-based scheme in Sect. 3. Our experimental setup is described in Sect. 4, and results are presented in Sect. 5. Our results include a detailed study of certain choices we made developing this technique, and help motivate the differences between our technique and past work. We conclude in Sect. 6.

2. BACKGROUND

There are many works which study software-based DVFS. In general, these approaches would generate too much run time overhead for use with smaller DVFS intervals, but can work well in conjunction with operating system scheduling periods. Teodorescu et al. [18] proposed a linear programming-based algorithm, *LinOpt*, to find good frequencies when dealing with core variation. They assume that some form of profiling provides parameters for every thread-core combination, and do not study the problem of training to find parameters. They also make several linearizing assumptions which was shown in [10] to hurt the accuracy of the linear programming.

Herbert and Marculescu analyzed several hardware-assisted DVFS policies [9], focusing on trade-offs from using per-core DVFS versus grouping cores into Voltage/Frequency Islands (VFIs), which share the same voltage and frequency setting. Multicore VFIs lose some flexibility in the range of possible frequency settings, but have a lower hardware cost to implement. In [10], Herbert and Marculescu consider process variation's effect on core power consumption and performance. They also improve upon the best policy from [9], *greedy*, and show that it outperforms *LinOpt*.

Networked clusters have always been an important target for energy-saving techniques, and some ideas can carry over to CMP energy-saving techniques. Generally, overhead is more of an issue for CMPs, which support shorter DVFS intervals. Freeh et al. [7] present a profile-based DVFS policy for server clusters running high performance computing applications parallelized with MPI. Their work uses a heuristic similar to the greedy policy to generate training data for phases. They record data to generate an energy-time graph, and partition it into example cases which guide the final frequency/voltage setting. We use a random policy for training rather than a heuristic. We also use a decision tree rather than checking all profiled cases to guide the final VF selection. Finally, while our work uses profiling, it generalizes profiles to work even on programs which may not have been profiled.

Some past DVFS policies targeted Multiple Clock Domain systems, usually a single-core processor and an L2 cache supporting independent frequency/voltage settings. Compared with CMPs, these domains are more heterogeneous and interact more directly with one another. Magklis et al. [13] target a Multiple Clock Domain processor by first profiling to identify phases. A program's phases are organized in a task graph, and task frequencies are set so tasks complete at the same speed as the slowest (at the same speed as tasks on the critical path). Finally, all tasks are then slowed to meet a slowdown threshold to further reduce energy consumption.

AbouGhazaleh et al [1] introduced a DVFS policy using machine learning also targeting a Multiple Clock Domain processor, with independent core and L2 clocks. They exhaustively check each program phase running all frequencies and use machine learning to build a decision tree for use at run time Our work is similar in that it also uses machine learning to learn a DVFS policy, but is targeted at multicore processors.

Dhiman and Rosing propose an online-learning scheme [4]

$(m1, m2, m3)$	($epui2$, *number of times encountered*)				
	0.6 GHz	0.8 GHz	1.0 GHz	1.2 GHz	1.4 GHz
(2.1,3.5,1.8)	(4.5,6) (5.0,9)	(5.0,12)	(5.5,13)	(4.5,7) (6.0,8)	(6.0,11)
(4,1,4)	(6.0,10)	(5.5,3) (6.0,8)	(5.5, 8) (5.0,1)	(4.0,11)	(4.5, 5) (4.0,3)

Table 1: $(m1, m2, m3)$ represent observed measurements. Subsequent columns represent observed $epui2$ values resulting from choosing various frequencies and the number of times these values occurred during training.

for single-core processors. While this scheme does not need training data, it requires a predetermined "μ-average" for each frequency/voltage setting towards which online training converges. In this setting, "μ-average" represents the portion of on-chip to off-chip computation each frequency is best suited for. Our work targets systems with multiple cores, where core interactions make DVFS more complex than simply determining the ratio of on-chip to off-chip computation.

Compiler-based approaches are an alternative to taking runtime measurements and feeding them to a policy. One example is from Shirako et al. [16]. The compiler divides computation into chunks and determines the slack available to that chunk. It can then slow down a core to use up that slack, or shut down the core completely if the slack is large enough. Our work uses runtime information not available to the compiler and targets multiprogrammed workloads that affect each other indirectly through core interactions.

3. LEARNING A DVFS POLICY

A system's optimal DVFS policy is heavily dependent on its architecture and workload. Multiple cores, with independent voltage/frequency domains, add complexity to a DVFS policy by exponentially increasing the search space of frequency combinations. Machine learning therefore becomes an attractive option, as it deals well with large search spaces [14].

Contention between cores further complicates the search for good frequencies. Cores share lower-level caches, network bandwidth, and other off-chip resources. If a core is insensitive to interference from other cores on its cache and network, it may assume a higher resource availability than is actually the case, resulting in wasted cycles. Ideally, a core should adjust its policy based on program behavior and the degree of interference from other cores.

3.1 Table-based learning

Statistical machine learning policies can be thought of as a table lookup. In this setting, each core's DVFS controller looks up a table of past measurements to find which V/F setting has worked best in the past. Note that we are not using the actual workload as one of the inputs to the DVFS controlller as our approach does not rely on workload-specific profiles.

First, the learner must have training data. Sample workloads, consisting of multiprogrammed, multithreaded benchmarks, are run to gather data; every DVFS interval, each core *randomly* chooses a new frequency and relevant data is gathered from hardware counters and written to a file—this process repeats for a number of intervals. Our experiments use a fixed number of intervals for training time per workload, ranging from 2,000 to 40,000 for 1ms and 50μs interval lengths, respectively.

$(m1, m2, m3)$	$frequency$
(2.1,3.5,1.8)	0.6 GHz
(4,1,4)	1.2 GHz

Table 2: The decision table derived from Table 1.

Recorded training data is then converted into a table. Entries are of the form: *(measurements, frequency, goal metric)*: *measurements* represent metrics characterizing a core's current environment (such as IPC or cache miss rate); *frequency* represents a possible frequency the core might choose after examining the measurements; and the *goal metric* represents the observed metric for energy efficiency resulting from selecting that *frequency*. In this work, we primarily consider "energy per (user-instruction)^2" ($epui2$) to describe energy efficiency, but also experiment using "energy per user-instruction" ($epui$), which has less emphasis on throughput. These are the same metrics used by Herbert and Marculescu in [10] and [9].

During table creation, all metrics (*measurements* and the *goal metric*) are discretized by rounding to a predetermined sensitivity. Discretization ensures that, when we insert values into the table, we may see some duplicate or conflicting entries with the same measurement values. These collisions allow us to keep the table size smaller than the total number of intervals we measured over. In addition, collisions mean that we have multiple data entries for a given set of measurements; this allows us to compare the effectiveness of different frequency decisions for that set of measurements.

Table 1 shows two fictional example table entries, using abstract measurements $m1$, $m2$, and $m3$. The first example indicates that, during training, we have observed $m1 = 2.1$, $m2 = 3.5$, and $m3 = 1.8$ a total of $15 + 12 + 13 + 15 + 11 = 56$ times. These measurements were observed during our random training policy. In the interval following our observation of these measurements, of the 15 times when the frequency was randomly set to 0.6 GHz, the $epui2$ was 4.5 six times and 5.0 nine times. All twelve times 0.8 GHz was randomly chosen, we observed an $epui2$ of 5.0. The remaining values in Table 1 can be similarly interpreted.

To use this table at run time, we first convert it into the decision table shown in Table 2. This table also contains measurements in the left column, but the right column contains only the estimated optimal frequency corresponding to the measurements. We estimate the optimal frequency by calculating the average $epui2$ for each frequency in Table 1, our guess for the optimal is the one with the lowest average $epui2$. In our example, for the measurement (2.1,3.5,1.8), 0.6 GHz has the lowest average $epui2$ (4.8) compared to all other frequencies. During run time, upon observing *measurements* of (2.1,3.5,1.8), we look up the second table for a matching measurement and find the best estimated fre-

most correlated for *epui*	most correlated for *epui*2	chosen for *epui*	chosen for *epui*2
$\frac{\text{user\_inst}}{\text{inst}}$	$\frac{\text{user\_inst}}{\text{L1\_accesses}}$	$\frac{\text{user\_inst}}{\text{inst}}$	$\frac{\text{user\_inst}}{\text{L1\_accesses}}$
$\frac{\text{user\_inst}}{\text{L1\_accesses}}$	$\frac{\text{user\_inst}}{\text{inst}}$	$\frac{\text{user\_inst}}{\text{cycles}}$	$\frac{\text{user\_inst}}{\text{inst}}$
$\frac{\text{user\_inst}}{\text{cycles}}$	$\frac{\text{user\_inst}}{\text{cycles}}$	$\frac{\text{L2\_misses}}{\text{L1\_misses}}$	$\frac{\text{user\_inst}}{\text{cycles}}$
$\frac{\text{L2\_misses}}{\text{L1\_misses}}$	$\frac{1}{\text{L1\_accesses}}$		
$\frac{1}{\text{L1\_accesses} \times \text{L1\_misses}}$	$\frac{1}{\text{user\_inst} \times \text{inst}}$		

Table 3: Results of correlation study.

quency is 0.6 MHz. For measurement (4,1,4), the lowest average *epui*2 (4.0) occurs at 1.2 GHz.

3.2 Choosing Measurement Metrics

Ideally, each unique *measurement* and *frequency* would have only a single observed *epui*2, indicating that the *measurements* and *frequency* perfectly predict the resulting goal metric. If possible, this should hold even when running different workloads, which is why a sufficiently varied training set is desired. Inevitably, there is some noise—where the same set of measurements results in different goal metric values—as illustrated by the measurements for (2.1,3.5,1.8) and 1.2 GHz: eight times out of fifteen when the random training policy chose 1.2 GHz, the resulting *epui*2 was 6.0; but seven times the *epui* was 4.5—lower than the average *epui*2 for 0.6 GHz.

To determine which measurements best minimize noise, we performed a correlation study on the random training data for all sampled intervals. For each goal metric (*epui* and *epui*2), we separated measurement candidates into bins by frequency and calculated the correlation of each measurement with the goal metric. We then compare the correlation for each measurement with the goal metric, averaged across all frequencies.

The set of measurement candidates we checked was generated automatically. First, we considered the raw measurements listed below. Note that all measurements are specific to a single core.

- **cycles**: The number of cycles in an interval.
- **user_inst**: The number of user instructions retired in an interval.
- **inst**: The total number of user and system instructions retired in an interval. This includes **user_inst**.
- **L1_accesses**: The number of L1 cache accesses in an interval.
- **L1_misses**: The number of L1 cache misses in an interval.
- **L2_accesses**: The number of L2 cache accesses (by a single core) in an interval.
- **L2_misses**: The number of L2 cache misses (by a single core) in an interval.
- **L2_stall**: The total time (not cycles) spent stalling by a core due to L2 cache accesses.

The raw measurements were chosen from what we felt to be easily accessible measurements for any architecture's hardware counters. We included both instructions and user instructions because our programs are multithreaded and one thread may spin while waiting for another thread at a barrier or lock, which are counted as instructions but not user instructions.

From these raw measurements, we then add all inverses (e.g. (1/cycles)); then combine all pairs of these multiplicatively. This process generates all commonly-used measurements such as IPC and L1 miss rate (L1 miss per L1 access). For example, (user_inst/cycles) represents the ratio of user instructions to cycles in an interval; and (1/L1_accesses) represents the 1 divided by the number of L1 accesses in an interval.

Once we have generated all measurement candidates, we check each measurement's correlation with the goal metric—these correlations were used to guide our feature selection. Table 3 shows the most correlated metrics and our measurement selection used during evaluations, as well as the measurements we used in our experiments for both *epui* and *epui*2. In addition to considering correlation, we tried to avoid choosing metrics which are very similar with one another. For example (user_inst/inst) and (user_inst/L1_accesses) are very similar. Because (L2_misses/$mboxL1\_misses$) also had high correlation for *epui*, we used that instead of (user_inst/L1_accesses). Other measurements were not as highly correlated for *epui*2, so we elected to keep (user_inst/L1_accesses) as a measurement.

3.3 Decision Tree Creation

While the table provides us with a history of *epui*2 and could guide a DVFS controller, it is too large for practical use at run time. For example, the full table uses 2.6 MB and 500 Bytes for 50μs intervals and 1ms intervals, respectively. It would be both difficult and expensive to store such a structure in hardware. Moreover, if training has never encountered a particular measurement value, then the table will not include an expected *epui*2 for this measurement. Therefore, once a table is populated, we compress it into a decision tree. We utilize the open source WEKA [12] machine learning package to generate our decision trees. The package is written in java and supports a graphical interface. Decision Tree classifiers are just one of WEKA's data mining operations.

Figure 2 shows a sample fictional decision tree. The generated decision tree can be looked up at run time to find the best frequency choice. Starting at the root of the binary tree, each node has a metric and a threshold. If the measurement for that metric falls under the threshold, we take the left branch; otherwise we take the right branch—this process continues until we reach a leaf which contains the frequency we should take. Using the example from Table 1,

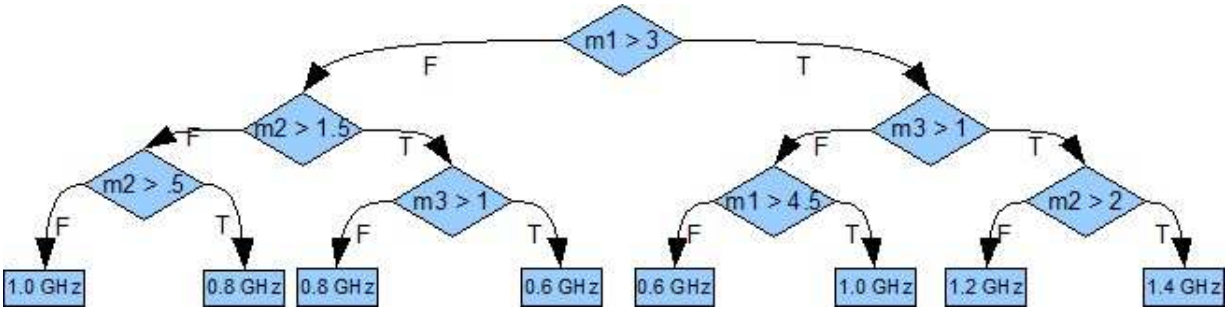


Figure 2: Sample fictional decision tree.

if we observed $m1 = 2.1$, $m2 = 3.5$, and $m3 = 1.8$; we would traverse the tree left, then right, then right—upon reaching the leaf node, we see we should set the frequency to 0.6 GHz. Note that for some paths (such as the leftmost), the same metric ($m2$) may be used more than once. This allows more relations more complex than a single binary split to be expressed. At each node, the decision tree algorithm splits on the metric best able to categorize the values—in this case the data indicated that $m2$ was more appropriate than $m3$ or $m1$ at this node.

Note that this tree has no blank entries—any set of measurements will arrive at a leaf. With the pure table, a measurement that was similar to, but not an exact match to an existing entry, would have missed in the table. With the decision tree this is not the case. An important effect of being a complete function is that the decision tree is now robust enough to deal with unfamiliar workloads. So long as the decision tree is representative of the best frequency decisions, it should still choose good frequencies for unfamiliar workloads.

Machine Learning via WEKA

To use WEKA, we first convert our table into a compatible format, Attribute-Relation File Format (arff). An arff file requires elements be formed of a set of keys and a single learned target value, which is slightly different from the table structure shown in Table 1 (it has both *frequency* and *epui*2 as values). We instead use the table format in Table 2, so we have one value—the predicted optimal frequency. Some measurements in the table are more likely than others and should have higher priority. Because we kept a count of table entries earlier, this is easily handled—each entry is replicated proportionally to its occurrence likelihood.

Once data is placed in an arff file, we run this through a machine learning classifier, which learns a function mapping the keys to values—in other words, it maps measurements from hardware counters into optimal frequencies. From a high level perspective, the decision tree is created such that each node partitions the data over a metric. The chosen metric should partition data into groups where members of one group have values as dissimilar as possible from members of the other group. We chose the REPTree decision tree, because it allows us to set the maximum tree depth. The example tree in Fig. 2 has a maximum depth of 3. We evaluate with tree depths limited to 3, 5, and 7.

Consider the example in Table 2 once again. The expected *epui*2 is lowest for 0.6 GHz, so we insert entries of the form "$2.1, 3.5, 1.8, 0.6$" into the arff file. Because there were 15 total occurrences for 0.6 GHz, we replicate that example 15 times in the file. If replication causes the file to be too large, we can approximate by replicating each value N/k times, where N is the number of occurrences, and k is some constant greater than 1.

Another property of the REPTree is that it can treat the learned value (the optimal frequency) as either a real value or a nominal value. With nominal frequencies, the learner only recognizes the five frequencies, but has no notion of their numerical nature (that "800" falls between "600" and "1000"). Using real values, on the other hand, captures this numerical nature, but the learner believes there are more frequencies than is the case; for example, "744" could show up in the decision tree. In this case, we would round to 800 MHz. We evaluate both approaches in the Results section.

3.4 Hardware Implementation

While our policy could be implemented in software, some minor hardware structures could greatly reduce switching time. A hardware-implemented decision tree can be looked up quickly by comparing a node's metric to its threshold until a leaf is reached—requiring max_depth comparisons.

Figure 3 shows the example decision tree from Fig. 2 as a hardware structure. We first map tree nodes to elements in a hardware array by assigning nodes addresses in order of a depth first traversal. Starting with the root node, we look up the array and use the metric type to drive a multiplexor that selects between three hardware counters that store the measurements ($m1$,$m2$,$m3$) for the interval. The value in the selected counter, as well as the threshold value from the array node, are sent to a comparator which drives the address generator for the next array lookup. This process iterates a number of times equal to the depth of the decision tree until we can look up a leaf node for the desired frequency. Leaf nodes are stored in a separate structure because they require fewer bits to store.

Space Overhead

We calculate the space overhead for our decision tree. Each of the $2^{depth} - 1$ nodes would need to store: the metric it branches on $\big(lg(\textit{the number of metrics}) = 2 \text{ bits}\big)$, and the threshold value. Full floating-point precision is not required for the threshold values, since the table holds already-rounded values due to discretization. By counting the number of unique table entries, we can bound the number of bits necessary to express the full range of thresholds for any single metric. For the rounding sensitivities we used, no more than 15 bits would be necessary for each threshold value.

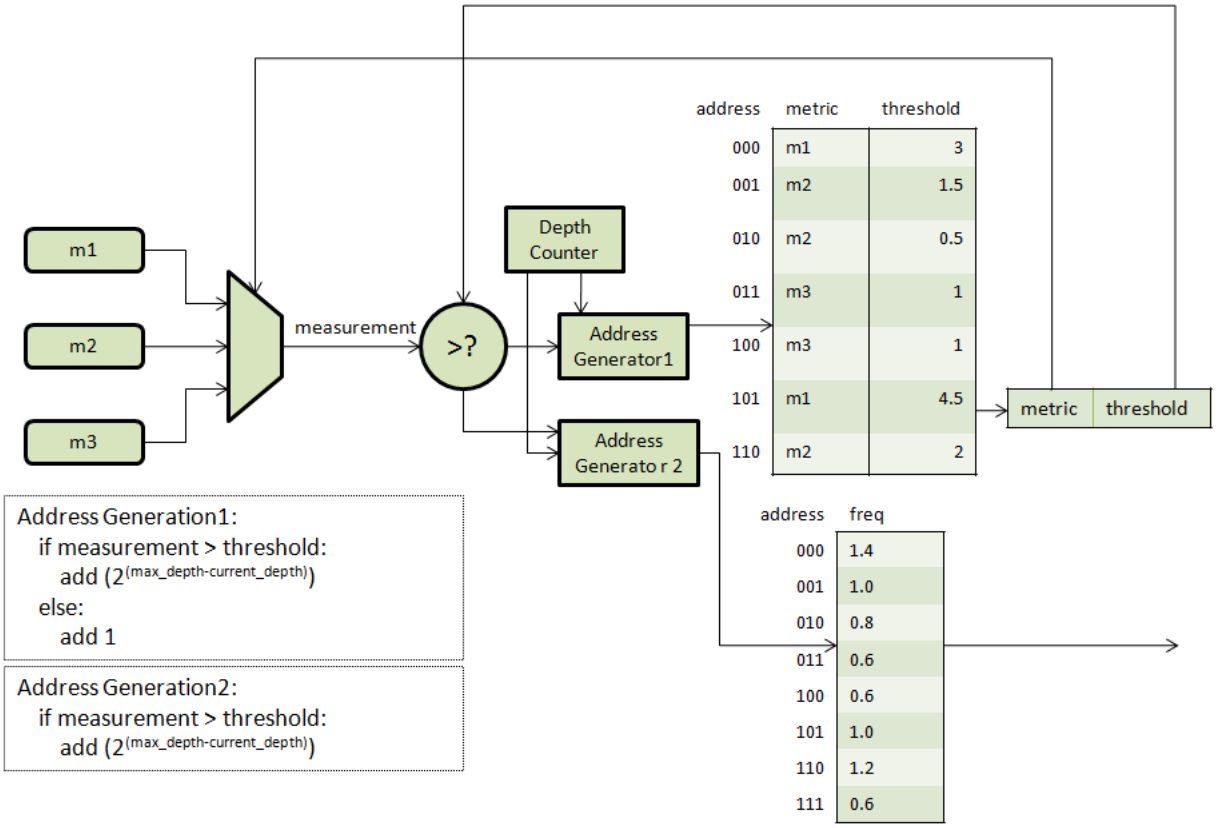


Figure 3: Hardware Implementation for decision tree lookup based on the example in Fig. 2. *max_depth* **for this tree is 3,** *current_depth* **starts at 0 and is stored in the depth counter. Address Generator 1 is used when looking up measurement metrics or branch nodes in the decision tree (when** $current\_depth < max\_depth$**), and Address Generator 2 is used to look up a frequency at a leaf node (when** $current\_depth = max\_depth$**).**

The 2^{depth} leaves would require lg(*the number of frequency settings*) bits each; 3 for our simulated machine. Table 4 shows estimated overheads for decision trees of depth 3, 5, and 7—access time and energy numbers were calculated using CACTI [15].

4. EXPERIMENTAL SETUP

We validate our policy via simulation with Virtutech's Simics [17] full-system simulator running Ubuntu 8.10 Linux. Machine parameters are shown in Table 5. We use an in-order 16-core machine with 5 frequency settings per core. For timing, we model a cache hierarchy with private L1 caches, a shared L2 cache, and memory with a fixed access time.

An L1 cache's access time scales with its home core's frequency and is always 1 cycle. An I-cache lookup is assumed to be included on the critical path and has a 0-cycle hit latency. The L2 cache has a fixed access time. In addition, the L2 cache models contention by having a fixed number of ports. A port is busy and cannot accept other requests for the on-chip portion of an access—it is freed if the request misses and goes to memory. An L2 request uses an available port if possible, or the port that will be freed the earliest.

Power consumption is modeled as having dynamic, leakage, and background components—dynamic power scales cubically with frequency [11], leakage power scales linearly [3] [11], and background power—representing the L2 cache and the rest of the system—is not in the same clock domain and consumes a constant amount of power. Note that we assume voltage scales linearly with frequency. In our model, at 1 GHz, leakage power consumption is double that of dynamic power—which is consistent with estimates from Kim et al. [11]. Background power is computed assuming that a loaded CPU at 1 GHz consumes 30% of the total system's power (for example, a processor that consumer 55W in a system that draws 170W). Note that we consider total *system* power in this work; neglecting system power may degrade overall energy efficiency even if it improves energy efficiency for the processor. However, because the chip consumes only a portion of the power in the system, total possible improvement is limited.

Our workloads are mostly multiprogrammed, multithreaded combinations of PARSEC benchmarks [2]. For each workload, all benchmarks start and fast forward until each reaches its Region of Interest. That benchmark is then artificially halted by lowering the clock frequency of any core running that benchmark while other benchmarks finish their initialization. Once all benchmarks have reached their region of interest, clock frequencies are restored, we warmup for 1 million cycles. The workload is then run for 2s: 2,000, 8,000, or 40,000 intervals for 1ms, 250μs or 50μs interval lengths, respectively. No benchmarks finish prematurely.

Machine Parameters	Values
Processor	16-core, x86, in-order, single-issue
Frequencies	0.6 GHz, 0.8 GHz, 1.0 GHz, 1.2 GHz, 1.4 GHz
L1 I Cache	32kB per core, 2-way, 16-byte line size, LRU
L1 D Cache	32kB per core, 2-way, 16-byte line size, LRU Snoopy Coherence
L2 Shared Cache	8MB, 16-way, 64-byte line size, LRU, 4 ports 13ns access time (13 cycles at 1 GHz)
Memory	150ns access time (100 cycles at 1 GHz)

Table 5: Machine Parameters.

Benchmark	Symbol
Blackscholes	bl
Bodytrack	bo
Ferret	fe
Freqmine	fr
Streamcluster	st
Swaptions	sw
Vips	vi
x264	x

(a) Benchmark Notation

A1	bl4l+bl4l+bl4l+bl4l
A2	bl4l+bl4l+bl4l+fe4l
A3	bl4l+bl4l+fe4l+fe4l
A4	bl4l+fe4l+fe4l+fe4l
A5	fe4l+fe4l+fe4l+fe4l
B1	st4l+st4l+st4l+st4l
B2	st4l+st4l+st4l+sw4l
B3	st4l+st4l+sw4l+sw4l
B4	st4l+sw4l+sw4l+sw4l
B5	sw4l+sw4l+sw4l+sw4l
C1	bo2l+fe4l+fr2l+sw2l+x4m
C2	fr2l+st4l+sw2l+vi2m+x4m

(b) Workload Composition

Table 6: 6(a) lists the Benchmarks used from the PARSEC [2] Suite. 6(b) lists the benchmark combinations that make up our workloads—symbols correspond to benchmarks in 6(a), followed by the thread count and input size: 'm' for simmedium or 'l' for simlarge.

We have three groups of workloads, which we denote as sets A, B, and C. Sets A and B serve as our training workload groups, but are also used in testing via cross-validation [14], while set C is only used for testing. In other words, training data from set A is used to test set B and vice versa. Set C is never used for training and is tested using data generated from A or B.

Sets A and B each contain two PARSEC benchmarks, one which scales well with higher frequency and one which does not scale as well. We run them in 4 multiprogrammed groups of 4 threads. Each set has all combinations of the two benchmarks (4 of benchmark 1 and 0 of benchmark 2),(3 of benchmark 1 and 1 of benchmark 2), etc. Each set thus shows a spectrum of varying behavior. Set C is composed of two workloads, each composed of 5 multiprogrammed PARSEC benchmarks.

Tables 6(a) lists the PARSEC benchmarks used, and 6(b) describes the workload combinations, including the input size for each benchmark and the number of cores assigned to it. For example, workload A2 (bl4l+bl4l+bl4l+fe4l) is composed of: 4 threads of Blackscholes, 4 threads of Blackscholes, 4 threads of Blackscholes, and 4 threads of Ferret; all using the *simlarge* input set.

Decision Tree Depth	Space (bits)	Overhead Relative to 64kB L1	Access Time (ns)	Access Energy (nJ)
3	143	0.02%	0.27×4	0.0051×4
5	623	0.12%	0.27×6	0.0051×6
7	2543	0.49%	0.29×8	0.0054×8

Table 4: Hardware Space Overhead. Access time and access energy computed using CACTI [15].

We evaluate our proposed lookup-based scheme against the machine's baseline, no-DVFS, fixed frequency—1.4 GHz. Note that, as shown in Fig. 1, our baseline is sometimes more efficient and sometimes less efficient than other fixed frequency settings. No fixed frequency performs best all the time. We show results for the table lookup described in Sect. 3.1. While the table scheme is not realistic, it serves to show the effectiveness of decision tree compression.

Our scheme is also compared to Herbert and Marculescu's adapted Greedy policy described in [9]. In that scheme, the DVFS controller either holds its frequency or explores new frequencies one step at a time. During exploration, each VFI estimates its energy consumption and the number of retired user instructions. The $epui2$ is calculated, then compared against a saved value from the previous interval. If energy efficiency improves, the controller continues in the same direction in the next interval. If energy efficiency degrades, the controller assumes the previous frequency was optimal; reverses its direction; transitions to that frequency; and holds at that frequency for $N = 5$ intervals. In addition, the policy implements the improvement described in [10]—the controller issues a hold if exploration would take it below the minimum frequency or above the maximum frequency.

Dynamic DVFS policies use 1ms, 250μs, and 50μs intervals for deciding new policies. Our simulations assume zero switching overhead for dynamic policies. A single PLL, shared by all cores and combined with a clock divider, can enable single-cycle frequency transitions [6]. With on-chip voltage regulators [8] and fast hardware-supported decision tree lookup, switching overhead can be kept under 10ns, which is 0.02% of a 50μs interval.

5. RESULTS

Results showing *epui*2 are presented in Figs. 4 and 5; lower is better. Shown are: the baseline policy running at 1.4 GHz, *greedy*, a raw table lookup with no decision tree compression, and decision trees of depth 3. For table and decision tree policies, we use a form of cross validation [14]. For set A, the trees are generated using tables from set B's training run, and vice versa. Our scheme shows an average *epui*2 improvement of 14.4% over the baseline and 11.3% over *greedy*.

Greedy does not perform very well in our experiments. The algorithm depends on program phases remaining stable for a certain period of time; namely the time it takes to explore (between 1 and 4 intervals) and find the best frequency, plus the hold time (for our experiments, 5 intervals) at that frequency. Shorter phases will cause the *epui* to change during exploration and force a premature hold. Longer phases trigger exploration every $(N+1)^{th}$ interval, meaning even during a stable period, *greedy* uses a suboptimal frequency 16.7% of the time for N=5.

It is interesting to note that the decision tree significantly outperforms the table. There are two major factors here—first, the decision tree lacks empty entries and can guide DVFS policies if measurements have not been encountered in the past. Second, each decision tree choice is influenced not only by measurements in the table that match the current measurements; but by all nearby measurements. This is because the decision tree algorithm clumps elements together; thus providing a smoother function than the raw table.

Figures 6 and 7 show the same trends hold when *epui* is used as the goal metric. The table and decision trees were built choosing the frequency with the lowest expected *epui*, and *greedy* optimizes *epui*. Average *epui* improvements of our decision tree policy are 15.3% over the baseline and 6.2% over *greedy*.

Partial Training

The decision tree built using set A performed well on set B; and set B's tree performed well on set A. Figure 8 shows the *epui*2 performance of both trees on set C for direct comparison. The table is not evaluated for these workloads because using table lookups for an unfamiliar workload causes many misses and very poor performance.

Trees trained with set A perform nearly the same as those trained with set B; both show a 9.3% and 8.5% improvement over the baseline and *greedy*, respectively. This shows us that the measurements reflect the execution environment and are

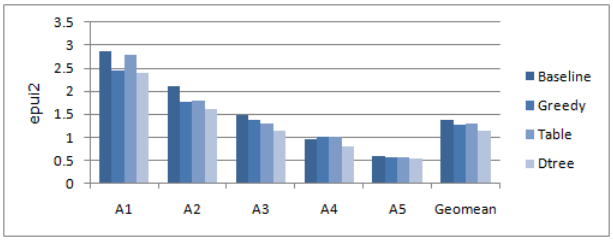


Figure 4: *epui*2 performance for workload set A, geometric mean on the right. Policies shown: baseline, greedy, table, and decision tree. Dynamic policies use 1ms interval lengths.

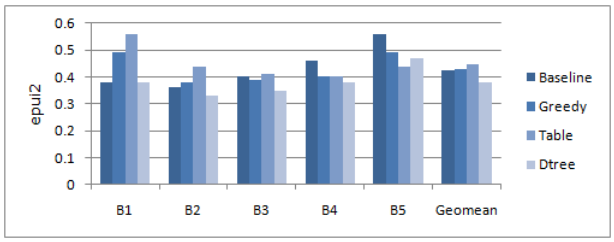


Figure 5: *epui*2 performance for workload set B, geometric mean on the right. Policies shown: baseline, greedy, table, and decision tree. Dynamic policies use 1ms interval lengths.

relatively independent of the specific workload executing. Further results using set C present the average result from training using set A and set B.

Effect of Interval Length

Figure 9 shows how our scheme and greedy scale with interval lengths. The graph shows the average *epui*2 for all workloads in sets A, B, and C. The decision tree does not scale well with shorter intervals. This is most likely because shorter intervals have more noise than larger intervals.

Effect of Tree Depth

Figure 10 shows the effect of deeper trees for our scheme. The maximum depth of the tree has no noticeable detrimental effect on performance; in fact, smaller trees sometimes outperform larger ones. This is due to *overfitting* [14], because specific examples in the training data do not necessarily match test samples perfectly. This also implies the ideal function for frequency settings is not too complex; a handful of parameters can approximate the function we not too complex; a handful of parameters can approximate the function we are learning.

Nominal vs Real Frequency

Knowledge of frequency's numerical properties is beneficial for our learner, as Fig. 11 demonstrates. Treating frequency as a real outperforms treating frequency nominally in almost all cases, despite the two using the same basic algorithm. This is because a misclassification error for the nominal tree is less likely to be close to the correct frequency; while the real-valued tree will try to cluster similar values together.

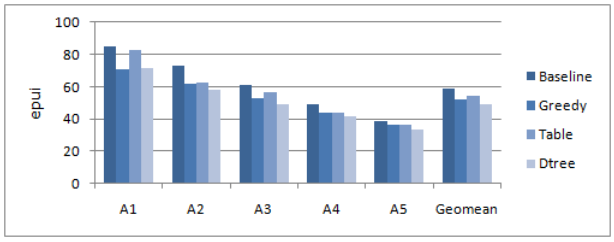


Figure 6: *epui* performance for workload set A, using training data from set B. Geometric mean on the right. Policies shown: baseline, greedy, table, and decision tree. Dynamic policies use 1ms interval lengths.

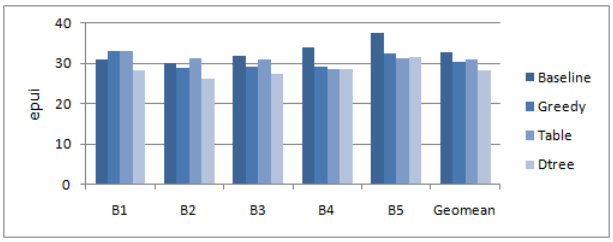


Figure 7: *epui* **performance for workload set B, using training data from set A. Geometric mean on the right. Policies shown: baseline, greedy, table, and decision tree. Dynamic policies use 1ms interval lengths.**

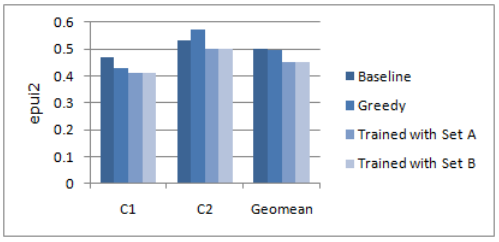
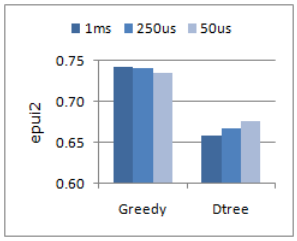
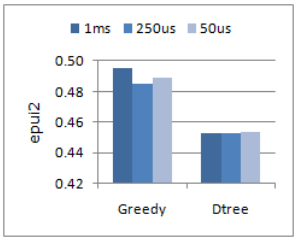


Figure 8: *epui*2 **performance for workload set C, comparing training data from sets A and B. Geometric mean on the right. Policies shown: baseline, greedy, and decision tree. Dynamic policies use 1ms interval lengths.**

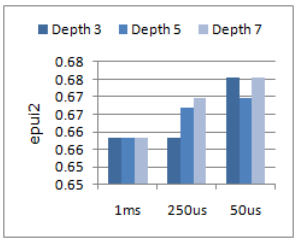


(a) Sets A and B

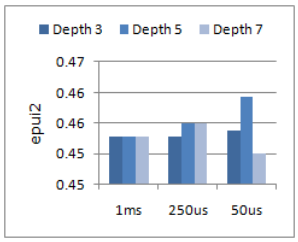


(b) Set C

Figure 9: Geometric mean of *epui*2 **for sets A, B, and C at varying interval lengths. Policies shown: greedy and decision tree. The decision trees are of depth 3.**



(a) Sets A and B



(b) Set C

Figure 10: Geometric mean of *epui*2 **for sets A, B and C. The decision tree policy is used with trees limited to varying depths and with varying interval lengths.**

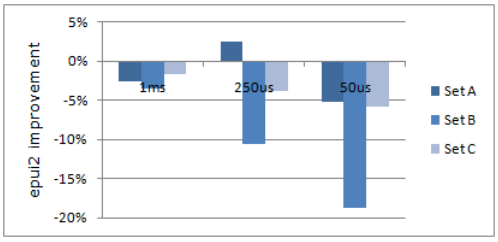


Figure 11: Results for treating frequency as nominal values during decision tree creation, performance is relative to using real frequency values (higher is better). Policies shown: decision trees of depth 3 at varying interval lengths.

6. CONCLUSION

We presented a DVFS policy which learns the best frequency settings for a CMP with per-core voltage/frequency scaling with minimal training required. Correlation drives our feature selection process in an effort to minimize noise in our training data. Following a set of random training runs, a table can be built mapping measurements to the best expected frequency. This learned table is then converted into a a decision tree—a tree structure can be cheaply implemented in hardware for fast lookups. Even though our scheme trains on different workloads than it is tested on, it showed improvement over the heuristic *greedy* policy.

We showed the effect of various choices made when building our decision tree. Regardless of which training set we used, the decision trees work equally well for new, unfamiliar workloads. Allowing the decision tree algorithm to treat frequency as a real value helps it cluster similar frequencies together in a single leaf if it cannot perfectly partition the training data. The table building process allows a user to specify any goal metric, and we showed that our scheme is flexible enough to use either *epui* or *epui*2 as the target.

7. REFERENCES

[1] Nevine AbouGhazaleh, Bruce R. Childers, Daniel Mossé, and Rami G. Melhem. Integrated cpu cache power management in multiple clock domain processors. In *HiPEAC*, pages 209–223, 2008.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[3] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *MICRO*, pages 191–201, 2000.

[4] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *ISLPED*, pages 207–212, 2007.

[5] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *ISSCC*, pages 102–103, Feb. 2007.

[6] T. Fischer, J. Desai, B. Doyle, S. Naffziger, and B. Patella. A 90-nm variable frequency clock system for a power-managed itanium architecture processor. volume 41, pages 218–228, Jan. 2006.

[7] Vincent W. Freeh, Feng Pan, Nandini Kappiah, David K. Lowenthal, and Robert Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *IPDPS*, 2005.

[8] P. Hazucha, T. Karnik, B.A. Bloechel, C. Parsons, D. Finan, and S. Borkar. Area-efficient linear regulator with ultra-fast load regulation. volume 40, pages 933–940, April 2005.

[9] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISLPED*, pages 38–43, 2007.

[10] Sebastian Herbert and Diana Marculescu. Variation-aware dynamic voltage/frequency scaling. In *HPCA*, pages 301–312, 2009.

[11] Nam Sung Kim, Todd M. Austin, David Blaauw, Trevor N. Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut T. Kandemir, and Narayanan Vijaykrishnan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003.

[12] Veronica Liesaputra and Ian H. Witten. Seeking information in realistic books: a user study. In *JCDL*, pages 29–38, 2008.

[13] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steve Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA*, pages 14–25, 2003.

[14] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, international edition, 1997.

[15] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. volume 28, pages 69–79, 2008.

[16] Jun Shirako, Naoto Oshiyama, Yasutaka Wada, Hiroaki Shikano, Keiji Kimura, and Hironori Kasahara. Compiler control power saving scheme for multi core processors. In *LCPC*, pages 362–376, 2005.

[17] Simics. http://www.simics.net/.

[18] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *ISCA*, pages 363–374, 2008.