Coordinated, Distributed, Formal Energy Management of Chip Multiprocessors

Philo Juang
pjuang@princeton.edu

Qiang Wu
jqwu@cs.princeton.edu

Li-Shiuan Peh
peh@ee.princeton.edu

Margaret Martonosi
mrm@ee.princeton.edu

Douglas W. Clark
doug@cs.princeton.edu

Departments of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ

ABSTRACT

Designers are moving toward chip-multiprocessors (CMPs) to leverage application parallelism for higher performance while keeping design complexity under control. However, to date, no power management techniques have been proposed for coordinated power control of multiple processor cores.

In this paper, we illustrate how the use of local, per-tile dynamic voltage and frequency scaling (DVFS) techniques can result in tiles counteracting each others' power management policies, significantly hurting chip power-performance. We then propose a coordinated DVFS scheme for CMPs, which eliminates the oscillations and ensures efficient and resilient DVFS control. Specifically, our proposed technique incorporates thread information collected at run-time across the chip. In addition, by extending a control-theoretic local DVFS control technique toward DVFS for chip-multiprocessors, our technique prescribes DVFS settings formally at each tile, thus ensuring **stable, distributed, coordinated** DVFS control of a CMP. Experimental results show that our technique achieves a 15.5% improvement in energy-delay product over a CMP with no DVFS control, and a 7% improvement in energy-delay product against the latest state-of-the-art local DVFS scheme.

Categories and Subject Descriptors: B.1.1 [Control Design Styles]: Hardwired control

General Terms: Design, Algorithms, Performance

Keywords: Power, Dynamic Voltage Scaling

1. INTRODUCTION

Power-efficiency and thermal-efficiency are increasing concerns for both embedded and high-end chip multi-processor systems. With uniprocessor chips we know that often there is insufficient work to fully occupy the processor, due to memory latencies and lack of parallelism. Running the processor at full speed thus only wastes energy. Instead, one can change the voltage and frequency to scale down the speed of the processor to match the decreased requirements in processing performance. This technique, called Dynamic Voltage and Frequency Scaling (DVFS), is common in both embedded as well as high-performance processors.

With multiple processors on the chip, power issues are compounded by the presence of more processors and that these processors often interact as they cooperatively process an application. Most DVFS techniques proposed to date typically apply to single processors such as the Intel XScale or Pentium-M [6, 3, 7]. Some recent work has looked at processors with several internal clock domains—multiple-clock-domain (MCD) processors [10, 15, 16, 19, 20]—but are restricted to local solutions in which each domain is considered separately. Such local, per-tile DVFS techniques lead to unstable DVFS control of CMPs processing multithreaded applications, worsening overall chip power-performance (See Section 2).

The majority of current DVFS work can be characterized along two general lines: the level of dynamism (online or offline) and of formality (ad hoc or formal). Most prior DVFS work is either a profile-based optimization approach (off-line formal) [5, 9, 10, 21] or a run-time heuristic based (online ad hoc) approach [15, 16, 8, 11]. However, CMPs pose a major problem to these techniques. First, CMPs typically execute several applications at once, making it difficult to obtain a representative profile. Second, the number of processors and possible placements explosively increases the tuning space for ad hoc approaches.

This motivates us to investigate run-time, control-theoretic solutions. Currently, the best known online formal DVFS approach is described in [19], which uses a control-theoretic approach in the context of MCD processors. We will refer to this scheme as *local-PID*. *Local-PID* is a purely local scheme—only information local to a tile is used, with interaction across multiple tiles ignored. In this paper, we propose a distributed version of this scheme, applying the basic mechanics of the formal approach to CMPs, while realizing stable, distributed, coordinated DVFS control.

Our paper is structured as follows. Section 2 describes the *local-PID* scheme and its drawbacks when used in a CMP. Section 3 proposes our scheme, called *dist-PID*. Next, Section 4 details our simulation setup and benchmarks, following with the results. Finally, we draw our conclusions and sketch plans for future work in Section 5.

2. MOTIVATION

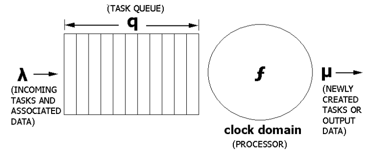
(TASK QUEUE)
q
λ
(INCOMING TASKS AND ASSOCIATED DATA)
f
clock domain
(PROCESSOR)
μ
(NEWLY CREATED TASKS OR OUTPUT DATA)

Figure 1: Each CMP tile modeled as a queueing system.

To motivate the need for distributed, coordinated DVFS, we first study the effect of local DVFS in a CMP, where each tile's frequency (and voltage) is set independently based on local information. Our study is based on *local-PID*, which is representative of per-tile queue-based DVFS policies [19, 20] and was shown to have better energy efficiency than prior ad hoc DVFS approaches.

```
void sum_sqrt() {
    int num, div = 0;
    float sum = 0.0;

    thread_recv(&num, sizeof(int), ANY);

    while(div != -1) {
        if((div%num) == 0) sum += sqrt(num);
        thread_recv(&div, sizeof(int), ANY);
    }

   _thread_send(&div, sizeof(int), ANY);
   _thread_terminate();
}
```

```
void main() {//Thread MT
    int i, child1, child2, child3;

    child1 = thread_create(&sum_sqrt);
    child2 = thread_create(&sum_sqrt);
    child3 = thread_create(&sum_sqrt);

    thread_send_int(2, sizeof(int), child1); //send num
    thread_send_int(17, sizeof(int), child2);
    thread_send_int(10000, sizeof(int), child3);

    for(i = 0; i < 1000; i++) {
      thread_send_int(i, sizeof(int), child1); //send div
      thread_send_int(i, sizeof(int), child2);
      thread_send_int(i, sizeof(int), child3);
    }
    thread_send_int(-1, sizeof(int), child1);
    thread_send_int(-1, sizeof(int), child2);
    thread_send_int(-1, sizeof(int), child3);
    thread_wait_children();
   }
```

Figure 2: Sample source code

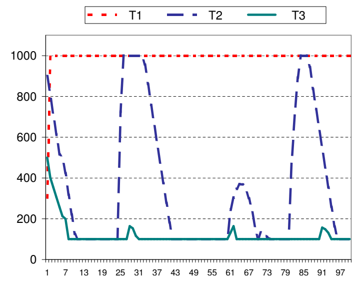

Figure 3: Frequency selection for execution of code in Figure 2

While used in an MCD context, the solid formal control-theoretic principles behind *local-PID* can be readily applied to single-clock-domain CMPs. Each tile is modeled as a local queue model as shown in Figure 1, with each tile processor fed by a task queue, where threads scheduled on the processor await execution. The service rate is denoted as μ, which is determined by the tile processor frequency *f*. Demand is represented as λ, which is the arrival rate of new tasks. Queue occupancy **q**, refers to the summation of each task in the task queue multiplied by its expected relative execution time (the *load factor*, whose derivation will be elaborated in Section 3.2). Conceptually, we seek to match the service rate μ with demand λ such that the average queue occupancy **q** remains constant from interval to interval; this implies that the processor has supplied just enough performance to meet the processing requirement of the application, and thus the maximum amount of energy has been saved.

Local-PID computes μ at interval *k* as:

$$\mu_k = \mu_{k-1} + K_i(q_k - q_{ref}) + K_p(q_k - q_{k-1}) \qquad (1)$$

where μ_{k-1} is the service rate of the last interval, q_k and q_{k-1} the average queue occupancy of the current interval *k* and previous interval *k-1*. The remaining variables are fixed: K_i and K_p constants that are picked based on control-theoretic principles to ensure stability, and q_{ref} the steady-state desired queue occupancy constant. Essentially, the above Equation 1 predicts the needed service rate to *eventually* bring the queue occupancy to q_{ref}.

Conversely, if one assumes a μ_k (proportional to the processor frequency), and solves for q_{ref}, then the equation produces what the eventual value of q_{ref} (target task queue size) should be in response to frequency setting μ_k. Solving for q_{ref} we get:

$$q_{ref} = (K_p(q_k - q_{k-1}) + K_i q_k - \mu_k + \mu_{k-1})/K_i \qquad (2)$$

2.1 Limitations of Local-Only Control

Consider the following four-threaded program as shown in Figure 2. The code on the right is the main "master thread" (Thread *MT*). *MT* launches three threads running the code on the left with initial parameters 2 (Thread *T(2)*), 17 (Thread *T(17)*), and 10000 (Thread *T(10k)*). Each thread is run on a separate tile on the CMP. *MT* creates three threads, running the function `sum_sqrt`. *MT* then sends each thread their initial argument (num), before moving into a loop in which it sends a stream of numbers (div) to each thread—each number is sent three times. Each thread then compares the value received and if it is divisible by the initial argument, adds the square root to a sum. If the value is -1, the program ends.

Based on profiling, *T(17)* spends 169 cycles per element, whereas *T(2)* needs 540 cycles on average. *T(10k)* has virtually no work, and spends fewer than 100 cycles per element. Ideally, the tile executing *T(2)* should run at full speed, while the one executing *T(17)* should run at approximately one-third full frequency to minimize energy-delay-product. *T(10k)* should execute at minimum speed.

Figure 3 shows a sample execution of Figure 2 using the default settings of (K_i=0.6 and K_p=0.2) for *local-PID*. *T(2)* saturates and runs at full frequency as expected, and likewise, *T(10k)* drops to the minimum frequency, occasionally increasing its speed to compensate for small perturbations in execution. Unexpectedly however, rather than settling at a lower frequency, *T(17)* oscillates between low and high DVFS settings. With only local information, there is no way that *T(17)* can know that it can slow down and match *T(2)*.

The oscillation may be smoothed out by using a larger interval, allowing *T(17)* to deal with the large jump in queue occupancy before the next decision is to be made. However, there are two conflicting objectives here. Larger intervals are desirable to reduce the effects of sharp transient spikes in queue occupancy. Shorter intervals, though, are desirable as they have more potential for DVFS.

T(17) being unable to realize it can slow down is not unique to *local-PID*. Basing DVFS decisions locally and statically, whether it be q_{ref} or IPC or some other metric, will be unrepresentative of thread imbalance between tiles. In *local-PID* in the above example, once *MT* stops sending items (indicating the end of this parallel section), the instant the queue occupancy drops below q_{ref} *local-PID* will begin slowing the tile down, even if it is the only thread still running and thus is the critical path. If a tile is on the critical path, the DVFS technique should not reduce the clock speed.

Even with the best-known local DVFS technique (*local-PID*)—a technique designed for multiple interacting domains (multiple clock domain processors)—purely local techniques performed still poorly with a simple parallel application on a CMP. Clearly, this example illustrates the limitations of oblivious, local DVFS control in a CMP, motivating the need for distributed, coordinated DVFS.

3. COORDINATED DVFS CONTROL ALGORITHM

Our example in Section 2 points to the limitations of DVFS based on purely local information. More specifically, the target q_{ref} needs to (1) adapt at run-time to match thread behavior; (2) be based on global information, rather than fixed locally; and (3) be set to preserve performance. Here we propose a formal, online method that supports stable, distributed, coordinated DVFS control in CMPs. We refer to this method as *dist-PID*.

The intuition behind *dist-PID* is that, to preserve performance while maximizing energy savings, it is necessary to identify threads that lie on the *critical path*. Those threads should be run at max speed to preserve performance, while others slowed to maximize energy savings without impacting performance. In parallel applications, these are last threads to reach a synchronization point. If each tile knows the expected execution time of the longest-running-thread, they can adjust their processing speeds to match that. Determining which specific threads are on the critical path is difficult; instead, *dist-PID* chooses which tile has the most work left to do. Because parallel sections require all threads to finish before moving on, the last tile to finish will essentially be the critical path.

Dist-PID operates in three steps:

1. At each tile, estimate future queue occupancy (tile workload) using Equation 2, assuming the maximum service rate, and renaming q_{ref} as q_{target}:

$$q_{target} = (K_p(q_k - q_{k-1}) + K_i q_k - \mu_k + \mu_{k-1})/K_i \quad (3)$$

2. Through pair-wise communications, each tile identifies the tile scheduled with the critical-path-thread through keeping track of the highest q_{target} across the chip
3. With this information, each tile re-solves Equation 1 to determine the new service rates (tile frequency settings), essentially slowing down tiles not executing critical-path-threads.

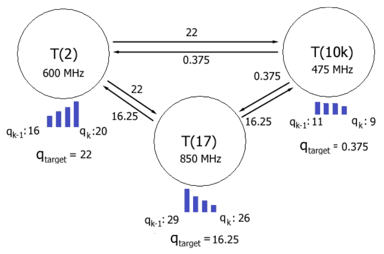

Figure 4: Example of how *dist-PID* assesses the code in Figure 2

Figure 4 shows a snippet of execution near the beginning of the code in Figure 2. On tile *T(2)*, queue occupancy increased as the tile is unable to keep up with the processing requirements, while the reverse is true at *T(17)* and *T(10k)*. Thus we expect to lower the frequency for *T(17)* and *T(10k)*, and increase it in *T(2)*. Equation 2 estimates the q_{target} for each tile, and as we expect, tile *T(2)* which is experiencing the highest load will have the highest q_{target}.

Thus each tile selects *T(2)*'s q_{target} to use as the q_{ref} to re-solve Equation 1. Mathematically *T(2)* must equal the max frequency (1000), running the tile on our identified critical path at highest frequency. In all other tiles, the equation produces a frequency lower than maximum, as their original q_{target}s are lower than q_{ref}.

Building on the basic example, consider if the example code (*MT* and child threads) has in turn been spawned by a parent thread *PT*. Here, the q_{target} passed to *MT+1* must not be *MT*'s, but rather *T(2)*'s—each thread passes to its parent the highest q_{target} among its siblings. By induction one can see that this holds all the way up to the original parent. Thus, one can trace the critical path through pair-wise parent-child communications. Once the maximum q_{target} is derived, it is then disseminated from each parent to its children.

3.1 Setting of stability constants K_i and K_p

From [15, 16, 19] we set the stability constants K_i and K_p to produce a maximum swing of 4-8% in frequency per interval. To achieve this, we assume that the load factor, or difference between (q_k - q_{ref}) and (q_k - q_{k-1}), is typically between 0 and 1000. Using the analysis in [19], Equation 7, we derive a range of stability constants. Constants from this range were then simulated through a set of benchmarks using *local-PID*. K_i=0.3 and K_p=0.1 were the best over the set of benchmarks. These are smaller than the default stability constants of K_i=0.6 and K_p=0.2 because in general the workload variation in CMPs tends to be shorter and sharper compared to MCDs, and thus we need to underdamp (relatively) the response. While different K_i and K_p may be better for *dist-PID*, for consistency both *local-PID* and *dist-PID* use the constants for *local-PID*.

3.2 Estimation of queue occupancies q_i

Our technique estimates the execution load on a tile in a CMP by monitoring the threads in the task queue of a tile. Each thread is associated with a load factor (a number ranging from 0 to 1000), provided by the compiler or programmer. Queue occupancies at each tile are then the summation of the load factors in the tile's task queue.

Load factors can be provided by programmer or the compiler. Using a performance macro-modeling tool [12, 13] for quicksort, a simple model based on the input argument size requiring a single multiply was sufficient to give a low 2% error in the run-time estimation of run time over various input distributions. For Othello, a linear model with a single multiplication led to just 6.8% error for random board configurations. Given that our technique requires only relative accuracy in thread run-time estimation these models can be further simplified so the processing overhead is negligible.

Simulator setup	
Processor clock	2-way, 1 GHz, 7 stage pipeline
Issue/Decode/Commit width	2/2/2 instructions per cycle
L1 D-cache Size	32KB, 4-way, 32B blocks, 1 cycle latency
L1 I-cache Size	32KB, 4-way, 32B blocks, 1 cycle latency
L2	None
Memory	20 cycles
Network topology	2-dimensional mesh
Channel Width and Flit Size	256-bit/256-bit
Router Pipeline	3 cycle latency (scouts), 1 cycle latency (others)
Link Traversal	1 cycle latency between hops
DVFS transition and setup delays	73.3ns/MHz, 171ns/2.86mV

Table 1: Architectural parameters.

3.3 Example Load Factors

We coded five multi-threaded benchmarks to evaluate *dist-PID*. Two are aggressively multithreaded kernels to stress *dist-PID* (recursive quicksort and Othello). Three others are SPEC [17] benchmarks (equake, twolf, and mcf) that are hand-partitioned.

Quicksort consists of threads created at each recursive invocation. Compared to the other benchmarks, quicksort creates threads 20 times more often, putting high thread pressure on the system. The load factor is based on the number of items to be sorted. Othello is a minimax tree search, where each move creates six or more new threads, creating significant variance in the load on each tile. Through profiling, each move has comparable (order of magnitude) complexity. Similarly, equake has two parallelized loops, which synchronize between loops, each thread having comparable complexity as well. Simplifying the linear model in [12], we predefine the load factors; 250 for each thread in Othello, 50 for each in equake.

In mcf, due to a large shared data structure, there is significant execution variability as a result of memory. In practice, however, threads are bimodal—either very small or very large. The load factor is set by profiling the tree traversal. If there exists a grandchild, the load factor is 750; otherwise, it is 0. For twolf we apply decoupled software pipelining [14] so producer threads prefetch nodes from memory, while consumer threads perform computation. The load factor is set by profiling the linked list traversal and multiplying the length by 100. These constant factors are required to set the load factors within the expected range of 0 to 1000.

4. RESULTS

For our results, we used a multiprocessor simulator based on XTREM [4]. XTREM is a validated SimpleScalar ARM [2] simulator, and our modifications added support for multiprocessing and networking. We model a 16-core chip multiprocessor with the architectural parameters given in Table 1. Threads are scheduled using a simple heuristic policy. First, look for free processors. If none are available, schedule onto the processor with the lightest load. $Q_{target}s$ is distributed via the on-chip network, with statistics transmission prioritized. Queue occupancy and load factor are sent every 2,500 cycles. The default DVFS interval was set at 50,000 cycles.

Dynamic power numbers for the CMP are obtained using Wattch [1] for the processor and memory components and Orion [18] for the network, running at a nominal voltage of 2.08V. Processors select frequencies between 100 MHz (0.45V) and 1 GHz (2.08V).

4.1 Energy-delay-product savings

To evaluate the efficiency of our proposed scheme, we evaluated three schemes, **baseline**, *local-PID*, and *dist-PID*. The baseline scheme is a processor with no DVFS, but "tile-gates" a tile when all the threads in a tile's task queue are stalled (waiting for other threads). We define tile-gating as shutting off the tile (and thus consuming no dynamic power). When a tile has no threads, it is also tile-gated. The interval length is 50,000 cycles. *Local-PID* uses a q_{ref} of 300, or one-third of the maximum expected queue occupancy, the default setting for q_{ref} in [19].

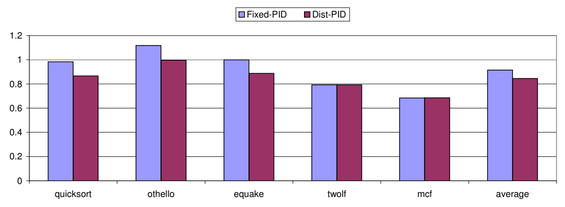

Figure 5: Normalized energy-delay product.

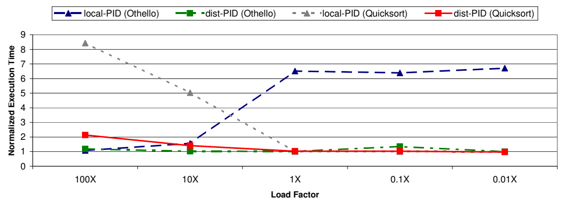

Figure 6: Run-time variations with varying load factor

Figure 5 shows the energy-delay product (EDP) for the five benchmarks. Overall the local scheme had 85.5% of the energy consumption of the baseline but increased run-time by 6.9%, producing an EDP of 97.3%. By comparison, our proposed scheme saved 20% of energy but increased run-time only 5.6%, giving an EDP of 84.55%.

Othello is problematic for both schemes. Othello tends to have long running threads punctuated by short outbursts where it launches many threads; this made it difficult to figure out the critical path. Quicksort on the other hand maps well to *dist-PID*, the size of the input arguments proved a good proxy for computation complexity.

4.2 Stability of DVFS control

Figure 6 shows the execution time (normalized against the baseline) after varying the load factor of Othello and quicksort from 100x to 0.01x for both schemes. *Local-PID* results are indicated with triangular markers and the *dist-PID* ones indicated with square one.

For both applications, *dist-PID* is relatively resilient toward load factor variation, remaining stable. *Local-PID*, however, is quite fragile. For Othello the performance of *local-PID* jumps quite suddenly at some point, when the load factor crosses over q_{ref}. Once the load factor consistently stays above q_{ref}, *local-PID* tries to preserve performance. If it is below q_{ref}, *local-PID* tries to save energy, without regard to the overall program state. This crossover point is not always foreseeable and not particularly predictable. Thus we see that distributed, coordinated control not only improves EDP, but ensures stability due to inter-tile coordination.

For quicksort, *local-PID* performs well when the load factors are overestimated—for example, when the list to be sorted is larger than what we tuned for. This tells the *local-PID* controller to try to preserve performance. When load factors were underestimated, though, performance skyrocketed—much more than in *dist-PID*. However, if the load factor is tuned for a large quicksort, a smaller one will take a massive hit in performance. Common sense would thus be to lean toward tuning for smaller quicksorts. Taken to the extreme, only very small quicksorts would be eligible for energy savings, and then we miss the point of DVFS.

4.3 Impact of Stability Constants

Figure 6 also indicates the impact of the stability constants, and whether stability constants rather than the distributed nature of *dist-PID* are responsible for the EDP improvement. Recall that in Equation 1 the stability constants K_i and K_p are multiplied by the queue occupancies and their load factors. Thus, a 10X increase in K_i and K_p is the same as a 10X increase in the load factors. From Figure 6 both schemes are never significantly better than the baseline; this indicates that the stability constants as chosen were fairly good. As both schemes used the same stability constants, we see that it is the distributed feature of *dist-PID* rather than the actual value of the stability constants that improves EDP. Thus, our results again show a compelling reason to have adaptable, coordinated, run-time DVFS.

5. CONCLUSIONS

We have shown that distributed, coordinated DVFS control is necessary to overcome the possibly counter-acting DVFS actions of local DVFS. Our proposal of *dist-PID* is shown to boost energy-performance on CMPs while ensuring the stability of DVFS control. Compared to *local-PID*, it achieves up to 8.8X improvement in EDP on benchmarks with substantial variance across the chip. We also show how *local-PID* can oscillate substantially for certain benchmarks, while our proposed *dist-PID* ensures stability.

As CMPs continue to be proposed and implemented, we believe that the techniques described in this paper can be used to extend on-line, formal approaches to DVFS from the uniprocessor realm into CMPs. Our approach is lightweight, requiring little extra hardware, and distributed, requiring little extra communication bandwidth. Overall, we believe our approach to be an effective way of improving DVFS for CMPs.

6. ACKNOWLEDGMENTS

We gratefully acknowledge the related simulator and research work by Julia Chen and Kevin Ko which added greatly to our understanding of the control issues. Our work has been supported in part by funding from NSF, Intel, IBM, and SRC.

7. REFERENCES

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architecture-Level Power Analysis and Optimizations. In *Proc. of ISCA27*, 2000.
[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, 1997.
[3] L. Clark. Circuit design of XScale (tm) microprocessors. *Proc. of 2001 Symposium on VLSI Circuits*, 2001.
[4] G. Contreras et al. XTREM: a power simulator for the Intel XScale (tm) core. *In Proc. of the 2004 ACM conference on Languages, compilers, and tools*, 2004.
[5] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *In Proc. of PLDI-2003*, 2003.
[6] Intel Corp. The Intel XScale Processor Architecture. *http://developer.intel.com/intelxscale*, 2002.
[7] Intel Corp. The Intel Pentium M Processor. *http://www.intel.com/design/mobile/pentiumm/documentation.htm*, 2004.
[8] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. *In Proc. of ISCA-29*, 2002.
[9] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithm with PACE. *In Proceedings of SIGMETRICS-2001*, 2001.
[10] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *In Proceedings of ISCA-30*, 2003.
[11] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. *Workshop on Complexity Effective Design*, 2000.
[12] A. Muttreja et al. Automated energy/performance macromodeling of embedded software. In *Proc. of the DAC*, 2004.
[13] A. Muttreja et al. Hybrid simulation for embedded software energy estimation. In *Proc. of the DAC*, 2005.
[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *Proc. of PACT-13*, 2004.
[15] G. Semeraro et al. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. *In Proc. of MICRO-35*, 2002.
[16] G. Semeraro et al. Energy efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. *In Proc. of HPCA-8*, 2002.
[17] The Standard Performance Evaluation Corporation. WWW Site. http://www.spec.org, Dec. 2000.
[18] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. *In Proc. of MICRO-35*, 2002.
[19] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *In Proc. of ASPLOS-XI*, 2004.
[20] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Event driven voltage and frequency control in multiple clock domain microprocessors. *In Proc. of HPCA-11*, 2005.
[21] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. *In Proceedings of PLDI-2003*, 2003.