Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture *

Greg Semeraro, David H. Albonesi,
Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas and Michael L. Scott
Department of Electrical and Computer Engineering and
Department of Computer Science
University of Rochester
Rochester, NY 14627

{semeraro,albonesi}@ece.rochester.edu
{dropsho,maglis,sandhya,scott}@cs.rochester.edu

Abstract

We describe the design, analysis, and performance of an on–line algorithm to dynamically control the frequency/voltage of a Multiple Clock Domain (MCD) microarchitecture. The MCD microarchitecture allows the frequency/voltage of microprocessor regions to be adjusted independently and dynamically, allowing energy savings when the frequency of some regions can be reduced without significantly impacting performance.

Our algorithm achieves on average a 19.0% reduction in Energy Per Instruction (EPI), a 3.2% increase in Cycles Per Instruction (CPI), a 16.7% improvement in Energy–Delay Product, and a Power Savings to Performance Degradation ratio of 4.6. Traditional frequency/voltage scaling techniques which apply reductions globally to a fully synchronous processor achieve a Power Savings to Performance Degradation ratio of only 2–3. Our Energy–Delay Product improvement is 85.5% of what has been achieved using an off–line algorithm. These results were achieved using a broad range of applications from the MediaBench, Olden, and Spec2000 benchmark suites using an algorithm we show to require minimal hardware resources.

1 Introduction

Microprocessor clock frequencies continue to rise at a rapid pace, *e.g.*, Intel presently ships the Pentium 4 at 2.8 GHz [8]. Transistor delays are reduced in each subsequent process generation at a much faster rate than wire delays. This disparity often affects the clock distribution network. Continuing to increase the clock frequency requires more and larger clock circuits [1, 24].

One solution to these obstacles is an MCD processor [15, 22] which allows parts of the processor (*i.e.*, domains) to be independently clocked, creating a globally–asynchronous, locally–synchronous (GALS) design. In an MCD design the chip is broken into smaller pieces, allowing traditional design techniques to be applied against substantially simpler problems. An MCD processor not only circumvents clock distribution obstacles, it introduces a new degree of freedom in energy conservation: the voltage and frequency for each domain may be independently and dynamically controlled. The difficulty is in determining when and how to change the frequency/voltage of the domains so as to minimize performance degradation and maximize energy reduction.

There are many techniques that can be used to control the domain frequencies. Off–line algorithms may be best suited for applications which can be hand–tuned, which will exhibit run–time characteristics similar to the off–line analysis, and which are likely to repeat this behavior whenever they are executed, *e.g.*, embedded systems where the environment and execution characteristics of the application are constrained and repeatable. On–line algorithms may be best suited for situations where the applications that will be executed are not known or not controllable. Since our existing off–line, optimistic algorithm [22] has the advantage of performing global analysis over the complete run of the application, it provides a benchmark against which to compare on–line algorithms.

We propose an on–line algorithm that reacts to the dynamic characteristics of the application to control domain frequencies. The algorithm uses processor queue utilization information and achieves a 16.7% improvement in

*This work was supported in part by NSF grants CCR–9701915, CCR–9702466, CCR–9705594, CCR–9811929, EIA–9972881, CCR–9988361, and EIA–0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; and by an external research grant from DEC/Compaq.

energy–delay product while degrading performance by only 3.2% *compared to a baseline MCD processor*; the improvement in energy–delay product is 85.5% of what has been achieved with the off–line algorithm. When taking into account the inherent inefficiencies of an MCD processor, the algorithm achieves a 13.8% improvement in energy–delay product while degrading performance by only 4.5% *compared to a conventional (fully synchronous) processor*.

The rest of this paper is organized as follows. In Section 2, we review the MCD microarchitecture. In Section 3, we describe our algorithm to control the domain frequencies and the required circuitry. In Section 4, we describe the simulation methodology employed to evaluate this algorithm and the benchmark applications used. In Section 5, we present results of this work. Sections 6 and 7 contain additional discussion of related work and concluding remarks.

2 Multiple Clock Domain Microarchitecture

Matzke has estimated that as technology scales down to a $0.1\mu m$ feature size, only 16% of the die will be reachable within a single 1.2GHz clock cycle [19]. Assuming a chip multiprocessor with two processors per die, each processor would need to have a minimum of three equal–size clock domains. The Multiple Clock Domain (MCD) architecture proposed by Semeraro *et al.* [22] uses four domains, one of which includes the L2 cache, so that domains may vary somewhat in size and still be covered by a single independent clock, with the voltage in each domain independently controlled. The boundaries between domains were chosen where (a) there existed a queue structure that served to decouple different pipeline functions, and/or (b) there was little inter–function communication. The four chosen domains, shown in Figure 1, comprise the front end (including L1 instruction cache, branch prediction, rename, and dispatch); integer processing core (issue/execute); floating–point processing core (issue/execute); and load/store unit (including L1 data cache and unified L2 cache). The main memory can also be considered a separate clock domain since it is independently clocked but it is not controllable by the processor, *i.e.*, the frequency and voltage of the main memory remain constant at the maximum values. We believe that the final result is an evolutionary departure from a conventional processor and would result in a physically realizable floorplan for an MCD processor. Table 1 lists the MCD–specific processor configuration parameters.

The MCD processor provides the capability of independently configuring each domain to execute at frequency/voltage settings at or below the maximum values. This allows domains that are not executing instructions critical to performance to operate at a lower frequency, and consequently, energy to be saved. An MCD processor potentially offers better energy savings than the global voltage/frequency scaling of commercial processors such as Transmeta [9] and XScale [7] because of the finer grain approach. Simultaneously, an MCD processor can potentially degrade the performance of applications significantly if critical resources are slowed down. Also, an MCD processor has the disadvantage that inter–domain communication may incur a synchronization penalty (depending on the relative frequency and phase of the domain clocks). These synchronization penalties are fully modeled for all inter–domain communication within our MCD simulator. We assume the arbitration and synchronization circuits developed by Sjogren and Myers [23], which detect whether the source and destination clock edges are far enough apart such that a source–generated signal can be successfully clocked at the destination. Based on [23], we assume a synchronization window of 30% of the period of the highest frequency.

Table 1. MCD processor configuration parameters.

Parameter	Value(s)
Domain Voltage	0.65 V – 1.20 V
Domain Frequency	250 MHz – 1.0 GHz
Frequency Change Rate	49.1 ns/MHz [7]
Domain Clock Jitter	110ps, normally distributed about zero
Synchronization Window	30% of 1.0 GHz clock (300ps)

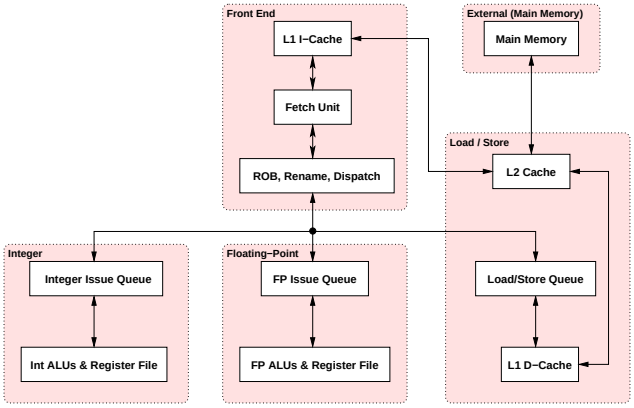


Figure 1. MCD processor block diagram.

Semeraro *et al.* [22] determined the inherent performance degradation of an MCD processor using architectural parameters modeled on the Alpha 21264, to be less than 4%. Our more recent experiments put that performance degradation at less than 2%. This difference is attributed to the fact that we have modified the simulator to include a more accurate clocking scheme which does not incur excess synchronization penalties due to integer rounding errors, as is the case in [22].

There is a potential performance improvement that can be achieved from an MCD processor which would likely outweigh the inherent performance degradation. That is, the maximum frequency of operation of all but that domain with the longest timing path would likely be higher for an

MCD processor than for the same design in a fully synchronous system. Similarly, a domain that performs best when clock frequency is traded off for IPC improvements would no longer be constrained to run at the faster overall frequency. This has the potential of increasing the per–domain performance relative to a fully synchronous chip. In addition, since the MCD processor does not need a low–skew global clock, the domain frequencies may be further increased. We have not attempted to exploit these potential improvements here, but these are areas of active research.

3 Dynamic Frequency Control Algorithm

To be effective, control of the frequency and voltage of each domain must be adaptive [15]. To accomplish this, the algorithm must take clues from the processor hardware to infer characteristics of the application that would allow the frequency of one or more domains to be decreased without significantly impacting performance.

Analysis of CPU structure resource utilization characteristics of many applications revealed that a correlation exists between the number of valid entries in the issue queues for each domain over an interval of instructions and the desired frequency for that domain (the desired frequencies were taken as the frequencies chosen by the off–line algorithm described in [22]). This observation is consistent with the conclusions of [4] where an issue queue utilization–based algorithm achieved similar results to a parallelism–based algorithm. These algorithms were used to dynamically resize the issue queue to achieve power efficiency. Although the parallelism–based algorithm takes into account where the instructions are being issued from within the queue, simply using the utilization of the queue was sufficient in most cases, indicating that issue queue utilization correlates well with instruction flow through the corresponding functional units. Intuitively, this correlation follows from considering the instruction processing core as the sink of the domain queue and the front end as the source. The queue utilization is indicative of the rate at which instructions are flowing through the processing core. If the utilization increases, instructions are not flowing through the processing units fast enough. This makes the queue utilization an appropriate metric for dynamically determining the appropriate frequency for each domain (except the front end domain).

This analysis also highlights the fact that decentralized control of each domain is possible. Since the benefits of an MCD architecture are in partitioning the processor, and global information needs to be synchronized to each domain, we decided that an algorithm that uses information generated locally in the domains is preferable. Given that the MCD architecture is partitioned in such a way that a queue exists on the input of each domain, other than the front end (see Figure 1), it seemed logical to exploit the existence of these queues and this correlation. (As discussed later, we use a fixed frequency for the front end). This correlation between issue queue utilization and desired frequency is not without problems. Notable among them is that as the frequency in a domain is changed it may have an impact on the issue queue utilization of that domain *and* possibly others. The interaction between the domains is a potential source of error that might cause the actual performance degradation to exceed the expected value and/or lead to lower energy savings. It is also possible that these interactions will cause perturbations in the domain frequencies as each domain is influenced by the changes in the other domains. These are issues we considered when designing the dynamic frequency control algorithm described next.

3.1 The Attack/Decay Algorithm

The algorithm developed consists of two components which act independently but cooperatively. The result is a frequency curve that approximates the envelope of the queue utilization curve, resulting in a small performance degradation and a significant energy savings. In general, an envelope detection algorithm reacts quickly to sudden changes in the input signal (queue utilization in this case), and in the absence of significant changes the algorithm slowly decreases the controlling parameter. This comprises a feedback control system. If the plant (the entity being controlled) and the control point (the parameter being adjusted) are linearly related, then the system will be stable and the control point will adjust to changes in the plant correctly. Due to the rapid adjustments required for significant changes in utilization and the slow adjustments otherwise, we call this an *Attack/Decay* algorithm. It is inspired by the Attack–Decay–Sustain–Release (ADSR) envelope generating techniques used in signal processing and signal synthesis [16]. The *Attack/Decay* algorithm is designed such that the loop delay (the time from when the control is changed until the effects of that change are seen on the feedback signal) is significantly smaller than the sampling period. The sampling period was chosen to be 10,000 instructions, which is approximately $10\times$ longer than the loop delay. In addition to ensuring that delay would not cause instability in the loop, we also accounted for the inaccuracy in the system (*i.e.*, the control point and the plant are *not* linearly related, they are highly correlated) by purposely *not* using the feedback signal to calculate a scaled error to be applied to the control point. Although a scaled error is typically used in control systems, in this case doing so would have almost certainly resulted in loop oscillations. Rather, we apply a fixed adjustment to the control point independent of the magnitude of the error. Although a fixed adjustment can result in a less than ideal envelope, it cannot produce excessive oscillation.

The implementation of the *Attack/Decay* algorithm is very simple. Listing 1 shows the code from the model. The algorithm requires the number of entries in the domain issue queue over the previous 10,000 instructions. Using that *QueueUtilization* and the *PrevQueueUtilization*, the algorithm is able to determine if there has been a significant change, in which case the *attack* mode is used (lines 10–21). If no such change has occurred or the domain is unused, the algorithm decreases the domain frequency slightly, in which case the *decay* mode is used (lines 22–27). In all cases, a *PerfDegThreshold* is examined when attempting to decrease the frequency (lines 19 & 25). If the IPC change exceeds this threshold, the frequency is left unchanged for that interval. The purpose is to catch natural decreases in performance that are unrelated to the domain frequency and prevent the algorithm from reacting to them. Changes in IPC related to frequency changes will cause queue utilization changes and subsequent frequency adjustments by the algorithm. The *PerfDegThreshold* also tends to minimize interaction with adjustments made in other domains. The IPC performance counter is the only global information provided to all domains. As an added measure against settling at a local minimum when a global minimum exists, the algorithm forces an attack whenever a domain frequency has been at one extreme or the other for 10 consecutive intervals (lines 4–9 & 38–47). This is a common technique applied when control systems reach an end point and the plant/control relationship is no longer defined. After the *Attack/Decay* algorithm is run, range checking is performed to force the frequency to be within the range of the MCD processor (not shown).

With exception of the Fetch/Dispatch domain, each domain operates independently using the same algorithm and configuration parameters. The Fetch/Dispatch domain behaves significantly differently than the others because it drives the remainder of the processor. We have found that decreasing the frequency of the front end causes a nearly linear performance degradation. For this reason, the results presented are with the front end frequency fixed at 1.0 GHz. In addition, the *Attack/Decay* algorithm operates on a queue structure at the front of the domain, and the Fetch/Dispatch domain does not have such a structure. An alternative algorithm would be needed for the Fetch/Dispatch domain. The range of parameter values used in our experiments are given in Table 2. The ranges for the parameters were chosen to be large enough to allow a sensitivity analyses (presented in Section 5) to be performed.

Figure 3(a) shows the Floating–Point Issue Queue (FIQ) utilization for the MediaBench application `epic decode`, and Figure 3(b) shows the frequency of the floating–point domain chosen by the *Attack/Decay* algorithm. This application is interesting because the floating–point unit is unused except for two distinct phases. The dynamic algorithm adapts accordingly, increasing the frequency during periods of activity and permitting it to decay when the domain is unused.

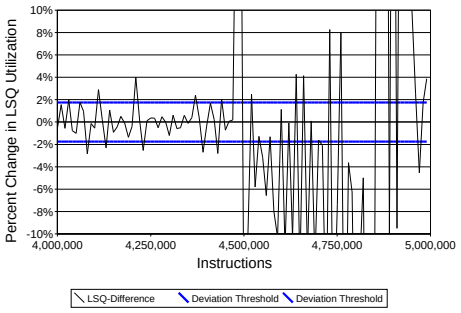
Table 2. *Attack/Decay* configuration parameters.

Algorithm Parameter	Range
DeviationThreshold	0–2.5%
ReactionChange	0.5–15.5%
Decay	0–2%
PerfDegThreshold	0–12%
EndstopCount	1–25 intervals

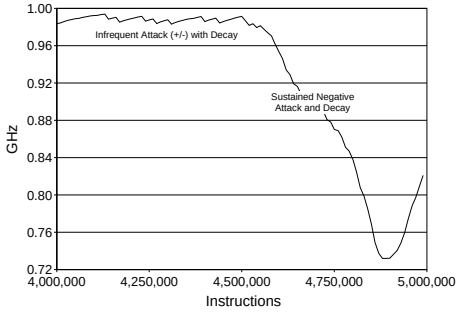
A similar relationship between utilization and frequency exists for the Load/Store domain. To understand this relationship it is useful to examine in detail the difference between Load/Store Queue (LSQ) utilization values in successive 10,000 instruction intervals (it is the difference that drives the *Attack/Decay* algorithm). A representative region exists in `epic decode` between 4–5M instructions. From Figure 2(a) and the load/store domain frequency shown in Figure 2(b), one can see that the domain frequency changes significantly when the utilization

```
1  /* Assume no frequency change required */
2  PeriodScaleFactor = 1.0;
3
4  if (UpperEndstopCounter == 10) {
5    /* Force frequency decrease */
6    PeriodScaleFactor = 1.0 + ReactionChange;
7  } else if (LowerEndstopCounter == 10) {
8    /* Force frequency increase */
9    PeriodScaleFactor = 1.0 - ReactionChange;
10 } else {
11   /* Check utilization difference against threshold */
12   if ((QueueUtilization - PrevQueueUtilization ) >
13      (PrevQueueUtilization * DeviationThreshold)) {
14     /* Significant increase since last time */
15     PeriodScaleFactor = 1.0 - ReactionChange;
16   } else
17     if (((PrevQueueUtilization  - QueueUtilization) >
18        (PrevQueueUtilization  * DeviationThreshold))
19        && ((PrevIPC / IPC) >= PerfDegThreshold)) {
20       /* Significant decrease since last time */
21       PeriodScaleFactor = 1.0 + ReactionChange;
22     } else {
23       /* The domain is not used or
24          no significant change detected... */
25       if ((PrevIPC / IPC) >= PerfDegThreshold) {
26         PeriodScaleFactor = 1.0 + Decay;
27       }
28     }
29 }
30
31 /* Apply frequency scale factor */
32 DomainFrequency = 1.0 / ((1.0 / DomainFrequency) *
33                          PeriodScaleFactor);
34
35 /* Setup for next interval */
36 PrevIPC = IPC;
37 PrevQueueUtilization = QueueUtilization;
38 if ((DomainFrequency <= MINIMUM_FREQUENCY) &&
39    (LowerEndstopCounter != 10))
40   LowerEndstopCounter++;
41 else
42   LowerEndstopCounter = 0;
43 if ((DomainFrequency >= MAXIMUM_FREQUENCY) &&
44    (UpperEndstopCounter != 10))
45   UpperEndstopCounter++;
46 else
47   UpperEndstopCounter = 0;
```

Listing 1. *Attack/Decay* algorithm

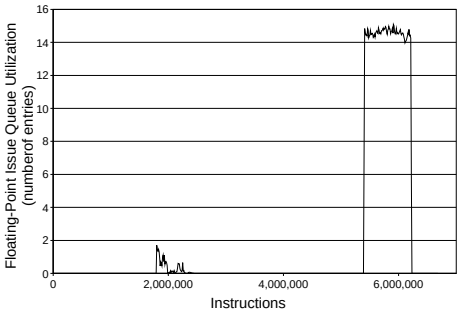


(a) Load/Store queue utilization differences.

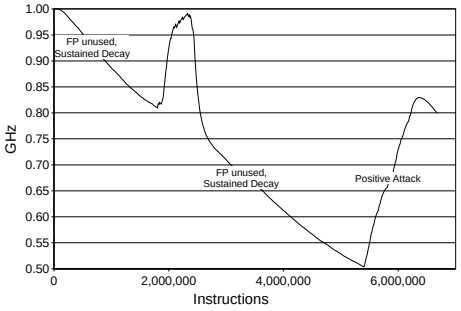


(b) Load/Store domain dynamic frequency.

Figure 2. Load/Store domain statistics for `epic decode`: 4–5M instructions.



(a) Floating–point issue queue utilization. The y–axis values are calculated as the average utilization for 10,000 instruction intervals. Since utilization is accumulated each cycle it is possible to show a utilization that exceeds the queue size. This occurs when the execution of the 10,000 instructions takes longer than 10,000 cycles.



(b) Floating–point domain dynamic frequency.

Figure 3. Floating point domain statistics for `epic decode`: 0–6.7M instructions.

difference exceeds the threshold (horizontal lines at $\pm$ 1.75% in Figure 2(a)). The first labeled region (Figure 2(b)) shows how the frequency is held constant through minor perturbations in queue utilization; the *attack* and *decay* modes counteract each other, keeping the frequency relatively stable. The second region shows the frequency actually being decreased in response to both the *attack* and *decay* modes of the algorithm.

3.2 Hardware Monitoring and Control Circuits

The hardware required to implement the *Attack/Decay* algorithm consists of several components. First, there must be an instruction counter to frame the intervals. Based on the general characteristics of the utilization curves and the decay rate range of the algorithm, we determined that an appropriate interval is approximately 10,000 instructions; therefore, a single 14–bit counter is sufficient. Second, the hardware must provide the queue counters that are used by the algorithm. These counters must exist for each domain that is being controlled and must be large enough to handle the worst case which is $QueueSize * IntervalLength * CPI$. It is not likely that the counters would ever reach this value but 15–16–bit saturating counters are not unreasonable. The potential error introduced if a counter were to saturate is negligible given the adaptive nature of the *Attack/Decay* algorithm.

The most complicated part of the hardware is the frequency computation logic. This logic must perform the following tasks:

- Calculate the difference between the present and the previous counter values (< 16–bit subtractor).
- Compare the differences with threshold values (<16–bit comparators in the worst case; simple masks may be possible).
- Determine the new domain frequency by scaling the present frequency by a constant value (16–24–bit multiplier, depending on the frequency resolution).

In addition, 4–bit counters are needed to determine when to force an attack when the frequency of the domain has been at one extreme or the other for 10 consecutive intervals. Clearly, scaling the domain frequency will require the most hardware, but since the result is not time–critical, it is possible to use a serial multiplication technique. From [27] it is possible to estimate the equivalent gate count for each of the above components. These estimates are given in Table 3 assuming 16–bit devices in all cases. For each do-

Table 3. Estimate of hardware resources requirement to implement the *Attack/Decay* algorithm.

Component	Estimation	Equivalent Gates
Queue Utilization Counter (Accumulator)	$7n$ (Adder) + $4n$ (D Flip–Flop) = $11n$	176
Comparators (2 required)	$6n \times 2 = 12n$	192
Multiplier (partial–product accumulation)	$1n$ (Multiplier) + $4n$ (D Flip–Flop) = $5n$	80
Interval Counter (14–bit)	$3n$ (Half–adder) + $4n$ (D Flip–Flop) = $7n$	112
Endstop Counter (4–bit)	$3n$ (Half–adder) + $4n$ (D Flip–Flop) = $7n$	28

Table 4. Architectural parameters for simulated Alpha 21264–like processor.

Configuration Parameter	Value
Branch predictor:	
Level 1	1024 entries, history 10
Level 2	1024 entries
Bimodal predictor size	1024
Combining predictor size	4096
BTB	4096 sets, 2–way
Branch Mispredict Penalty	7
Decode Width	4
Issue Width	6
Retire Width	11
L1 Data Cache	64KB, 2–way set associative
L1 Instruction Cache	64KB, 2–way set associative
L2 Unified Cache	1MB, direct mapped
L1 cache latency	2 cycles
L2 cache latency	12 cycles
Integer ALUs	4 + 1 mult/div unit
Floating–Point ALUs	2 + 1 mult/div/sqrt unit
Integer Issue Queue Size	20 entries
Floating–Point Issue Queue Size	15 entries
Load/Store Queue Size	64
Physical Register File Size	72 integer, 72 floating–point
Reorder Buffer Size	80

main, 476 gates are required (this includes full magnitude comparators even though simple masks may be sufficient). In addition, a single 14–bit interval counter would require 112 gates. Overall, fewer than 2,500 gates are required to fully control a four–domain MCD processor.

4 Simulation Methodology

Our simulation environment is based on the SimpleScalar toolset [3] with the Wattch [2] power estimation extension and the MCD processor extensions [22]. The MCD extensions include modifications to more closely model the microarchitecture of the Alpha 21264 microprocessor [18], *i.e.*, the Register Update Unit (RUU) has been split into separate reorder buffer (ROB), issue queue, and physical register file structures. A summary of our simulation parameters appears in Table 4.

We selected a broad mix of compute bound, memory bound, and multimedia applications from the MediaBench, Olden, and Spec2000 benchmark suites. Table 5 specifies the benchmarks along with the input data set and/or parameters used for the simulation runs. In addition, the instruction windows and total number of instructions for each benchmark are given. For all results reported, benchmarks for which multiple data sets were run and for which multiple programs comprise the benchmark, we compute an average result for that benchmark weighted by the instruction counts of the individual components.

Table 5. Benchmark applications, from the MediaBench, Olden and Spec2000 benchmark suites.

Benchmark	Suite	Datasets	Simulation window
`adpcm`	Media-Bench	ref	`encode` (6.6M)
			`decode` (5.5M)
`epic`		ref	`encode` (53M)
			`decode` (6.7M)
`jpeg`		ref	`compress` (15.5M)
			`decompress` (4.6M)
`g721`		ref	`encode` (0–200M)
			`decode` (0–200M)
`gsm`		ref	`encode` (0–200M)
			`decode` (0–74M)
`ghost-script`		ref	0–200M
`mesa`		ref	`mipmap` (44.7M)
			`osdemo` (7.6M)
			`osdemo` (75.8M)
`mpeg2`		ref	`encode` (0–171M)
			`decode` (0–200M)
`pegwit`		ref	`encrypt key` (12.3M)
			`encrypt` (32.4M)
			`decrypt` (17.7M)
`bh`	Olden	2048 1	0–200M
`bisort`		65000 0	Entire program (127M)
`em3d`		4000 10	70M–119M (49M)
`health`		4 1000 1	80M–127M (47M)
`mst`		1024 1	70M–170M (100M)
`perimeter`		12 1	0–200M
`power`		1 1	0–200M
`treeadd`		20 1	Entire program (189M)
`tsp`		100000 1	0–200M
`voronoi`		60000 1 0	0–200M
`bzip2`	Spec2000 Integer	source 58	1000M–1100M
`gcc`		166.i	2000M–2100M
`gzip`		source 60	1000M–1100M
`mcf`		ref	1000M–1100M
`parser`		ref	1000M–1100M
`vortex`		ref	1000M–1100M
`vpr`		ref	1000M–1100M
`art`	Spec2000 Floating-Point	ref	300M–400M
`equake`		ref	1000M–1100M
`mesa`		ref	1000M–1100M
`swim`		ref	1000M–1100M

For the baseline processor, we assume a 1.0 GHz clock and 1.2V supply voltage, based on that projected for the forthcoming CL010LP TSMC low–power 0.1μm process [25]. For configurations with dynamic voltage and frequency scaling, to approximate the smooth transition of

XScale [7] we use 320 frequency points spanning a linear range from 1.0 GHz down to 250 MHz. Corresponding to these frequency points is a linear voltage range from 1.2V down to 0.65V.[1] Our voltage range is tighter than that of XScale (1.65–0.75V), reflecting the compression of voltage ranges in future generations as supply voltages continue to be scaled aggressively relative to threshold voltages. In addition, the full frequency range is twice that of the full voltage range. These factors limit the amount of power savings that can be achieved with conventional (global) dynamic voltage and frequency scaling.

We use the XScale model for dynamic voltage and frequency scaling [7] because it allows the processor to execute through the frequency/voltage change. We believe this attribute to be essential to achieving significant energy savings without excessive performance degradation. We assume that the frequency change can be initiated immediately when transitioning to a lower frequency and voltage and that the voltage and frequency increase simultaneously when transitioning to higher values. In both cases, the voltage transition rate is defined by [7].

The disadvantage of multiple clock domains is that data generated in one domain and needed in another must cross a domain boundary, potentially incurring synchronization costs [5, 6]. To accurately model these costs, we account for the fact that the clocks driving each domain are independent by modeling independent jitter on a cycle–by–cycle basis. Our model assumes a normal distribution of jitter with a mean of zero. The variance is 110ps, consisting of an external Phase Lock Loop (PLL) jitter of 100ps (based on a survey of available ICs) and 10ps due to the internal PLL.

Our simulator tracks the relationships among the domain clocks on a cycle–by–cycle basis, based on their scaling factors and jitter values. Initially, all clock starting times are randomized. To determine the time of the next clock pulse in a domain, the domain cycle time is added to the starting time, and the jitter for that cycle (which may be positive or negative) is obtained from the distribution and added to this sum. By performing this calculation for all domains on a cycle–by–cycle basis, the relationship among all clock edges is tracked. In this way, we can accurately account for synchronization costs due to violations of the clock edge relationship or to inter–domain clock rate differences.

For all configurations, we assume that all circuits are clock gated when not in use. We do not currently estimate the power savings or clock frequency advantage (due to reduced skew) from the absence of a conventional global clock distribution tree that supplies a low–skew clock to all chip latches. In fact, we assume the MCD clock subsystem (separate domain PLLs, clock drivers and clock grid) *increases* the clock energy by 10% relative to a globally synchronous processor. This translates into a 2.9% increase in total energy for the MCD configurations over the single clock, fully synchronous system. We believe this is a very conservative assumption as others suggest that the use of multiple PLLs would increase the clock energy by an insignificant amount [11, 12].

[1] In Wattch, we simulate the effect of a 1.2–0.65V voltage range by using a range of 2.0–1.0833V because Wattch assumes a supply voltage of 2.0V.

5 Results

The results presented are for the *Attack/Decay* algorithm configured with *DeviationThreshold* = 1.75% *ReactionChange* = 6.0%, *Decay* = 0.175%, and *PerformanceDegradationThreshold* = 2.5%. In addition, results for the off–line algorithm from [22] are included to provide a comparison point. Results for the off–line algorithm are presented for two configurations: *Dynamic–1%* and *Dynamic–5%*. *Dynamic–1%* has performance degradation closest to *Attack/Decay* and therefore provides a useful comparison point. *Dynamic–5%* is the configuration reported in [22]; it achieves a higher energy–delay product improvement, but at a significantly higher performance degradation. These algorithms attempt to cap the performance degradation at 1% and 5% above that of the baseline MCD processor by finding slack in the application execution off–line, then configuring the domain frequencies dynamically (*i.e.*, during re–execution using the *same* input data set) to eliminate that slack.

The results (averaged over all 30 benchmark applications) and the analysis that follows include the inherent performance degradation and energy losses of the MCD processor (including the additional 2.9% energy for multiple PLL clock circuits). Note that these results do not include the energy consumed by the hardware resources required to implement the algorithm (described in Section 3.2). We believe this to be inconsequential given the modest hardware requirements relative to the complexity of the processor itself. Nor do these results include the power consumed by the synchronization circuits within the interface queues. Others [5, 6] have shown that multiple clock domain FIFOs do not add significantly to the energy consumed by a traditional synchronous FIFO.

The *Attack/Decay* algorithm achieves an energy–delay product improvement of 13.8% with an average energy per instruction reduction of 17.5% and a performance degradation of 4.5% relative to a *conventional, fully synchronous processor*. The performance degradation goal was 2.5% for this configuration (with 1.3% from the inherent performance degradation of the MCD architecture). The *Attack/Decay* algorithm also compares well to the *Dynamic–1%* algorithm described in [22] (Table 6, results shown are *relative to a baseline MCD processor*). Results for the *Dynamic–5%* algorithm are also shown, but since the per-

Table 6. Comparison of *Attack/Decay*, *Dynamic–1%* and *Dynamic–5%* algorithms, relative to a baseline MCD processor. The *Global(···)* results are for a fully synchronous processor using global frequency/voltage scaling to achieve the performance degradation of the respective algorithms. Note the power/performance ratio of approximately 2 for the *Global(···)* results.

Algorithm	Performance Degradation	Energy Savings	Energy–Delay Product Improvement	Power / Performance Ratio
Attack/Decay	3.2%	19.0%	16.7%	4.6
Dynamic–1%	3.4%	21.9%	19.6%	5.1
Dynamic–5%	8.7%	33.0%	27.5%	3.8
Global (Attack/Decay)	3.2%	6.5%	7.8%	2.0
Global (Dynamic–1%)	3.4%	6.6%	3.6%	2.0
Global (Dynamic–5%)	8.7%	12.4%	5.0%	1.9

formance degradation is significantly higher, it is difficult to make a comparison. Note that the off–line algorithm is able to take advantage of full knowledge of the slack existing within the application. In addition, the off–line algorithm is able to request frequency changes prior to the point in the program execution where that frequency is required, and therefore the slew rate of the voltage/frequency changing hardware does not create a source of error within that algorithm. The on–line algorithm is not able to take advantage of either of these and is by nature reactive. Still, it is able to achieve 85.5% of the energy–delay product improvement of the *Dynamic–1%* off–line algorithm.

The power savings to performance degradation ratio (calculated as the average percent power savings divided by the average percent performance degradation) is meant to express the relative benefit (in terms of power savings) for each percentage of performance degradation [21]. In other words, a ratio of X indicates that for every 1 percent of performance degradation, X percent of power is saved. The *Attack/Decay* power savings to performance degradation ratio achieved was 4.6. This compares favorably with a ratio of 2–3 for global voltage scaling techniques. Our analysis of the global voltage scaling technique using realistic frequency/voltage scaling [7, 9] resulted in a ratio of 2 (Table 6, based on the simulator model and benchmark applications described in Section 4). Others [21] suggest that a ratio of 3 can be achieved with global voltage scaling.

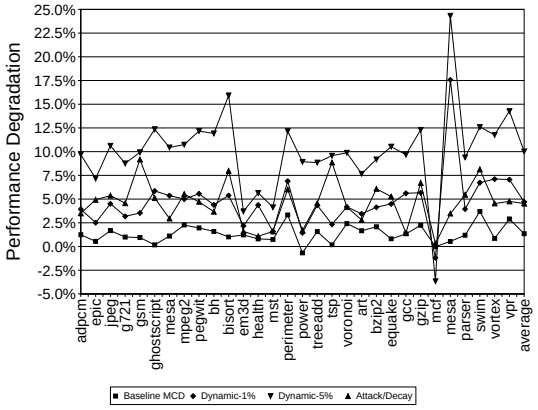
The individual application results for performance degradation, energy savings and energy–delay product are given in Figures 4(a), 4(b) and 4(c), respectively.[2] All of these results are referenced to a fully synchronous processor. One observation from Figure 4(a) is that the off–line algorithm is able to achieve a *negative* performance degradation, *i.e.*, a performance improvement, for the `mcf` benchmark. The gains are modest (-1.2% and -3.7% for the *Dynamic–1%* and *Dynamic–5%* algorithms, respectively), but somewhat surprising. Analysis of per–interval processor statistics reveals a subtle interplay between the timing of loads that miss all the way to main memory and penalties associated with branch mis–predictions. In the case of *Dynamic–5%* the MCD architecture results in an *increase* in average memory access time of 2.8%. The change in instruction timing resulting from this causes a *decrease* in average branch mis–predict penalty of 15.5%, resulting in an overall performance improvement of 3.7%. This same anomaly exists for the *Dynamic–1%* algorithm, although to a lesser degree. The *Attack/Decay* algorithm achieves a performance degradation of 0.3% for `mcf`; analysis of the per–interval statistics for this algorithm shows that it does not exhibit the interaction between main memory latency and branch mis–predict penalties. This is because the *Attack/Decay* algorithm never decreases the frequency of the Load/Store domain below the maximum value in `mcf`, and therefore the instruction timing is not perturbed as it is for the off–line algorithms.

Originally we used 1.0–1.1B as the instruction window for `gcc`. We decided to change to 2.0–2.1B because we determined that 1.0–1.1B was part of the initialization phase of the program. Although the 1.0–1.1B instruction window *is* part of the initialization phase, it does have some interesting properties as the following analysis illustrates: During this phase of execution, 80% of the instructions are memory references all of which miss all the way to main memory. With the load/store–to–main memory interface getting this highly utilized, the increased memory access time directly impacts the MCD baseline performance degradation. This is similar to what happens to `mcf` for the off–line algorithm except that the change in memory timing due to the synchronization penalty *does not* cause a decrease in branch mis–predict penalty for `gcc` as it does in `mcf`. This is because the branches in `gcc` have no mis–predict penalty (we are getting 99% branch prediction accuracy for the 1.0–1.1B instruction window of `gcc`, and about 84% for `mcf`).

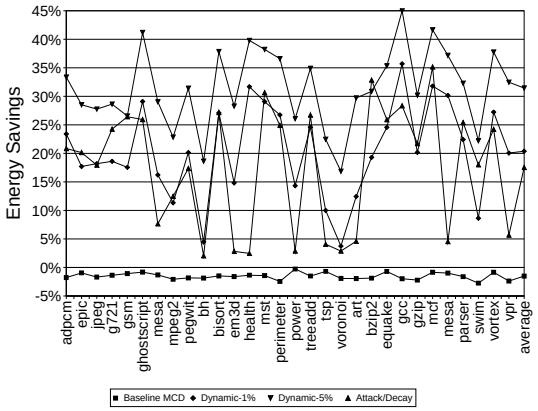
In addition to the individual results, it is interesting to examine the sensitivity of the algorithm to the configuration parameters.[3] The sensitivity results are for reasonable algo-

[2] Lines on the graph are meant to ease understanding of the data grouping and not meant to imply any relationship between data points.

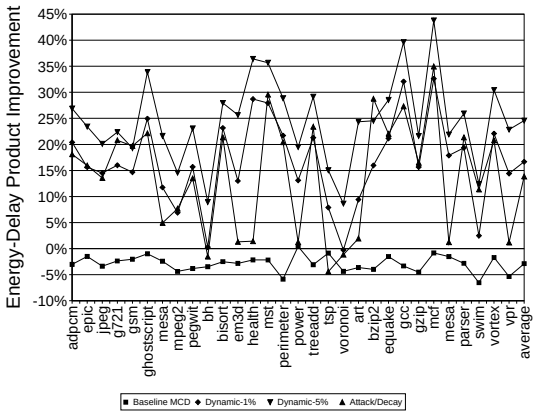
[3] The Attack/Decay algorithm is specified in the legends by four per-



(a) Performance Degradation.



(b) Energy Savings.

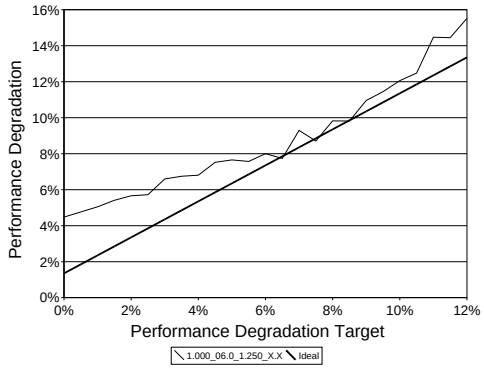


(c) Energy–Delay Product Improvement.

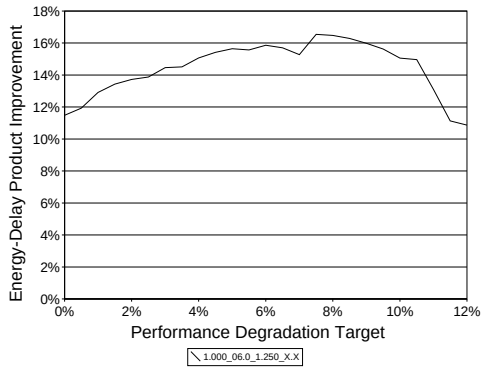
Figure 4. Application result summary.

rithm parameter values where individual parameters were swept through the range specified in Table 2. Figure 5(a) shows the average performance degradation achieved by the algorithm, versus the target performance degradation. In

centage parameters: *DeviationThreshold*, *ReactionChange*, *Decay* and *PerformanceDegradationThreshold*, respectively. *X.XX* is used as a placeholder for swept parameters.



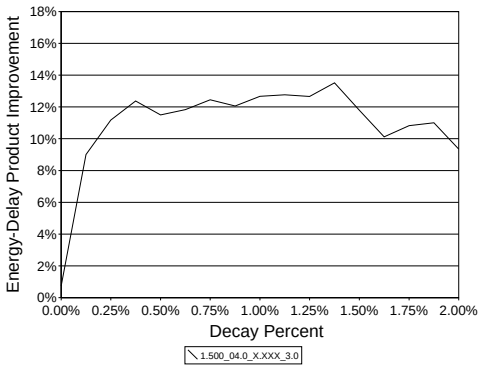
(a) Performance degradation accuracy, offset by the baseline MCD performance degradation of 1.3%.



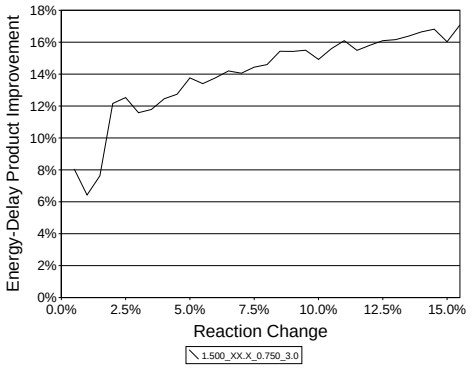
(b) Performance degradation target trend: Energy–delay product improvement.

Figure 5. *Attack/Decay* performance degradation target analysis.

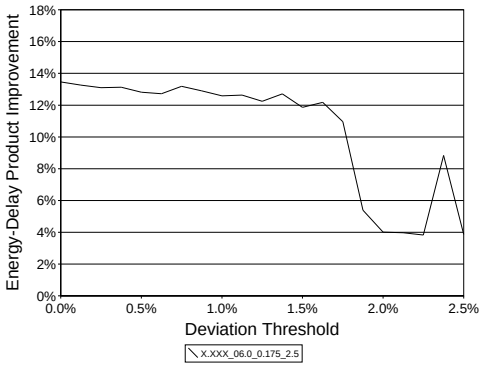
this figure, the bold line indicates the performance degradation of an 'ideal' algorithm, *i.e.*, a fictitious algorithm which achieves the precise performance degradation requested. (Note that performance degradations of all algorithms are offset by the inherent MCD performance degradation of 1.3%.) The *Attack/Decay* algorithm is able to provide approximately the performance degradation specified by the configuration, over the range of 4–10%. Figure 5(b) shows energy–delay product improvement versus the performance degradation target. The trend in this graph is that as the performance degradation target increases beyond approximately 9%, the energy–delay product improvement tends to decrease. This is consistent with the characteristics of the *Attack/Decay* algorithm. The algorithm attempts to decrease frequencies only as much as is possible without impacting performance. As performance degradation is increased in an attempt to save more energy, the energy savings come at the expense of more and more performance degradation. In other words, for small performance



(a) *DecayPercent* sensitivity.

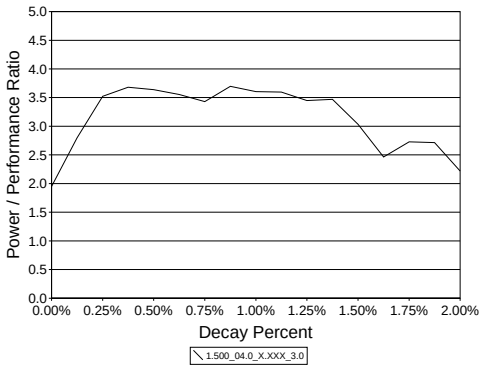


(b) *ReactionChangePercent* sensitivity.

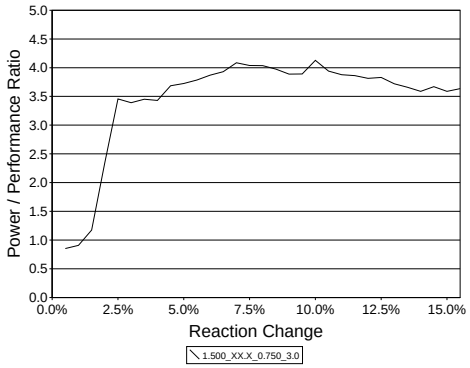


(c) *DeviationThresholdPercent* sensitivity.

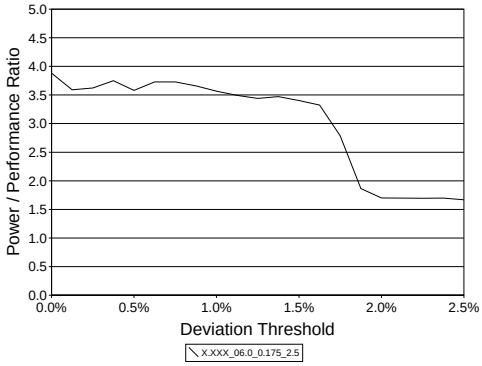
Figure 6. *Attack/Decay* sensitivity analysis: Energy–Delay Product Improvements.



(a) *DecayPercent* sensitivity.



(b) *ReactionChangePercent* sensitivity.



(c) *DeviationThresholdPercent* sensitivity.

Figure 7. *Attack/Decay* sensitivity analysis: Power/Performance Ratio.

degradation values, the energy savings come quickly; for larger performance degradation values, the energy savings are harder to achieve.

Figure 6(a) and 7(a) show the algorithm sensitivity to *DecayPercent* for the energy–delay product and power/performance ratio, respectively. We can see that for small and large values of *DecayPercent*, the performance of the algorithm diminishes. It is encouraging that there is a rather large range (from 0.5% to 1.5%) where the algorithm performance is essentially flat, and thus relatively insensitive to this parameter. This behavior follows from the effect that the *DecayPercent* parameter has on the *Attack/Decay* algorithm. The *DecayPercent* controls how quickly the algorithm reduces the frequency when queue utilization is essentially unchanged. If the decay is too slow, potential energy savings are not realized, too fast and performance is

degraded excessively.

Figure 6(b) and 7(b) show the algorithm sensitivity to *ReactionChangePercent*. There is a large range of values for the this parameter (from 3% to 12%) which produce the best results, higher values continue to show good results but the incremental improvements diminish quickly. Again, this behavior follows from the effect the *ReactionChangePercent* has on the *Attack/Decay* algorithm. If the value of this parameter is too small, the algorithm is unable to quickly adapt to the needs of the application, too large and overshooting and undershooting are likely to occur. Both of these extremes can result in excess performance degradation and/or unrealized energy savings.

Figure 6(c) and 7(c) show the algorithm sensitivity to *DeviationThresholdPercent*. Although it might appear that values less than 1.75% perform equally well, it is important to understand how this parameter influences the *Attack/Decay* algorithm. This parameter determines the sensitivity of the algorithm to changes in queue utilization. Small values of *DeviationThresholdPercent* will cause the *Attack/Decay* algorithm to enter the *attack* mode excessively. Although the net result shows that the performance is still good, this is not a desirable condition since excessive energy may be consumed as the PLL and voltage control circuits are continuously activated. For these reasons, the *DeviationThresholdPercent* should be kept within the range of 0.75–1.75%.

We also performed a sensitivity analysis on the *EndstopCount* parameter. The *Attack/Decay* algorithm was insensitive to this parameter between the values of 2 and 25, although an infinite value *did* degrade the effectiveness of the algorithm.

In summary, the *Attack/Decay* algorithm exhibits the expected parameter sensitivity; the performance of the algorithm diminishes for both large and small values of the parameters. In addition, the extreme values of the parameters cause wider variation in the overall and individual application results. It is also clear that there exists a large parameter space over which the algorithm is relatively insensitive. This is encouraging since it allows significant latitude in the implementation of the algorithm. The expected results over this parameter space exhibit the combination of low performance degradation and high energy savings required by an on–line algorithm.

6 Related Work

To the best of our knowledge, this is the first work to describe an on–line dynamic algorithm to control the frequency and voltage for domains in an MCD architecture (although using signal processing techniques in microarchitecture is not without precedent [17]). In [15] the authors conclude that a multiple clock domain processor can only reduce energy over a fully synchronous processor if the frequency/voltage are dynamically controlled, but they do not describe any control algorithms.

There has been significant research related to controlling the frequency and voltage for a globally controllable processor, *i.e.*, Dynamic Voltage Scaling (DVS). Weiser *et al.* [26] and Govil *et al.* [10] describe algorithms developed for DVS systems which assume that the frequency of the CPU is controllable via software, that the voltage changes to the minimum value possible for the chosen frequency, and that the frequency and voltage changes occur instantaneously. The algorithms developed use the operating system to execute the DVS algorithm at either normal scheduling points [26] or at prescribed time intervals [10]. In [20] the authors adapt traditional DVS algorithms to the real–time systems domain by adding deadline information available to the real–time operating system as part of the DVS algorithm. In [13] and [14] voltage scheduling instructions are inserted by the compiler when the compiler is able to determine that processor frequency will not impact overall application execution performance.

7 Conclusions

We have described and evaluated an on–line, dynamic frequency control algorithm for an MCD processor that uses a simple *Attack/Decay* technique to reduce the frequency/voltage of individual domains, resulting in energy per instruction savings of 19.0% and an improvement in energy–delay product of 16.7% with only 3.2% degradation in performance, referenced to a *baseline MCD processor with all domain frequencies of 1.0 GHz*. These results were achieved over a broad range of applications taken from standard benchmark suites. The algorithm is also able to maintain consistent performance levels across these vastly different applications (multimedia, integer, and floating–point) using only modest hardware resources, achieving 85.5% of the energy–delay product improvements of our previous, off–line algorithm.

Whereas traditional global voltage scaling techniques are able to achieve a power savings to performance degradation ratio of only 2–3, the *Attack/Decay* algorithm coupled with an MCD microarchitecture achieves a ratio of 4.6.

Our current analysis shows that this algorithm is feasible and can produce significant energy savings with minimal performance degradation. Future work will involve developing alternative on–line algorithms, including approaches for effective scaling of the front end. Another area of future research involves the microarchitecture of an MCD processor and the combination of on–line and off–line algorithms. It may be possible to change domain boundaries and/or the number or size of processor resources such that the frequency of one or more domains could be further reduced.

References

[1] D. W. Bailey and B. J. Benschneider. Clocking Design and Analysis for a 600-MHz Alpha Microprocessor. *Journal of Solid-State Circuits*, 36(11):1627–1633, Nov. 1998.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[3] D. Burger and T. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, Wisconsin, June 1997.

[4] A. Buyuktosunoglu, D. H. Albonesi, S. E. Schuster, D. M. Brooks, P. Bose, and P. W. Cook. Power Efficient Issue Queue Design. In *Power Aware Computing*, pages 37–60. Kluwer Academic Publishers, 2002.

[5] T. Chelcea and S. M. Nowick. A Low-Latency FIFO for Mixed-Clock Systems. In *IEEE Computer Society Annual Workshop on VLSI (WVLSI)*, Orlando, Florida, Apr. 2000.

[6] T. Chelcea and S. M. Nowick. Low-Latency Asynchronous FIFO's Using Token Rings. In *Proceedings of the 6th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async-00)*, Eilat, Israel, Apr. 2000.

[7] L. T. Clark. Circuit Design of XScale^{TM} Microprocessors. In *2001 Symposium on VLSI Circuits, Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits*. IEEE Solid-State Circuits Society, June 2001.

[8] I. Corp. Intel Pentium 4 Processor. http://www.intel.com/-products/desk_lap/processors/desktop/pentium4/, 2002.

[9] M. Fleischmann. Longrun^{TM} power management. Technical report, Transmeta Corp., Jan. 2001.

[10] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. In *Proceedings of 1st ACM International Conference on Mobile Computing and Networking*. ACM, 1995.

[11] V. Gutnik. Personal communication, May 2002.

[12] V. Gutnik and A. Chandrakasan. Active GHz Clock Network using Distributed PLLs. In *IEEE Journal of Solid-State Circuits*, pages 1153–1559, Nov. 2000.

[13] C. Hsu and U. Kremer. Compiler–Directed Dynamic Voltage Scaling Based on Program Regions. Technical Report DCS-TR-461, Rutgers University, Nov. 2001.

[14] C. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems, in conjunction with the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Nov. 2000.

[15] A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, Anchorage, Alaska, May 2002.

[16] K. Jensen. Envelope Model of Isolated Musical Sounds. In *Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects (DAFx99)*, Dec. 1999.

[17] M. K., P. Stenström, and M. Dubois. The FAB Predictor: Using Fourier Analysis to Predict the Outcome of Conditional Branches. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 223–232. IEEE Computer Society, Feb. 2002.

[18] R. E. Kessler, E. McLellan, and D. Webb. Circuit Implementation of a 600 MHz Superscalar RISC Microprocessor. In *Proceedings of the International Conference on Computer Design*, Austin, Texas, Oct. 1998. IEEE Computer Society.

[19] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, Sept. 1997.

[20] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[21] R. Ronen. Power: The Next Frontier. In *Proceedings of the Workshop on Power-Aware Computing Systems, in conjunction with the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, Feb. 2002. Keynote Address.

[22] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 29–40. IEEE Computer Society, Feb. 2002.

[23] A. E. Sjogren and C. J. Myers. Interfacing Synchronous and Asynchronous Modules Within A High-Speed Pipeline. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 47–61, Ann Arbor, Michigan, Sept. 1997.

[24] S. Tam, S. Rusu, U. Desai, R. Kim, J. Zhang, and I. Young. Clock Generation and Distribution for the First IA-64 Microprocessor. *IEEE Journal of Solid State Circuits*, 35(11):1545–1552, Nov. 2000.

[25] TSMC Corp. TSMC Technology Roadmap. http://www.-tsmc.com, July 2001.

[26] M. Weiser, A. Demers, B. Welch, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, CA, Nov. 1994. USENIX Assoc.

[27] R. Zimmermann. Computer Arithmetic: Principles, Architectures, and VLSI Design. Personal publication (http://www.iis.ee.ethz.ch/∼zimmi/publications/-comp_arith_notes.ps.gz), Mar. 1999.