Integrated CPU and L2 Cache Voltage Scaling using Machine Learning

Nevine AbouGhazaleh Alexandre Ferreira Cosmin Rusu Ruibin Xu Frank Liberato
Bruce Childers Daniel Mossé Rami Melhem

Department of Computer Science
University of Pittsburg
Pittsburgh, PA 15260
{nevine, apf75, rusu, xruibin, frank, childers, mosse, melhem}@cs.pitt.edu

Abstract

Embedded systems serve an emerging and diverse set of applications. As a result, more computational and storage capabilities are added to accommodate ever more demanding applications. Unfortunately, adding more resources typically comes on the expense of higher energy costs. New chip design with Multiple Clock Domains (MCD) opens the opportunity for fine-grain power management within the processor chip. When used with dynamic voltage scaling (DVS), we can control the voltage and power of each domain independently. A significant power and energy improvement has been shown when using MCD design in comparison to managing a single voltage domain for the whole chip, as in traditional chips with global DVS.

In this paper, we propose PACSL a Power-Aware Compiler-based approach using Supervised Learning. PACSL automatically derives an integrated CPU-core and on-chip L2 cache DVS policy tailored to a specific system and workload. Our approach uses supervised machine learning to discover a policy, which relies on monitoring a few performance counters. We present our approach detailing the role of a compiler in constructing a custom power management policy. We also discuss some implementation issues associated with our technique. We show that PACSL improves on traditional power management techniques that are used in general MCD chips. Our technique saves 22% on average (up to 46%) in energy-delay product over a DVS technique that applies independent DVS decisions in each domain. Compared to no-power management, our technique improves energy-delay product by 26% on average (up to 64%).

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General—*Hardware/software interfaces*; D.3.4 [*Programming Languages*]: Processors—*Run-time environment*

General Terms Performance, Management, Design

Keywords Power management, Integrated DVS policy, Machine learning, Multiple Clock Domains

LCTES'07 June 13–16, 2007, San Diego, California, USA.

1. Introduction

Dynamic Voltage Scaling (DVS) is a technique that can be used to reduce power consumption in CMOS digital circuits. A low clock frequency allows the use of low supply voltage. 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 [15].

Embedded systems are evolving to accommodate a new and diverse set of applications. Such applications require increased processor computational power, larger storage capacity, and longer operation time. The advancement in processor technology creates the opportunity for embedded processors to approach the performance of general purpose processors by adopting performance solutions like large caches, superscalar, and multiple cores. However, adopting most of these solutions leads to more power consumption. Unfortunately, with the plethora of embedded system designs and their applications, it is hard to construct a power management policy that can be directly applied to a variety of embedded systems.

Due to the continuous increase in the number of transistors and decrease in feature size, higher chip densities create a problem for clock synchronization among 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.

In this work, we automatically generate a custom power management policy for embedded processors. We are especially interested in managing the power of the CPU-core and the on-chip L2 cache, as they consume a large fraction of the total power in current processors. We propose a Power-Aware Compiler-based approach using Supervised Learning, *PACSL*. PACSL provides a novel methodology to automatically derive an integrated CPU-core and L2 cache DVS policy. The derived policy dynamically adapts the domains' voltages and frequencies to current workload in an MCD processor. Our approach identifies application phases at runtime and takes corresponding actions (i.e., setting the voltage and frequency of both the processor and the L2 cache).

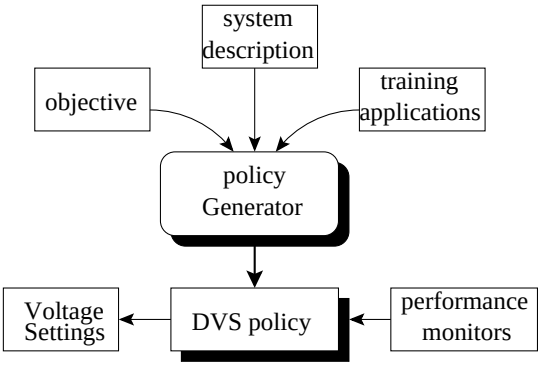

Figure 1. Information flow in PACSL

In PACSL, the automated generation of power management policies relies on given system settings. Our special-purpose compiler takes as an input a state description of the system, which includes the architectural and application behaviors, and an optimization criterion. Based on this input, it generates a custom policy for this particular system. PACSL uses supervised learning process on a set of representative training workload to derive the DVS policy. Figure 1 shows the inputs and outputs in our approach. We evaluate our approach on different processor configurations and compare its performance against a well known online DVS policy that manages each domain voltage independently. Results show 22% average (up to 46%) improvement in energy-delay product over a DVS technique that apply independent DVS decisions in each domain. Compared to no-power management, our technique improves energy-delay product by 26% on average (up to 64%).

One of the advantages of using our approach is its ability to automatically construct a power-management policy for different architectures (embedded and general-purpose) and different classes of applications. Our technique also can be used to optimize for different metrics (such as energy and energy-delay product) that can be set by the user, which is useful with systems that operate in one of multiple operation modes (such as power-saving mode, high performance mode and high performance with temperature/power budget throughout their missions). For each mode, our approach derives a policy that can be loaded at a mode switch to optimize the system for its current optimization criteria. All policies are derived using the same methodology.

The rest of the paper is organized as follows. A motivation and an overview of our approach are presented in Section 2. We describe the essential phases of our compiler for obtaining a policy using supervised learning technique in Section 3, followed by a discussion of some practical design issues in Section 4. We present an evaluation of our technique in Section 5, and briefly discuss related work in Section 6. Finally, we conclude the paper and discuss future work in Section 7.

2. Integrated DVS policy

2.1 Motivation

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 using performance monitors that relate to the activity in each of these domains. Figure 2 shows the variations in three performance counters as examples of monitors that can be used to represent a program behavior. The figure shows: cycles per instruction (CPI), number of L2 accesses per instruction (L2PI), and memory accesses per instruction (MPI). CPI and L2PI are selected as indications of the amount of workload in the CPU-core and L2 cache, respectively. On the other hand, L2PI and MPI can be used to indicate the idleness in the CPU core and the L2 cache domains, respectively.

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 (as with most phases in *equake* in Figure 2). On the other hand, a memory bound phase benefits the most from reducing the gap between the core and cache performance (as with most phases in *art* in Figure 2). Typical applications have alternate CPU and memory bound phases (as shown in *gzip* in Figure 2). 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 appropriate frequencies for the CPU and L2 cache domains for each code section.

2.2 Optimization metrics

The "best frequencies" to use for the CPU-core and L2 cache domains are defined in terms of some optimization metric. There are three natural metrics: energy, performance, and energy-delay product. When the metric is energy, it would seem that the most energy-efficient frequencies are the minimum ones, due to the well-known quadratic relationship between frequency and power [1]. However, when looking at the system as a whole, this is no longer true [4]. Reducing the frequency of one component (e.g., the CPU-core) increases execution time, which increases the energy consumption of other components (due to static power dissipation for longer periods). Thus, the problem of identifying the optimal frequencies that minimize the system energy is far from trivial. When performance is the main requirement, we are interested in minimizing energy while maintaining execution time within a specified percentage of full performance (which corresponds to the highest frequencies available for both CPU and L2 cache). When energy and performance are equally important, the optimization metric is defined as the energy-delay product.

2.3 Overview of our compiler-based approach

We manage domain voltages through a hardware-software codesign approach. Our power management approach consists of two main stages. First, an offline stage where a special-purpose compiler constructs a power management policy. Second, a run-time stage where an embedded microcontroller monitors the system (including the application behavior) and accordingly reacts by setting domain voltages as defined by the policy. Figure 3 shows a schematic diagram of an example processor chip with two domains and a microcontroller that manages the domains voltages according to the policy derived by our special-purpose compiler. PACSL can be used to construct policies for different processor designs.

In the offline stage, PACSL learns the power management policy using sample applications. For this, we use a special-purpose compiler that (a) analyzes the behavior of sample applications and (b) develops runtime DVS policy according to a given optimization metric. The analysis takes into account the architectural behavior while executing the given applications. PACSL derives a policy using a supervised machine learning technique introduced in [12]. The compiler derives separate policies for different optimization metrics. For systems that operate at different modes, the operating system loads the policy that corresponds to the current system

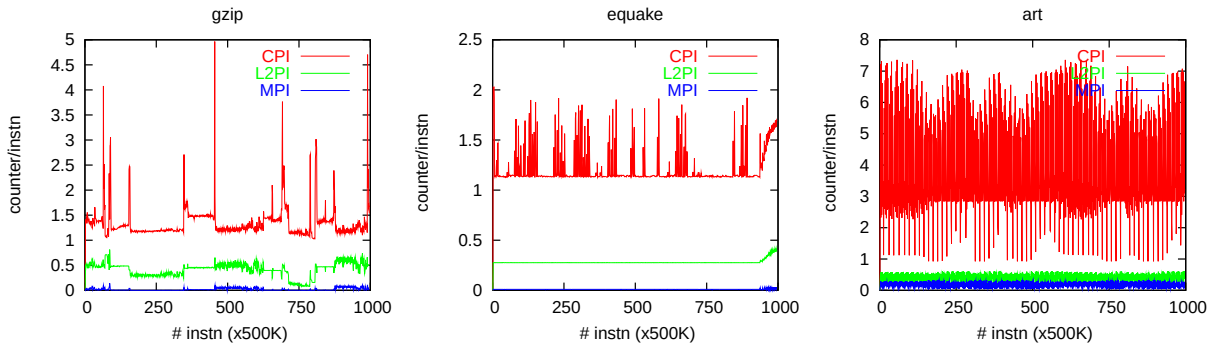

Figure 2. Variations in application phases throughout execution.

mode. Only one analysis phase is needed for all optimization criteria.

During runtime, the microcontroller periodically monitors the activity in each domain by recording a set of performance counters. The microcontroller executes the policy to determine the best frequency combination based on the values of the latest performance counters read.

In the next section we focus on the policy construction process detailing the role of the compiler in this stage.

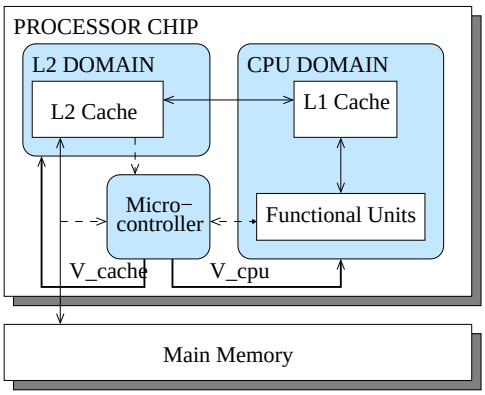

Figure 3. Example of an MCD processor design with integrated DVS control.

3. Construction of DVS Policy

To automatically construct a power management policy, PACSL relies on a description of 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/memory demands of the application. A separate runtime characteristics description captures program latencies during a given program phase. The goal is to identify for each possible system state the correct *action*. For example, an action determines how the CPU-core and L2 cache frequencies should be adjusted to minimize energy-delay product. The compiler outputs a policy that maps states to actions with the objective of optimizing a performance metric (for example: energy and/or delay). The compiler creates the DVS policy by conducting two main stages: data analysis and policy learning. Figure 4 illustrates the main tasks for deriving the power management policy. Below, we describe the tasks performed in each of these two stages.

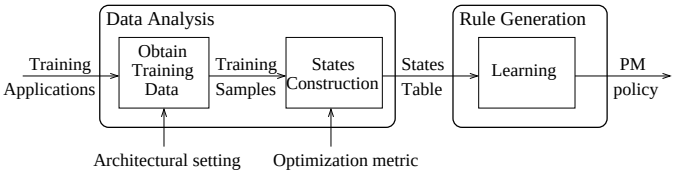

Figure 4. Stages for automatic DVS policy generation.

3.1 Stage I: Data analysis

In this stage, we represent all possible system states by a set of performance counters readings (Section 3.1.1). The compiler discovers the best operating frequencies for each state through an exhaustive search of the training data (Section 3.1.2). The compiler then uses the training data to construct a table that maps a state to a CPU and cache frequency combination (Section 3.1.3).

3.1.1 State representation

In order to train our policy, we need a representation of the system state that encapsulates the program and architectural behavior with simple performance metrics. For example, we select the CPI, L2PI, and MPI, 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, high cache access latency, or long instruction latencies (such as division). Adding both L2PI and MPI into the state description can identify the reason behind the high/low CPI, and hence more fully describes application behavior. However, the CPI, L2PI and MPI do not take into account the effective latency of instruction execution or cache accesses (hits and misses), and to fully characterize the program, these latencies have to be factored into the state description. We describe the effective latency as a tuple of CPU-core and L2 cache frequencies. Since we do not modify the memory speed, we don't include the memory frequency in our state representation. This representation (CPI, L2PI, MPI, CPU-core and cache frequencies) not only captures the total latency, but it also provides an estimate of the total energy, since it is closely related to the operating frequencies.

3.1.2 Obtaining Training Data

The data used to learn the policy is obtained from training benchmarks in the following manner. Let N_c and N_h be the number of CPU and cache frequencies, respectively. We run all training benchmarks at all CPU and cache frequency combinations ($N_c \cdot N_h$

combinations). A sample is defined as a continuous code section of fixed number of instructions equal to $size$. Thus, each training benchmarks with a total of $inst$ instructions will generate $K = inst/size$ code samples for one particular CPU/cache frequency, and $N_c \cdot N_h \cdot K$ samples for all frequency combinations. We denote the samples by $S_{kij} = \{CPI_{kij}, L2PI_{kij}, MPI_{kij}, M_{kij}\}$, where k represents the code sample ($0 \leq k < K$), and i and j are frequency indexes ($0 \leq i < N_c$ and $0 \leq j < N_h$). M_{kij} is the metric to be optimized. Based on the user setting, the PACSL substitutes M_{kij} by either E_{kij}, or ED_{kij} when optimizing for energy or energy-delay product, respectively.

Thus, a state is described by five parameters: CPI, L2PI, MPI, CPU-core frequency and L2 cache frequency. CPI, L2PI, and MPI are continuous variables and need to be discretized. We choose a number of discrete intervals, discretization bins, in a way that the sample densities in each bin are almost equal. 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).

As an illustrative example, consider a system with two CPU and L2 cache frequencies: 0.5GHz and 1GHz. For ease of presentation, we use values for two performance monitors: CPI and L2PI. To reduce the state space, we discretize CPI and L2PI values into two bin intervals. CPI bins cover values [0,1.39] and]1.39,∞], and L2PI bins cover [0,0.02] and]0.02,∞]. Each sample lists its CPI and L2PI bin indexes and the energy-delay product when running at each frequency combination. Table 1 shows the collected samples in our example.

3.1.3 ST construction

After collecting data for all samples, S_{kij}, we construct the state table *ST*, which contains the correct action for each state as determined by the training data. *ST* includes all possible system states. Let c, l, m be the number of discrete values (bins) of the CPI, L2PI, and MPI, respectively. The state table is defined as: $ST[CPI_c][L2PI_l][MPI_m][i][j]$, where CPI_c, $L2PI_l$, and MPI_m are the discretized values. For each state we want to determine the action that minimizes a user-selected optimization metric; for example, the energy-delay product.

We construct the table 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:
$Acc[CPI_{kij}][L2PI_{kij}][MPI_{kij}][i][j][x][y]$, where CPI_{kij}, $L2PI_{kij}$, MPI_{kij}, 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_{kij} and each possible action x, y (x is the next CPU frequency, y is the next cache frequency), we update the array as follows.

for all samples **do**
  for all CPU frequency $i = 0 \ldots N_c$ **do**
    for all Cache frequency $j = 0 \ldots N_h$ **do**
      for all future CPU frequency $x = 0 \ldots N_c$ **do**
        for all future Cache frequency $y = 0 \ldots N_h$ **do**
          $Acc[CPI_{kij}][L2PI_{kij}][MPI_{kij}][i][j][x][y] += M_{kxy}$
        end for
      end for
    end for
  end for
end for

The array Acc accumulates the values of the optimization metric, M_{kxy}, for all training samples and under all possible actions. After updating the array for all samples, the action for each state is the one that minimizes the metric. In other words, after updating the Acc table for all samples, the action for each state, $ST[CPI_{kij}][L2PI_{kij}][MPI_{kij}][i][j]$, is the frequency combination $\langle x, y \rangle$ that produce the minimum $Acc[CPI_{kij}][L2PI_{kij}][MPI_{kij}][i][j][x][y]$. The resulting state table is a five dimensional table (three features and frequencies for two domains) that represents all possible system states. *ST* maps a state to the best frequency that optimizes the metric under consideration.

Table 2 shows the state table for the training samples shown in Table 1. State description is composed of CPI, L2PI and the two domain frequencies (f_{cpu} and $f_\$$). The table shows the best frequency combinations (CPU frequency / Cache frequency) for each state description. Note that not all of the *ST* states are populated (unpopulated states are marked by -). That is why we use machine learning to learn the best frequency combination in the missing states based on the discovered ones.

Table 2. Constructed *ST* from samples in Table 1.

		f_{cpu}=0.5GHz		f_{cpu}=1GHz	
CPI	L2PI	$f_\$$=0.5	$f_\$$=1	$f_\$$=0.5	$f_\$$=1
0	0	1/0.5	1/0.5	-	-
0	1	1/1	1/1	-	1/1
1	0	1/0.5	-	1/0.5	1/0.5
1	1	1/0.5	-	1/1	1/0.5

Since the training data do not cover all possible states in the table (because some states may not be discovered from the training applications/data). We use a supervised machine learning algorithm to derive the DVS policy that can react (select new CPU and cache frequencies) to any possible system state.

3.2 Stage II: Integrated DVS policy learning

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 most closely models ST.

We use the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) learner [3]. 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 propositional (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 [5]. RIPPER repeatedly calls IREP to construct the rule set with low error rates.

IREP iteratively builds its rule set in a greedy fashion; that is, one rule at a time. IREP works in two phases: growing and pruning. First, it randomly partitions the data set in to two subsets: the 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 that 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.

Table 1. Eight training samples: CPI, L2PI and energy-delay product (ED) at all frequency combinations. O and 1 are the index of the CPI and L2PI bins.

f_{cpu} / $f_\$$	0.5GHz / 0.5GHz			0.5GHz / 1GHz			1GHz / 0.5GHz			1GHz / 1GHz		
s	CPI	L2PI	ED	CPI	L2PI	ED	CPI	L2PI	ED	CPI	L2PI	ED
1	0	1	200	0	1	354	1	1	183	1	1	187
2	0	1	242	0	1	428	1	1	223	1	1	226
3	0	0	436	0	0	768	1	0	395	1	0	403
4	0	1	274	0	1	481	1	1	252	0	1	250
5	0	0	473	0	0	826	1	1	430	0	0	430
6	1	1	330	0	1	588	1	1	309	1	1	317
7	1	0	361	0	0	642	1	0	327	1	0	339
8	1	0	401	0	0	709	1	0	363	1	0	374

The resulting rules are generated in the form of: IF <*cond*> THEN <*set_freq*>, where *cond* is a conjunction of one or more of the following sub-conditions. ($CPI_{cur} \leq CPI_c$), ($CPI_{cur} \geq CPI_c$), ($L2PI_{cur} \leq L2PI_l$), ($L2PI_{cur} \geq L2PI_l$), ($MPI_{cur} \leq MPI_m$), ($MPI_{cur} \geq MPI_m$), (c_f = i), and (m_f = j) where CPI_{cur}, $L2PI_{cur}$, MPI_{cur}, c_f and m_f are the current CPI, L2PU, MPI, CPU frequency and cache frequency, respectively. *set_freq* specifies the value of the next CPU or cache frequencies. Rules learned from *ST* in Table 2 are shown in Table 3. Note that the number of rules is very small because of the simplified system setting chosen in our hypothetical example.

Table 3. Example of a policy to minimize energy-delay product.

#	Rule
1	if (L2PI $\geq$ 1) and (CPI $\leq$ 0) then $f_\$$=1GHz
2	else $f_\$$=0.5GHz
3	f_{cpu}=1GHz

4. Design Issues

Feature Selection Ultimately, all DVS policy decisions are based entirely on the current system state. It is important, therefore, to characterize the state in terms of features which provide relevant information about the current application phase. Our hypothesis, based on architectural knowledge, is that CPI, L2PI, and MPI (along with the current CPU-core/cache frequencies) capture enough about application behavior to make informed choices, while still being inexpensive to gather at runtime. We can use more features to represent a system state; however, using too many features can cause an *overfitting* (a known problem in machine learning) where it is hard to create a general state description for the unseen states.

Optimization metrics Once the training samples are collected, the data used to learn the DVS policy can be obtained for different metrics. The collection of training samples is a one-time step and is independent to the optimization metric. Thus, metrics can later be changed by simply updating M_{kxy} in Equation (6). One of the strengths of our approach is that the compiler needs to analyze the system only once and therefore generate different policy for each optimization metric.

Training applications Training applications are selected based on the diversity of the states each application can produce. Applications that compose the set of training applications should complement the others in the set by increasing the number of different states with information. The more ST is populated, the more accurate the policy can derive actions for the unseen states. In general, it is desirable to use representative applications that include memory-bound and CPU-bound phases to cover more states in the table. Also, applications that have a large variation of the behavior in its phases, such as *gcc*, will highly contribute to *ST* population.

Sample size We chose DVS control intervals measured in number of instructions instead of number of cycles (or time) to evaluate different actions for the same code sample. The table describing the derived policy can actually be used with a periodic timer-based mechanism. Different sample $size$ values result in the same policy rules, as long as $size$ is not larger than the application phases. From an architectural perspective, periodically selecting new frequencies is a more immediate approach, because it only requires a simple timer-based interrupt mechanism.

Overheads The overhead of a frequency change is typically just a few microseconds. However, voltage change overheads are higher, ranging from a few dozen microseconds to a few milliseconds (e.g., for StrongARM SA-1100 the voltage change overhead is $140\mu s$ [10]). The sample $size$ depends on the total overhead: a small sample may have high frequency/voltage change overheads, while a large sample may exceed typical code phases of applications. For example, changing the frequency/voltage every $1ms$ with an overhead of $140\mu s$ yields an overhead as large as 14%. The overhead can be mitigated by enforcing a limit on the number of speed changes. For example, simple schemes can enforce at most one speed change say every $10ms$, without changing the sample size. Alternatively, the size can be increased. Even better, while the overhead for a voltage change is large, frequency changes are very fast, and the system is operational while the voltage is scaling. When the frequency is decreased, the system immediately changes the frequency, although it will take a while for the voltage to lower. When the frequency is increased, the system runs on the old frequency until the voltage is raised, after which the frequency is increased as well. Thus, the actual overhead is just the frequency change overhead, though frequency increases may take effect with a delay (for example, $140\mu s$ in the strong arm processor). Such delays are much shorter than application phases and do not affect the policy. Note also that voltage change overheads are today in the microseconds range [9].

Inefficient operating points Processors may have inefficient frequency/voltage combinations [8]. A frequency is inefficient if there exists a higher frequency that results in lower energy consumption. Another advantage in our approach is that the compiler can determine the best action and inefficient operating points are naturally eliminated if they exist.

Measurement-based versus theoretical models We use a measurement based approach (i.e., experiments are run to derive a policy), as opposed to an analytic model-based approach. Thus, there is no implicit assumption of theoretical power models (such as power relationship with the voltage and frequency). This means

Table 4. Simulation configurations

Parameter	Config A	Config B
Decode width	1 insn	4insn
Issue width	1 insn	6insn
dL1 cache	64KB, 2-way	64KB, 2-way
iL1 cache	64KB, 2-way	64KB, 2-way
L2 Cache	1MB DM	1MB DM
L1 lat.	2 cycles	2 cycles
L2 lat.	12 cycles	12 cycles
Int ALUs	2+1 mult/div	4+1 mult/div
FP ALUs	1+1 mult/div	2+1 mult/div
INT Issue Queue	4 entries	20 entries
FP Issue Queue	4 entries	15 entries
LS Queue	8	64
Reorder Buffer	40	80

that the policy works well in identifying the correct actions without assuming of whether the system supports DVS or just frequency scaling (for example) and without assuming of the relationship among voltage, frequency and power. While a model can be inaccurate and difficult to construct, measurement-based approaches eliminate this problem from the start, at the expense of one-time offline measurements.

5. Evaluation

In this section, we analyze the effectiveness of PACSL methodology. We state our experimental setup, evaluate PACSL by comparing it to an independent DVS policy under several system settings, and analyze the training process.

5.1 Experimental Setup

We use the Simplescalar and Wattch architectural simulators with an MCD processor extension [19]. The MCD extension by Zhu et al. models inter-domain synchronization events and voltage scaling overheads. We alter the design in [19] to construct two domains: CPU-core and L2 cache, as shown in Figure 3. 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. The processor configuration used in our simulations is listed in Table 4. Unless stated otherwise, we use the single-issue processor configuration (Config A) for our results. We discretize CPI values into 11 bins, L2PI into 8 bins, and MPI into 4 bins.

To obtain the propositional rules, we use *JRip* from the WEKA data mining software package [17]. *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 (that is, CPI, L2PI, MPI, old CPU and cache frequencies).

An important aspect of using JRip 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 CPI, L2PI, and MPI are discretized into bins), we specify in the input to JRip that all 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 ran a mixture of the SPEC2000 and Mibench benchmarks. The simulations are split into *training* and *evaluation* simulations. The training simulation executes a subset of the applications to generate the samples used for deriving the policy (i.e., the mapping of states to actions as described in Section 3). Training applications are listed in Figure 9. For the SPEC2000 benchmarks, the training simulations use the *train input data set* and the evaluation simulations use the *ref input data set*. For the Mibench, we use the *small data set* for training and the *large data set* for evaluation. For unbiased evaluation, 21 out of the 24 reported benchmarks were not part of the training process. We fast forward the first 1000 (500) million instructions for the SPEC2000 (Mibench) benchmarks and simulate the following 500M instruction. Exceptions are the small benchmarks in Mibench suite: *dijkstra*, *fft* and *jpeg*, where we run the application for the first 500M instructions or until completion.

We compare our derived policies against the *attack-decay* policy proposed in [7], which periodically monitors CPI and L2PI to control the CPU-core and L2 cache domains independently. We use a 500K instruction control period for the periodic voltage changes. We normalize all results to no-power management case, which operates both domains at the maximum frequencies.

5.2 Experimental Results

In this section, we show the energy-delay product results of the policies learned from PACSL in comparison with an independent CPU-core and L2 cache DVS policy [7]. We show how PACSL is affected by the class of applications, architectural configurations, and the optimization metric (as shown in Figure 1).

Energy-delay product improvement Figure 5 shows the energy-delay product for the independent DVS policy versus the ones generated by PACSL[1]. The generated policy (with the use of MPI) contains 33 rules. On average, PACSL's policy improves energy-delay product over the independent policy by 21% and 22% for the Mibench and SPEC2000, respectively. The independent policy being a heuristic-based can perform badly with some applications as seen in *crc32*, *dijkstra* and *mesa*. This is because the policy was unable to select the best frequency to minimize energy-delay. It rather reacts to the CPI and L2PI changes by changing the frequencies in the same direction of their change, which does not guarantee operating on an efficient frequency.

State description Figure 5 also shows the energy-delay product in case of discarding the MPI feature from PACSL state description. In this case, PACSL generates a policy with fewer rules: 27 rules. In Mibench, using MPI does not improve the energy-delay product because most of the applications' data accesses occur in the caches. So memory latency has trivial effect on the applications performance and energy. In contrast, SPEC2000 benchmarks have larger memory footprints, thus using MPI in the state description enables the rule learner to distinguish between memory bound versus L2 cache bound phases. Hence, PACSL with MPI further improves energy-delay product by 1.8% on average and up to 8.6% (in *mgrid*).

Optimization metric PACSL analyzes the application and the architectural behavior once, then it can generate policies geared towards optimizing a given metric. To show this, we use the same

[1] Note that results for the independent policy differ from the ones reported in [7]. This is because of few reasons. As reported by [18], we use an updated version of the simulator and simulate different window size. Moreover, our MCD design includes only two domains versus six in the original design, and we use only four frequency levels whereas work in [7] uses continuous frequency range (320 levels).

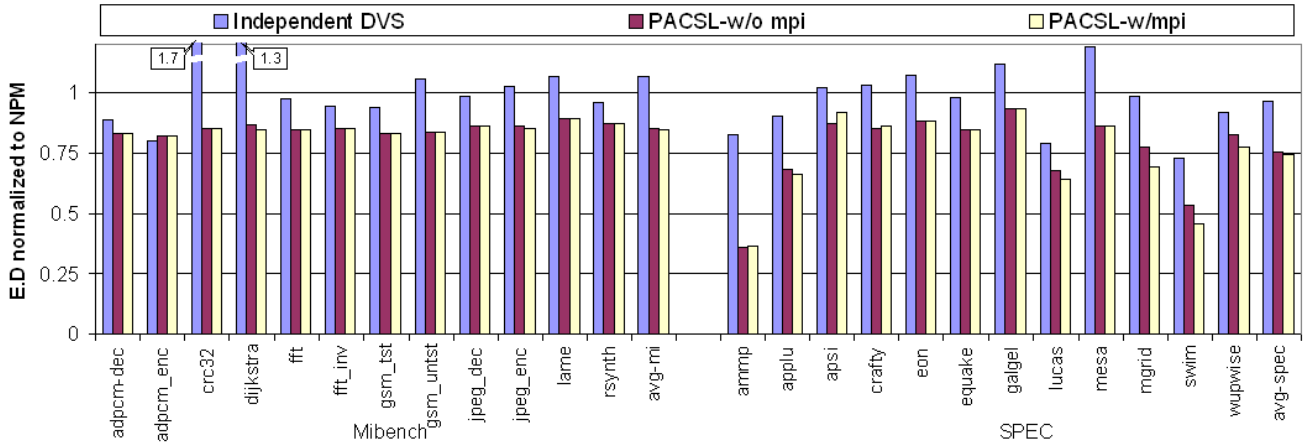

Figure 5. Energy-delay product for SPEC2000 and Mibench benchmarks when using Independent DVS versus PACSL.

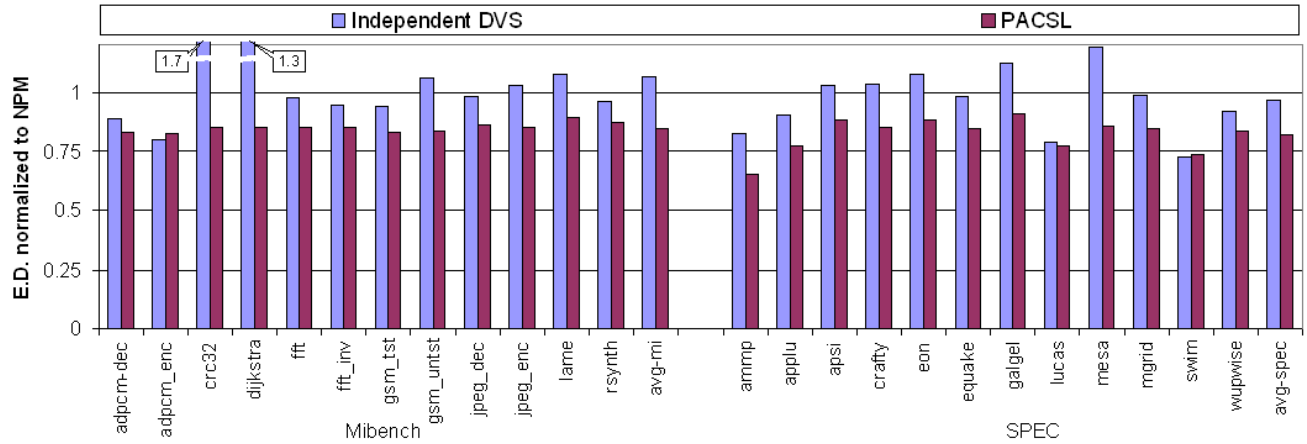

Figure 6. Energy-delay product when optimizing energy with delay bound.

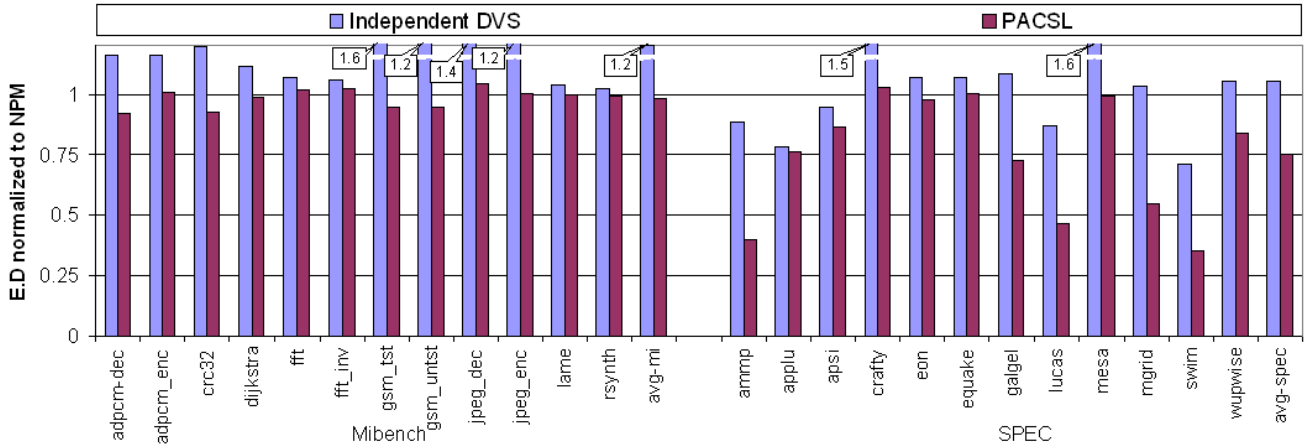

Figure 7. Energy-delay product for policies running on system with configuration Config B in Table 4.

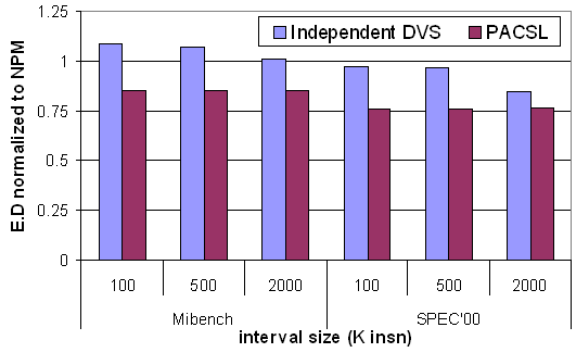

Figure 8. Average energy-delay product at different DVS control-interval sizes (using Config A).

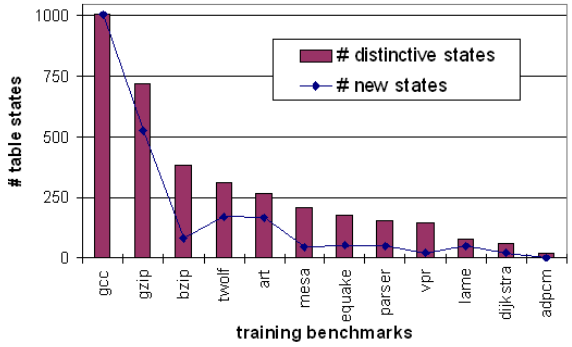

Figure 9. *ST* coverage.

training samples obtained for energy-delay product and construct *ST* to select the best frequency combination that minimizes the energy while maintaining the delay within bound. We show results for 10% bound on performance degradation. A new policy is generated with this objective. Figure 6 shows the energy-delay product of our benchmarks (when accounting for MPI). On average, we achieve 21% and 14% improvement over the independent policy for the Mibench and SPEC2000, respectively.

Processor configuration PACSL can be used with different architectural configurations. To show the impact of using PACSL with wide range of processor configurations, we experiment with high-performance processor configuration. For this experiment, we use an alpha-like configuration shown in Table 4 (Config B). Figure 7 shows an improvement of 22% and 31% for the Mibench and SPEC2000 benchmarks, respectively.

Control Granularity One important parameter of a DVS policy is how often to trigger a speed change. Few speed changes reduce overhead but also eliminate the fine grain control to adapt to shorter program phases, and vice versa. In this experiment, we vary the period in which we trigger the DVS policy. We report the average energy-delay product– normalized to no-power management– of all reported Mibench and SPEC2000 benchmarks. Figure 8 shows that by increasing the control interval size, the independent policy reduces the number of speed changes, which reduces the policy overhead and thus reduces its energy-delay product. On the other hand, our DVS policy naturally has fewer speed changes because it selects the best frequencies for a given state rather than changing the frequency based on reaction to a change in feature value as in the independent policy. Hence, increasing the control interval size has minimal impact on the energy-delay product in the tested range.

From the results in this section, we conclude that our learning methodology is capable of generating policies that can used to optimize different systems. The policies being aware of the system state are effective in optimizing the system (for example, by reducing the energy-delay product or energy with limited performance degradation).

5.3 Analysis of the training process

In this section, we study the data analysis phase for obtaining a policy to optimize the energy-delay product (same policy used to obtain the results shown in Figure 5).

5.3.1 *ST* coverage

We investigate the efficacy of the training data in discovering the possible states in *ST*, which are used in generating the policy rules (as described in Section 3). The objective is to discover most of *ST* from the training sample. Each of the applications used in training covers a number of states in the table with some applications having larger coverage than others. Figure 9 shows the number of distinct states that each application can discover. The applications are sorted in descending order by number of states. The line graph in the figure represents the number of new states contributed to *ST* by each application. Intuitively, using applications with a higher number of discovered states is more beneficial in the training process as they add more information to *ST*. For example, the first four applications (*gcc*, *gzip*, *bzip* and *twolf*) were responsible for 82% of the *ST* coverage. However, using applications with large number of distinctive states but populating states that were already discovered in *ST* is not useful. For example, *bzip*, exhibits similar behavior to *gzip*, thus, few new states were populated in *ST* by *bzip*. Conversely, *art* is very useful to populate an area not covered by the other applications. Hence, a desired characteristic for applications to use in training is to exhibit large variations in program behavior (phases) that are different than other training applications used. By carefully choosing a few applications with varying behavior, *ST* can be covered with relatively small number of training applications.

5.3.2 Rule Simplification

As mentioned earlier, when populating the *Acc* table with samples from the training data, inevitably some entries will have no data. These are extrapolated by the RIPPER algorithm, when the table is converted into rules. However, some entries will have a small, but non-zero, number of samples. Since these under-populated entries might not represent good average-case behavior for the corresponding system state, it might be beneficial to exclude them from the data given to the RIPPER algorithm.

By omitting states which contain fewer than 60 samples when applying the RIPPER algorithm, we arrive at a DVS policy containing only 11 rules (with MPI). The difference in energy-delay product between the original and reduced policies is generally less than 1%. We hypothesize that many of the under-populated states simply add noise to the resulting DVS policy, while also increasing the complexity of the rules. Determining the value of a given state in deriving a DVS policy is a subject of future work. Reducing the number of rules has the advantage of reducing the run-time overhead of the DVS policy as fewer conditions are tested.

6. Related Work

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). Magklis

et al. propose an online power management policy that monitors queue occupancy of a domain and adapts the domain's voltage accordingly [7]. 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 [6]. 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 [19]. Voltage scaling of off-chip L2 caches for embedded systems is studied in [11].

Sherwood et al. showed that programs have repeatable phase-based run-time behavior over many hardware metrics, such as cache behavior or branch prediction [14]. 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 [13].

Applying machine learning techniques to reconfigure architectural and compiler settings is a relatively unexplored field. Wildstrom et al. present a policy to alter server configuration in reaction to workloads [16]. 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.

7. Conclusions

In this work, we propose a compiler-based approach to automatically generate integrated DVS policies, which manage power in both CPU-core and L2 cache. We characterize the system state by the running application behavior on the given architectural configuration. Our power-aware compiler approach, PACSL, uses machine learning to learn policies given description of the system states. The learned policies are constructed to optimize the system power according to a user selected optimization metric, such as energy, energy-delay product or energy under limitation on performance constraints. We show that our approach generates efficient policies that can achieve larger improvement in energy-delay product over a heuristic-based policy. Our technique saves 22% on average (up to 46%) in energy-delay product over a DVS technique that apply independent DVS decisions in each domain. Compared to no-power management, our technique improves energy-delay product by 26% on average (up to 64%). Our approach can be applied to a wide range of systems that employ dynamic voltage scaling.

References

[1] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.

[2] J. 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. ACM Press, 2004.

[3] W. W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, June 1995.

[4] X. Fan, C. S. Ellis, and A. R. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems (PACS'03)*, 2003.

[5] J. Furnkranz and G. Widmer. Incremental reduced error pruning. In *International Conference on Machine Learning*, pages 70–77, 1994.

[6] 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.

[7] G. Magklis, G. Semeraro, D. H. Albonesi, S. G. . Dropsho, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage scaling for a multiple clock domain microprocessor. *IEEE Micro*, 23(6):62–68, 2003.

[8] A. Miyoshi, C. Lefurgy, E. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16^{th} Annual ACM International Conference on Supercomputing*, New York, June 2002.

[9] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lparm microprocessor system. In *Proc. of the International Symposium on Low Power Electronics and Design (ISLPED'00)*, pages 96–101, 2000.

[10] J. Pouwelse, K. Langendoen, and H. Sips. Application-directed voltage scaling. In *IEEE Transactions on Very Large Scale Integration (TVLSI)*, Sept. 2002.

[11] K. Puttaswamy, K. Choi, J. Park, V. J. M. 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.

[12] C. Rusu, N. AbouGhazaleh, A. Ferreria, R. Xu, B. Childers, R. Melhem, and D. Mossé. Integrated cpu and l2 cache frequency/voltage scaling using supervised learning. In *Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilation (SMART)*, 2007.

[13] 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*, 2002.

[14] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30^{th} International Symposium on Computer Architecture (ISCA'03)*, 2003.

[15] 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, 1994.

[16] J. Wildstrom, E. Witchel, and R. 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.

[17] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2005.

[18] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS-XI: Proc Intl Conf on Architectural support for programming languages and operating systems*, pages 248–259, 2004.

[19] Y. Zhu, D. H. Albonesi, and A. Buyuktosunoglu. A high performance, energy efficient gals processor microarchitecture with reduced implementation complexity. In *ISPASS'05: Proc Intl Symp on Performance Analysis of Systems and Software*, pages 42–53, 2005.