Integrated CPU and L2 cache Frequency/Voltage Scaling using Supervised Learning

Cosmin Rusu, Nevine AbouGhazaleh, Alexandre Ferreira, Ruibin Xu, Bruce Childers, Rami Melhem, and Daniel Mossé
{rusu, nevine, apf75, xruibin, childers, melhem, mosse}@cs.pitt.edu

Department of Computer Science, University of Pittsburgh

Abstract. Multiple clock domain (MCD) chip design addresses the problem of the increasing clock skew in the different chip units. MCD design opens the opportunity for independent power management in each domain when used in conjunction with dynamic voltage scaling (DVS). A significant power and energy improvement has been shown for finer control of each domain voltage rather than managing the chips single voltage, as in traditional chips with global DVS. However, published policies in the literature focus on each domain in isolation without considering the possible inter-domain effects when varying their clock/voltage from other domain.
In this paper we propose to use a supervised machine learning technique to automatically derive an integrated CPU-core and on-chip L2-cache DVS policy. Our policy relies on simple performance counters that can be easily monitored. We discuss the machine learning process and the implementation issues associated with our technique. We show that our derived policy improves on traditional power management techniques used in MCD chips. Our technique saves up to 34% (10% on average) over a DVS techniques that apply independent DVS decisions in each domain. Moreover, energy and energy-delay product results are within 3% of a near-optimal scheme.

1 Introduction

Dynamic Voltage Scaling (DVS) is a technique that can be used to reduce power consumption in CMOS digital circuits. A lower frequency of operation gives the possibility that a lower supply voltage can be applied. A convex relationship holds between frequency and power consumption for specific types of circuits and thus a small decrease of frequency/voltage can have a substantial impact on energy [14].

Due to the continuous increase in the number of transistors and lower feature size, higher chip densities create a problem for clock synchronization among different chip computational units. An effective solution to this problem is the use of design techniques for multiple clock domains (MCD) chips. In MCD, a processor chip is divided into multiple domains. Each domain operates synchronously with its own clock, and communicates with other domains asynchronously through FIFO queues. MCD design allows for fine grain power management of each domain especially when using dynamic voltage and frequency scaling (DVS). Since each domain has its own clock and voltage (i.e., independent of the other domains), DVS can be applied in each domain for an extra level of power management (rather than applying DVS at the chip level). Power and energy can be reduced with minimal impact on performance by dynamically reducing the clock speed and voltage in domains with low activity.

Several power management policies have been proposed to incorporate DVS into MCD chips. The published results show a significant power and energy improvement over applying DVS to a fully synchronized chip (i.e., with a single master clock) [7]. However, these policies focus on each domain in isolation without considering the possible effect of varying one domain's clock speed and voltage on other domains. Moreover, existing techniques rely on online heuristics.

In this work, we are interested in minimizing the overall energy-delay product in a processor. We are especially interested in the CPU-core and the on-chip L2-cache, as they consume a large fraction of the total power in current processors. In this paper, we propose a novel methodology to derive an integrated CPU-core and L2-cache DVS policy. The integrated policy identifies application phases at runtime and takes corresponding actions (i.e., setting the voltage and frequency of both the processor and the L2-cache). The policy is derived with a supervised learning process on a representative training workload. We present and evaluate a policy that optimizes for either energy or energy-delay product of the entire processor (including the core and caches).

The rest of the paper is organized as follows. We briefly discuss related work in Section 2. Our problem description is given in Section 3. We describe the supervised learning technique we use to determine an integrated CPU-core and L2-cache DVS policy in Section 4, followed by evaluation in Section 5. Finally, we conclude the paper and discuss future work in Section 6.

2 Related Work

DVS was extensively explored for a variety of systems (from embedded devices to server farms) and application areas. For embedded systems, DVS techniques save energy by lowering the voltage and frequency for just-in-time completion of real-time applications [9, 3, 14]. For personal computers running Linux, DVS is used to lower the energy consumption while maintaining performance requirements of applications and good responsiveness of interactive jobs [5]. For web servers, utility-based DVS schemes adapt the frequency and voltage according to the incoming load [1]. In server clusters, DVS is used as a local power management scheme aware of Quality-of-Service constraints [11]

Multiple clock domains (MCD) are proposed as a fine grain processor DVS mechanism in [7]. Magklis et al. propose an online power management policy that monitors queue occupancy of a domain and adapts the domain's voltage accordingly [8]. For each domain, the policy computes the change in the average queue length among consecutive intervals. When queue length increases, the voltage and clock speed are increased. Similarly, when queue length decreases, the voltage and clock speed are decreased. However, this policy does not take into account the cascading effects of changing a domain voltage on other domains. Another technique by Magklis et al. uses a profile-based approach to identify program regions that justify reconfiguration [7]. This approach incurs extra overhead due to profiling and analysis phases for each application under consideration. In contrast, our technique learns the DVS policy with training samples and can be directly applied to new applications without profiling. Zhu et al present architectural optimizations for improving power and reducing complexity [17]. Voltage scaling of off-chip L2 caches for embedded systems is studied in [10].

Sherwood et al. showed that programs have repeatable phase-based run-time behavior over many hardware metrics, such as cache behavior or branch prediction [13]. The authors also provide a tool, called SimPoint, that automatically identifies and clusters the phases in a program in order to speed up architectural simulations [12]. Application phases and predictable behavior are essential to our work as well.

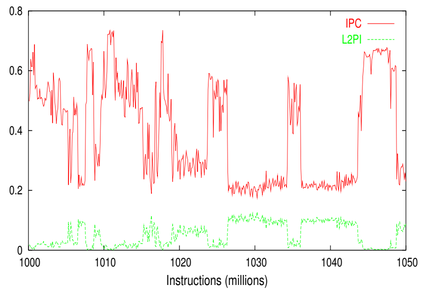
Applying machine learning techniques to reconfigure architectural and compiler settings is relatively a newly explored field. Wildstrom et al. present a policy to alter server configuration in reaction to workloads [15]. The policy learns to identify preferable CPU and memory configurations. They showed significant performance benefits using machine learning policy over any fixed configuration. Cavazos et al. use supervised learning to predict which application's basic blocks can benefit from scheduling [2]. The learned policy selects whether to schedule a block or not. The policy achieves most of the potential performance improvement with significantly less overhead.

3 Problem Description

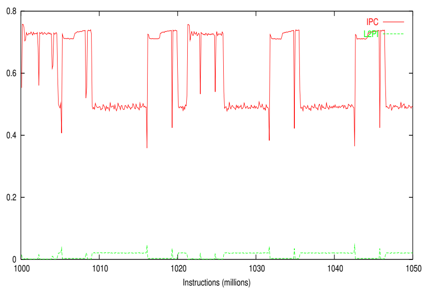
A typical application goes through phases throughout its execution. An application has varying cache/memory access patterns and CPU stall patterns. In general, application phases correspond to loops, and a new phase is entered when control branches to a different code section. Since we are interested in the performance and energy of the CPU-core and L2-cache, we characterize each code segment in a program with two metrics: cycles per instruction (CPI) and number of L2 accesses per instruction (L2PI). CPI and L2PI are selected as indications of the amount of workload in the CPU-core and L2-cache, respectively. Examples of CPI and L2PI showing different program phases can be seen in Figure 1 for two benchmarks: *gcc* and *gzip* (from the SPEC2000 benchmark suite).

Intuitively, each program phase has a different requirement and preference toward a certain "configuration" of the CPU-core and L2-cache frequencies. For example, if a section of code is CPU bound, it will benefit from running at high CPU frequencies, and may be insensitive to L2-cache latency. On the other hand, a memory bound phase benefits the most from reducing the gap between the core and cache performance. This is precisely the intuition behind our approach. Our goal is to construct an integrated CPU-core and L2-cache DVS policy that identifies application phases and selects good frequencies for the CPU-core and L2-cache domains for each section of code.

Clearly, the L2-cache and CPU frequencies can be set independently based on activity represented by CPI and L2PI. Thus, we need to answer the following questions about an integrated policy: (a) Is an integrated CPU-core and L2-cache DVS scheme better than an independent scheme? (b) What are the mechanisms to be adopted with respect to these options? (c) What are the frequencies/voltages to be chosen at each program phase? We attempt to answer these questions throughout this paper.



(a) *gcc* phases



(b) *gzip* phases

Fig. 1. Application phases variation throughout execution.

One approach to building an integrated DVS policy is inspired by control systems. A phase change can be easily identified from simple performance counters. For example, a decrease in the CPI (after filtering noise) may suggest that a higher CPU frequency is needed, or that a higher cache latency is tolerable. A stable system phase is defined as a small variation (within a threshold) around the average CPI. The goal is then to change (i.e., increase/decrease) the CPU and/or L2-cache frequencies when a phase shift is identified.

The problem with a control approach is not identifying application phases, but selecting the correct frequencies on phase changes. The problem is that we can identify the correct action *towards* the optimal configuration, but not the optimal configuration itself. Using performance counters, we could decide, for example, whether the frequency of the CPU should be increased or decreased, but not the exact amount. This is because phase changes are not gradual, but *instantaneous*, corresponding to a jump to a different section of code. As soon as the jump is taken and the code enters a new phase (with stable CPI) there is no more feedback regarding how good the frequency change actually was, and if it was just a step in the right direction. Typically, there is little correlation between the amount of variation of some performance metric (such as CPI) and the right frequency. Furthermore, for more complex metrics such as energy or energy-delay product, even the step towards the correct action (i.e., increase or decrease frequency by one level) is hard to identify, as it is not trivial to estimate how energy consumption relates to the performance counters.

4 An Integrated CPU-core and L2-cache DVS policy

Because control-based approaches can fail to identify a good policy for integrated CPU-core and L2-cache DVS, we propose an approach based on a supervised machine learning. Our technique derives a policy expressed in the form of propositional rules for a particular system by analyzing a training program workload. For a given architecture, our approach analyzes the system to derive a DVS policy for both the CPU-core and L2-cache to optimize the energy-delay product. The approach describes the *state* of the system under different program behaviors and run-time system characteristics. A program behavior description captures the instruction level parallelism and cache demands of the application and a run-time characteristics description captures program latencies during a given program phase. The goal is to identify for each possible system state the correct *action*. An action determines how the CPU-core and L2-cache frequencies should be adjusted to minimize energy-delay product. The derived policy is thus a function that maps states to actions that take into account the effect on the energy and delay.

We first describe the methodology to obtain the training data used to learn the policy and then our learning approach.

4.1 Obtaining Training Data

It is our hypothesis that for a relatively simple (single issue) processor the system state that encapsulates the program behavior can be described by simple performance metrics. These metrics are the CPI and L2PI, which can be determined from hardware performance counters. The CPI indicates the CPU utilization; however, it does not by itself fully describe program phases. For example, a high CPI corresponds either to a high cache miss ratio, a high cache access latency, or long instruction latencies (such as division). Adding the L2PI into the state description eliminates the confusion and more fully describes application behavior. However, the L2PI does not take into account the effective latency of cache accesses, and to fully characterize the program, this latency has to be factored into the state description. We describe the effective access latency as a tuple of CPU-core and L2-cache frequencies. This representation of cache access latency provides similar information to the effective cache access latency but it also captures the energy, as energy cost is closely related to the operating frequencies.

Thus, a state is described by four parameters: CPI, L2PI, CPU-core frequency and L2-cache frequency. CPI and L2PI are continuous variables and need to be discretized. We choose a number of intervals (discretization bins) for both CPI and L2PI in such a way that the samples in the training data are distributed evenly. For example, because of the L2-cache efficiencies in current designs, if most samples have low L2PI, this would consequently create more L2PI ranges with lower values (i.e., finer granularity where the density is higher). Let K and L be the number of discrete values of the CPI and L2PI, respectively. Let M be the number of available CPU frequencies and N be the number of cache frequencies. The state is a table $State[CPI_k][L2PI_l][i][j]$, where CPI_k and $L2PI_l$ are the discretized values of CPI and L2PI ($0 \leq k < K$ and $0 \leq l < L$), respectively. i and j are the CPU-core and cache frequencies ($0 \leq i < M$ and $0 \leq j < N$), respectively. For each state we want to determine the action that minimizes energy-delay product.

The training data used to learn the policy is obtained from training benchmarks in the following manner. We run all training code at all CPU/cache frequency combinations (MN combinations). A sample is defined as a continuous sequence of code of fixed number of instructions equal to $size$. Thus, a set of training benchmarks with a total of $inst$ instructions and $size$ instructions will generate $C = inst/size$ code samples for one particular CPU/cache frequency, and MNC samples for all frequency combinations. We denote the samples by $S_{ij}^c = \{CPI_{ij}^c, L2PI_{ij}^c, ED_{ij}^c\}$, where c represents the code sample ($0 \leq c < C$). Each sample contains three values: CPI_{ij}^c, $L2PI_{ij}^c$, and ED_{ij}^c, namely, the discretized CPI, discretized L2PI, and energy-delay product of the sample while running at frequencies i and j.

After collecting these values for all samples, S_{ij}^c, the correct action for each state is determined as follows. Since for each section of code all the possible frequency combinations are available, the best action can be determined by adding the energy-delay product of each sample running at the new frequency. Since different sections of code may have the same state, an array that accumulates all values for the same state are used: $Cum[CPI_k][L2PI_l][i][j][x][y]$, where CPI_k, $L2PI_l$, i, and j are the state parameters and x and y are the new CPU and cache frequencies (that is, the action). For each training sample S_{ij}^c and each possible action x, y (x is the next CPU frequency, y is the next cache frequency), we update the arrays as follows:

$$Cum[CPI_{ij}^c][L2PI_{ij}^c][i][j][x][y] += ED_{xy}^c \qquad (1)$$

Equation (1) accumulates the energy-delay product for all training samples and all possible actions. After updating the counters for all samples, the action for each state is the one that minimizes the actions. After updating the counters for all samples, the action for each state, $State[CPI_k][L2PI_l][i][j]$, is the frequencies $\langle x, y \rangle$ that minimizes $Cum[CPI_k][L2PI_l][i][j][x][y]$.

4.2 Learning DVS Policy

With the training data, we can use supervised learning to derive the DVS policy. There are many supervised learning techniques, including logistic classification, neural network, decision tree, and propositional rule. We prefer the propositional rule approach because it is more compact, more expressive, and more human readable than the other techniques. Furthermore, propositional rules are easy to implement in hardware. In fact, we tried all the aforementioned techniques on the training data and the propositional rule approach had the least error.

We use the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) learner [4]. The RIPPER algorithm is known to achieve low error rates while being efficient on large data sets. RIPPER represents the collected states in the form of prepositional (if-then) rules. Each rule specifies the desirable CPU frequency and cache frequency for the next program interval based on the current state. The learner is based on the Incremental Reduced Error learning IREP algorithm [6]. RIPPER repeatedly calls IREP to construct the rule set with low error rates.

IREP iteratively builds its rule set in a greedy fashion; one rule at a time. IREP works in two phases: growing and pruning phases. First, it randomly partitions the data set in to two subsets: growing and pruning sets. The rule growth phase constructs an initial rule set. It starts with an empty clause and then repeatedly adds sub-conditions to the antecedent. The sub-conditions maximize the coverage of the rule (represents more states). The stopping criterion for adding sub-conditions is either covering all the input states or not being able to improve the rule coverage. After growing a rule, the rule is immediately pruned in the pruning phase. Pruning is an attempt to prevent the rules from being too specific. IREP chooses the candidate literals for pruning based on a score which is applied to all the sub-conditions of the antecedent and evaluate the score using the pruning data. This process is repeated until all states are covered or the learned rules have very small error.

The resulting rules are generated in the form of: IF <*condition*> THEN <*set freq*>, where *condition* is a conjunction of one or more of the following sub-conditions. ($CPI_{cur} \leq CPI_k$), ($CPI_{cur} \geq CPI_k$), ($L2PI_{cur} \leq L2PI_l$), ($L2PI_{cur} \geq L2PI_l$), ($c_f = i$), and ($m_f = j$) where CPI_{cur}, $L2PI_{cur}$, c_f and m_f are the current CPI, L2PU, CPU frequency and and cache frequency, respectively. *set freq* specifies the value of the next CPU or cache frequencies.

5 Evaluation

In this section, we evaluate the effectiveness of an integrated CPU-core and L2-cache DVS policy derived with the supervised learning technique from Section 4. We compare the derived policy to (a) a local clairvoyant solution, which is near optimal for the energy-delay metric and (b) an independent CPU-core and L2-cache DVS policy [8].

5.1 Experimental Setup

We use the Simplescalar and Wattch architectural simulators with an MCD processor extension [17]. The MCD extension by Zhu et al. models inter-domain synchronization events and voltage scaling overheads. We alter the design in [17] to merge their individual core domains into a single domain and to separate the L2-cache into its own domain. The simulated frequencies for both domains vary from 250MHz to 1GHz with 250MHz steps. Voltage scales linearly with the frequency in the specified range. Memory is considered an external domain with a fixed latency.

We evaluate the policy learned with our method using an Alpha-like core configuration. We use a small number of functional units and narrow decode/issue widths to emphasize the CPU-core and L2-cache performance gap. Wider issue and decode widths combined with more functional units increase ILP are more likely to mask cache latencies. The processor configuration used in our simulations is listed in Table 1.

To obtain the propositional rules, we use *JRip* from the WEKA data mining software package [16]. *JRip* is an optimized implementation of the RIPPER learner. The rules are produced based on the data collected for the given architectural configuration. Each rule specifies the desirable CPU frequency and cache frequency for the next program interval based on the current state: CPI, L2PI, old CPU and cache frequencies.

An important aspect of using JRip in the WEKA engine is the format of the training data, which affects the quality of the generated rule set. Although all the state parameters of the training data are discrete (cache and CPU frequencies are discrete in nature, while L2PI and CPI are discretized into bins), we specify in the input to JRip that all

Table 1. Simulation configurations

Parameter	Configuration
Dec./Iss. Width	1/1
dL1 cache	64KB, 2-way
iL1 cache	64KB, 2-way
L2 Cache	1MB DM
L1 lat.	2 cycles
L2 lat.	12 cycles
Int ALUs	2+1 mult/div
FP ALUs	1+1 mult/div
INT Issue Queue	4 entries
FP Issue Queue	4 entries
LS Queue	8
Reorder Buffer	40

parameters are continuous to get a more compact rule set. Using JRip also involves tuning the parameters for the RIPPER algorithm. For instance, the RIPPER algorithm needs to partition the training data into a growing set and a pruning set. We choose the partition size to be two thirds for the growing set. Since RIPPER is a randomized algorithm, different randomization seeds will lead to different results. We experimented with different values and chose a seed value that reduced the error rate and rule set size for our input.

We run a mixture of integer and floating point benchmarks from SPEC2000. The simulations are split into "training" and "evaluation" data. The training data contains the samples used for deriving the policy (i.e., the mapping of states to actions). The policy is evaluated on the evaluation data. In particular, for SPEC benchmarks, the "train" input data set was used for training samples and the "ref input data set" was used for evaluation runs. For both training and evaluation simulations, we fast forwarded the first one billion instructions and simulated the following 500M instruction.

We normalize results to a clairvoyant technique. The clairvoyant policy is obtained by selecting the best CPU-core and L2-cache frequencies for each sample (that is, the CPU-core and L2-cache frequency combination that minimizes the metric). While the clairvoyant algorithm is optimal for energy, note that it is only an approximation of optimal when the metric is the energy-delay product, as minimizing the energy-delay product for every interval does not necessarily minimize the overall energy-delay product for the entire application. We refer to this technique as *local-clairvoyant* in case of optimizing energy-delay product and as *clairvoyant* when optimizing for energy. We report how far the optimized metric is from the local-clairvoyant and clairvoyant results.

We compare our derived policy versus a base policy proposed in [8]. The base policy periodically monitors CPI and L2PI to control the CPU-core and L2-cache domains independently. We use a 500K cycle control period for the periodic voltage changes.

5.2 Experimental Results

Using the methodology from Section 4.1, we derived an integrated DVS policy for our experimental target system. Figure 5.2 compares the energy-delay product resulting from using the independent DVS policy versus our integrated DVS policy. Data is normalized to a local-clairvoyant policy. Lower values in Figure 5.2 are better as they are closer to the local-clairvoyant results. In all applications, we achieve an energy-delay product lower than the independent DVS policy. Reduction in energy-delay product over the independent policy is up to 34% in *art* (10% on average) across all application. More interestingly, the energy-delay results from our policy is within 3% of the local-clairvoyant technique.

In this setting we divided the CPI values into 11 bins (discretization intervals), and eight L2PI bins. Data from the training phase were able to cover 945 states out of 1408 possible states (11 CPI bins x 8 L2PI bins x 16 frequency combinations).

Mapping the states table into rules using *JRip* involves an approximation error. The error rate obtained in our set of rules is 6%. This corresponds to coverage of the training data by the rules of 94% . This implies that the rules are a good approximation of the full training data. For the states not covered by the rules, the action selected, though different, is close to the original. In fact, the differences in the optimization metric results are so negligible that the average error (relative to the full table) across all benchmarks is just 0.1%.

From these results, we conclude that our learning methodology being aware of the CPU-core and L2-cache states is effective and able to derive beneficial policies for the optimization metric (energy-delay product) on our experimental platform.

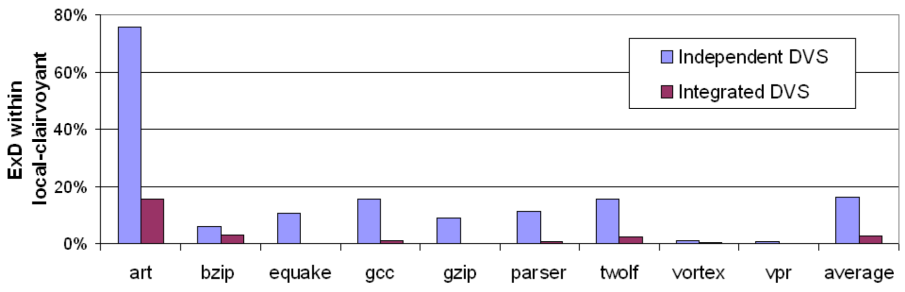


Fig. 2. Percentage increase in energy-delay relative to local-clairvoyant policy.

6 Conclusions and Future Work

In this work, we proposed the use of two important techniques for controlling the power and energy consumption in multiple clock domain processors. First, we proposed an integrated CPU-core and L2-cache DVS scheme that is based on simple performance counters (cache misses and instructions per cycle). Second, we used a supervised machine learning technique to derive a DVS policy for a given architecture. Our proposed scheme learns a frequency and voltage setting policy for scaling both CPU-core and L2-cache simultaneously. Our policy is within 3% of a locally clairvoyant policy.

In future work, we intend to study the impact of the different architectural configurations on our technique's accuracy. Also, we will investigate the significance of varying the learning process parameters (such as training data size, sampling size, and discretization granularity of both CPI and L2PI) on the results.

References

1. P. Bohrer, E. Elnozahy, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case for power management in web servers. In *Power Aware Computing, Kluwer Academic Publications*, 2002.
2. John Cavazos, J. Eliot, and B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 183–194, New York, NY, USA, 2004. ACM Press.
3. K. Choi, K. Dantu, W. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD'02)*, San Jose, CA, November 2002.
4. W. W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, June 1995.
5. K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. In *Proceeding of the* 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.
6. Johannes Furnkranz and Gerhard Widmer. Incremental reduced error pruning. In *International Conference on Machine Learning*, pages 70–77, 1994.
7. G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the* 30^{th} *International Symposium on Computer Architecture (ISCA'03)*, June 2003.
8. Grigorios Magklis, Greg Semeraro, David H. Albonesi, Steven G . Dropsho, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage scaling for a multiple-clock-domain micro processor. *IEEE Micro*, 23(6):62–68, 2003.
9. D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, Philadelphia, PA, October 2000.
10. K. Puttaswamy, K. Choi, J. Park, V. J. Mooney III, A. Chatterjee, and P. Ellervee. System level power-performance trade-offs in embedded systems using voltage and frequency scaling of off-chip buses and memory. In *Proceedings of International Symposium on System Synthesis (ISSS'02)*, Kyoto, Japan, 2002.
11. C. Rusu, A. Ferreira, C. Scordino, A. Watson, R. Melhem, and D. Mossé. Energy-efficient real-time heterogeneous server clusters. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 418–427, San Jose, California, April 2006.
12. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

13. T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30^{th} International Symposium on Computer Architecture (ISCA'03)*, June 2003.
14. M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, California, 1994.
15. Jonathan Wildstrom, Emmett Witchel, and Raymond J. Mooney. Towards self-configuring hardware for distributed computer systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 241–249, Washington, DC, USA, 2005. IEEE Computer Society.
16. I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2005.
17. YongKang Zhu, David H. Albonesi, and A. Buyuktosunoglu. A high performance, energy efficient gals processor microarchitecture with reduced implementation complexity. In *IS-PASS'05: Proc Intl Symp on Performance Analysis of Systems and Software*, pages 42–53, 2005.