Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor[*]

Grigorios Magklis[†], Michael L. Scott[†], Greg Semeraro[‡], David H. Albonesi[‡], and Steven Dropsho[†]

[†] Department of Computer Science
[‡] Department of Electrical and Computer Engineering
University of Rochester, Rochester, NY 14627

Abstract

A Multiple Clock Domain (MCD) *processor addresses the challenges of clock distribution and power dissipation by dividing a chip into several (coarse-grained) clock domains, allowing frequency and voltage to be reduced in domains that are not currently on the application's critical path. Given a reconfiguration mechanism capable of choosing appropriate times and values for voltage/frequency scaling, an MCD processor has the potential to achieve significant energy savings with low performance degradation.*

Early work on MCD processors evaluated the potential for energy savings by manually inserting reconfiguration instructions into applications, or by employing an oracle driven by off-line analysis of (identical) prior program runs. Subsequent work developed a hardware-based on-line mechanism that averages 75–85% of the energy-delay improvement achieved via off-line analysis.

In this paper we consider the automatic *insertion of reconfiguration instructions into applications, using profile-driven binary rewriting. Profile-based reconfiguration introduces the need for "training runs" prior to production use of a given application, but avoids the hardware complexity of on-line reconfiguration. It also has the potential to yield significantly greater energy savings. Experimental results (training on small data sets and then running on larger, alternative data sets) indicate that the profile-driven approach is more stable than hardware-based reconfiguration, and yields virtually all of the energy-delay improvement achieved via off-line analysis.*

1. Introduction

The ongoing push for higher processor performance has led to a dramatic increase in clock frequencies in recent years. The Pentium 4 microprocessor is currently shipping at 3.06 GHz [27], and is designed to throttle back execution when power dissipation reaches 81.8W. At the same time, technology feature sizes continue to decrease, and the number of transistors in the processor core continues to increase. As this trend toward larger and faster chips continues, designers are faced with several major challenges, including global clock distribution and power dissipation.

It has been suggested that purely asynchronous systems have the potential for higher performance and lower power compared to fully synchronous systems. Unfortunately, CAD and validation tools for such designs are not yet ready for industrial use. An alternative, which addresses all of the above issues, is a *Globally Asynchronous Locally Synchronous* (GALS) system [9]. In previous work we have proposed a GALS system known as a *Multiple Clock Domain* (MCD) processor [30]. Each processor region (domain) is internally synchronous, but domains operate asynchronously of one another. Existing synchronous design techniques can be applied to each domain, but global clock skew constraints are lifted. Moreover, domains can be given independent voltage and frequency control, enabling dynamic voltage scaling at the level of domains.

Global dynamic voltage scaling already appears in many systems, notably those based on the Intel XScale [25] and Transmeta Crusoe [17] processors, and can lead to significant reductions in energy consumption and power dissipation for rate-based and partially idle workloads. The advantage of an MCD architecture is its ability to save energy and power even during "flat out" computation, by slowing domains that are comparatively unimportant to the application's critical path, even when those domains cannot be gated off completely.

The obvious disadvantage is the need for inter-domain synchronization, which incurs a baseline performance penalty—and resulting energy penalty—relative to a globally synchronous design. We have quantified the performance penalty at approximately 1.3% [29]. Iyer and Marculescu [23] report a higher number, due to a less precise estimate of synchronization costs. Both studies confirm that an MCD design has a significant potential energy *advantage*, with only modest performance cost, if the frequencies and voltages of the various domains are set to appropriate values at appropriate times. The challenge is to find an effective mechanism to identify those values and times.

[*] This work was supported in part by NSF grants CCR–9701915, CCR–9702466, CCR–9811929, CCR–9988361, CCR–0204344, and EIA–0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; by an IBM Faculty Partnership Award; and by equipment grants from IBM, Intel, and Compaq.

As with most non-trivial control problems, an optimal solution requires future knowledge, and is therefore infeasible. In a recent paper, we proposed an on-line *attack-decay* algorithm that exploits the tendency of the future to resemble the recent past [29]. Averaged across a broad range of benchmarks, this algorithm achieved overall energy×delay improvement of approximately 85% of that possible with perfect future knowledge.

Though a hardware implementation of the attack-decay algorithm is relatively simple (fewer than 2500 gates), it nonetheless seems desirable to find a software alternative, both to keep the hardware simple and to allow different control algorithms to be used at different times, or for different applications. It is also clearly desirable to close the energy×delay gap between the on-line and off-line (future knowledge) mechanisms. In this paper we address these goals through *profile-driven* reconfiguration.

For many applications, we can obtain a better prediction of the behavior of an upcoming execution phase by studying the behavior of similar phases in prior runs than we can from either static program structure or the behavior of recent phases in the current run. The basic idea in profile-driven reconfiguration is to identify phases in profiling runs for which certain voltage and frequency settings would be profitable, and then to modify the application binary to recognize those same phases when they occur in production runs, scaling voltages and frequencies as appropriate.

Following Huang [21], we assume that program phases will often be delimited by subroutine calls. We also consider the possibility that they will correspond to loop nests within long-running subroutines. We use traditional profiling techniques during training runs to identify subroutines and loop nests that make contributions to program runtime at a granularity for which MCD reconfiguration might be appropriate (long enough that a frequency/voltage change would have a chance to “settle in” and have a potential impact; not so long as to suggest a second change). We then employ off-line analysis to choose MCD settings for those blocks of code, and modify subroutine prologues/epilogues and loop headers/footers in the application code to effect the chosen settings during production runs.

Following Hunt [22], we accommodate programs in which subroutines behave differently when called in different contexts by optionally tracking subroutine call chains, possibly distinguishing among call sites within a given subroutine as well. Experimentation with a range of options (Section 4) suggests that most programs do not benefit significantly from this additional sophistication.

The rest of the paper is organized as follows. In Section 2, we briefly review the MCD microarchitecture. In Section 3, we describe our profiling and instrumentation infrastructure, and discuss alternative ways to identify corresponding phases of training and production runs. Performance and energy results appear in Section 4. Additional discussion of related work appears in Section 5. Section 6 summarizes our conclusions.

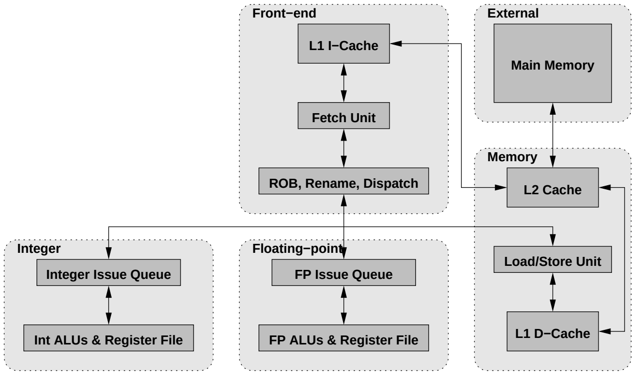


Figure 1. MCD processor block diagram.

2. Multiple Clock Domain Microarchitecture

In our study, we use the MCD processor proposed in [30]. The architecture consists of four different on-chip clock domains (Figure 1) for which frequency and voltage can be controlled independently. In choosing the boundaries between them, an attempt was made to identify points where (a) there already existed a queue structure that served to decouple different pipeline functions, or (b) there was relatively little inter-function communication. Main memory is external to the processor and for our purposes can be viewed as another, fifth, domain that always runs at full speed.

The disadvantage of an MCD design is the need for synchronization when information crosses the boundary between domains. The synchronization circuit is based on the work of Sjogren and Myers [31]. It imposes a delay of one cycle in the consumer domain whenever the distance between the edges of the two clocks is within 30% of the period of the faster clock. Our simulator, derived from the one used in [30], includes a detailed model of the synchronization circuit, including randomization caused by jitter.

For the baseline processor, we assume a 1GHz clock and 1.2V supply voltage. We also assume a model of frequency and voltage scaling based on the behavior of the Intel XScale processor [11], but with a tighter voltage range, reflecting expected compression in future processor generations. A running program initiates a reconfiguration by writing to a special control register. The write incurs no idle-time penalty: the processor continues to execute through the voltage/frequency change. There is, however, a delay before the change becomes fully effective. Traversing the entire voltage range requires $55\mu s$. Given this transition time, changes will clearly be profitable only if performed at a granularity measured in thousands of cycles.

3. Application Analysis

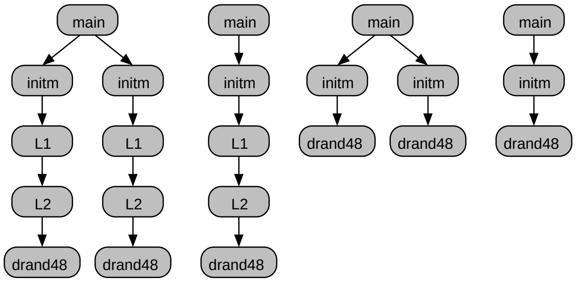
Our profile-based control algorithm can be divided into four phases, which we discuss in the four subsections below. Phase one performs conventional performance profiling to identify subroutines and loop nests that are of appropriate length to justify reconfiguration. Phase two constructs a DAG that represents dependences among primitive operations in the processor and then applies a “shaker” algorithm to distribute the *slack* in the graph in a way that minimizes energy [30]. Phase three uses per-domain histograms of primitive operation frequencies to identify, for each chosen subroutine or loop nest, the minimum frequency for each domain that would, with high probability, allow execution to complete within a fixed slowdown bound. Phase four edits the application binary to embed path-tracking or reconfiguration instructions at the beginnings and ends of appropriate subroutines and loop nests.

3.1. Choosing Reconfiguration Points

Changing the domain voltage takes several microseconds and thus is profitable only over intervals measured in thousands of instructions. Our goal is therefore to find the boundaries between major application phases. While our previous off-line [30] and attack-decay [29] algorithms make reconfiguration decisions at fixed intervals, regardless of program structure, it is clear that program phases correspond in practice to subroutine calls and loops [21], and that the boundaries of these structures are the natural points at which to instrument program binaries for the purpose of reconfiguration.

Phase one of our profile-driven reconfiguration mechanism uses the ATOM binary instrumentation package [13] to instrument subroutines and loops. Because we are working with binaries, loops are defined as the strongly connected components of subroutine control flow graphs; they may or may not correspond to loop constructs in the source code. When executed, the instrumented binary counts the number of times that each subroutine or loop is executed in a given “context”. In an attempt to evaluate the tradeoff between reconfiguration quality and the overhead of instrumentation to be inserted in phase four, we consider several different definitions of context. The most general employs a *call tree* reminiscent of the *calling context tree (CCT)* of Ammons et al. [1]. Each node of the call tree is a subroutine or loop in context. The path from the root of the tree to a node captures the callers *and call sites* on the path back to `main` at the time the subroutine or loop was entered.

The call tree differs from the static call graph constructed by many compilers in that it has a separate node for every path over which a given subroutine can be reached (it will also be missing any nodes that were not encountered during the profiling run). At the same time, the call tree is not a true dynamic call trace, but a compressed one, where multiple instances of the same path are superimposed. For example if a subroutine is called from inside a loop it will have the same call history every time, and thus it will be represented by a single node in the tree, even though it may actually have been called many times. In the case of recursive subroutines only the initial call is recorded in the tree; the rest of the recursion is folded into this one node. Our notion of call tree differs from a CCT in the provision of nodes for loops and the differentiation of nodes based on call site within the caller.



```
void initm(double **a, int m, int n)
{
L1: for (int i = 0; i < m; i++)
L2:   for (int j = 0; j < n; j++)
          a[i][j] = drand48() * 1e6;
}

int main(int argc, char **argv)
{
   initm(A, 10, 10);
   initm(B, 10, 10);
}
```

Figure 2. Example code and call trees.

By way of example, the bottom part of Figure 2 shows code to initialize a pair of matrices. The top of the figure contains four trees, the leftmost of which is the call tree. In this tree there are two `initm` children of `main`, because `initm` is called from two different places in `main`. There are also separate nodes for the `L1` and `L2` loops. On the other hand there is only one `drand48` child of the `L2` node, though it is called 100 times.

The other three trees in the top portion of Figure 2 correspond to simplified definitions of context. The second and fourth do not distinguish among call sites. The third and fourth do not track loops. The fourth tree is the CCT of Ammons et al. [1]. In an attempt to determine how much information is really needed to distinguish among application phases, our phase one tool instruments the application binary in a way that allows us to construct all four trees. After running the binary and collecting its statistics, we annotate each tree node with the number of dynamic instances and the total number of instructions executed, from which

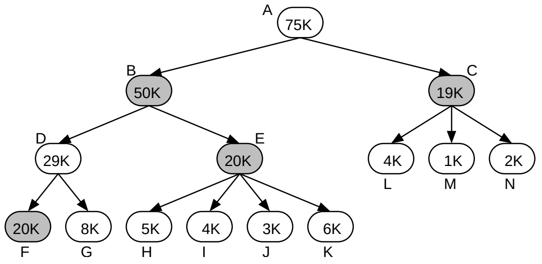


Figure 3. Call tree with associated instruction counts. The shaded nodes have been identified as candidates for reconfiguration.

we can calculate the average number of instructions per instance (including instructions executed in the children of the node). We then identify all nodes that run long enough for a frequency change to take effect and to have a potential impact on energy consumed. We use 10,000 instructions as the definition of "long enough". A longer window could only reduce the quality of reconfiguration, and would have a negligible impact on the instrumentation overhead.

Starting from the leaves and working up, we identify all nodes whose average instance (*excluding* instructions executed in long-running children) exceeds 10,000. Figure 3 shows a call tree in which the long-running nodes have been shaded. Note that these nodes, taken together, are guaranteed to cover almost all of the application history of the profiled run (all that can be missing is a few short-running nodes near the root).

We report results in Section 4 for six different definitions of context. Four correspond to the trees in the upper portion of Figure 2. We call these **L+F+C+P**, **L+F+P**, **F+C+P**, and **F+P**, where **F** stands for "function" (subroutine), **L** stands for "loop", **C** stands for "call site", and **P** stands for "path". We also consider two simpler definitions, **L+F** and **F**, which use the **L+F+P** and **F+P** trees to identify long-running nodes in phase one, but ignore calling history during production runs, allowing them to employ significantly simpler phase four instrumentation.

3.2. The Shaker Algorithm

To select frequencies and corresponding voltages for long-running tree nodes, we use the "shaker" and "slowdown thresholding" algorithms from the off-line analysis of [30]. We run the application through a heavily-modified version of the SimpleScalar/Wattch toolkit [4, 5], with all clock domains at full frequency. During this run we collect a trace of all primitive *events* (temporally contiguous work performed within a single hardware unit on behalf of a single instruction), and of the functional and data dependences among these events. The output of the trace is a dependence DAG for each long-running node in the call tree. Working from this DAG, the shaker algorithm attempts to "stretch" (scale) individual events that are not on the application's critical execution path, as if they could be run at their own, event-specific lower frequency.

Whenever an event in the dependence DAG has two or more incoming arcs, it is likely that one arc constitutes the critical path and that the others will have "slack". Slack indicates that the previous operation completed earlier than necessary. If all of the outgoing arcs of an event have slack, then we have an opportunity to save energy by performing the event at a lower frequency. With each event in the DAG we associate a *power factor* whose initial value is based on the relative power consumption of the corresponding clock domain in our processor model. When we stretch an event we scale its power factor accordingly.

The goal of the shaker is to distribute slack as uniformly as possible. It begins at the end of the DAG and works backward toward the beginning. When it encounters a stretchable event whose power factor exceeds the current *threshold*, originally set to be slightly below that of the few most power-intensive events in the graph (this is a high-power event), the shaker scales the event until either it consumes all the available slack or its power factor drops below the current threshold. If any slack remains, the event is moved later in time, so that as much slack as possible is moved to its *incoming* edges. When it reaches the beginning of the DAG, the shaker reverses direction, reduces its power threshold by a small amount, and makes a new pass forward through the DAG, scaling high-power events and moving slack to outgoing edges. It repeats this process, alternately passing forward and backward over the DAG, reducing its power threshold each time, until all available slack has been consumed, or until all events adjacent to slack edges have been scaled down to one quarter of their original frequency. When it completes its work, the shaker constructs a summary histogram for each clock domain. Each histogram indicates, for each of the frequency steps, the total number of cycles for events in the domain that have been scaled to run at or near that frequency. Histograms for multiple dynamic instances of the same tree node are then combined, and provided as input to the slowdown thresholding algorithm.

3.3. Slowdown Thresholding

Phase three of our profile-driven reconfiguration mechanism recognizes that we cannot in practice scale the frequency of individual events: we must scale each domain as a whole. If we are willing to tolerate a small percentage performance degradation, *d*, we can choose a frequency that causes some events to run slower than ideally they might. Using the histograms generated by the shaker algorithm, we calculate, for each clock domain and long-running tree node, the minimum frequency that would permit the domain to complete its work with no more than *d*% slowdown. More specifically, we choose a frequency such that the sum, over all events in higher bins of the histogram, of the extra time required to execute those events at the chosen frequency is less than or equal to d% of the total length of all events in the node, run at their ideal frequencies. This delay calculation is by necessity approximate. For most applications the overall slowdown estimate turns out to be reasonably accurate: the figures in Section 4 show performance degradation (with respect to the MCD baseline) that is roughly in keeping with d.

3.4. Application Editing

To effect the reconfigurations chosen by the slowdown thresholding algorithm, we must insert code at the beginning and end of each long-running subroutine or loop. For all but the **L+F** and **F** definitions of context, we must also instrument all subroutines that appear on the call chain of one or more long-running nodes. For the **L+F+C+P** and **F+C+P** definitions, we must instrument relevant call sites within such routines. In other words, *reconfiguration points* in the edited binary are a (usually proper) subset of the *instrumentation points*.

To keep track of run-time call chains, we assign a static numeric label to each node in the tree, starting from 1 (0 is a special label that means that we are following a path that did not appear in the tree during training runs). We also assign a static label, from a different name space, to each subroutine that corresponds to one or more nodes in the tree. If the nodes of a call tree are labeled from 1 to N, the corresponding subroutines will be labeled from 0 to $M - 1$, where $M \leq N$. In the most general case (**L+F+C+P**), we instrument the prologue of each of the M subroutines that participate in the tree to access an $(N + 1) \times M$ lookup table, using the label of the previous node at run time (held in a global variable) and the label of the current subroutine (held in a static constant in the code) to find the new node label. Subroutine epilogues restore the previous value.

Headers and footers of long-running loops are also instrumented, as are call sites that may lead to long-running nodes (assuming we're tracking call sites), but these do not need to use the lookup table. It is easy to guarantee that the label of every loop and call site differs by a statically known constant from the label of the subroutine in which it appears. To obtain the new node label, a loop header or call site can simply add an offset to the current label.

We also generate an $N+1$-entry table containing the frequencies chosen by the slowdown thresholding algorithm of phase three. When entering a subroutine (or loop) that corresponds to one or more long-running nodes, we use the newly calculated node label to index into this table. We then write the value found into an MCD hardware reconfiguration register. We assume that this write is a single, unprivileged instruction capable of setting the frequencies of all four domains to arbitrary values.

In the call tree shown in Figure 3, nodes A through F have long-running descendants, or are themselves long-running. Subroutines and loops corresponding to these nodes will therefore have instrumentation instructions at their entry and exit points to keep track of where we are in the tree. Subroutines and loops corresponding to nodes B, C, E and F will also have instructions for reconfiguration. Nodes G through N will not be instrumented, because they cannot reach any long-running nodes.

The two definitions of context that do not use call chain information lead to much simpler instrumentation. Call sites themselves do not need to be instrumented, and there is no need for the lookup tables. Every instrumentation point is in fact a reconfiguration point, and writes statically known frequency values into the hardware reconfiguration register.

Because ATOM does not support in-line instrumentation code, and because subroutine call overhead is high compared to the overhead of the instrumentation itself, we cannot obtain an accurate estimate of run-time overhead using ATOM-instrumented code. For the sake of expediency we have therefore augmented our simulator to emulate the instrumentation code. The current emulation charges a fixed performance and energy penalty for each type of instrumentation point. These penalties are based on worst case values gleaned from the simulated execution of a hand-instrumented microbenchmark. We assume that accesses to the various lookup tables miss in the L1 cache, but hit in the L2 cache. The performance penalty for an instrumentation point that accesses the 2-D table of node labels is about 9 cycles. For a reconfiguration point that subsequently accesses the table of frequencies and writes the reconfiguration register, the penalty rises to about 17 cycles. These relatively small numbers reflect the fact that instrumentation instructions are not on the application's critical path, and can be used to fill pipeline slots that would otherwise be wasted.

4. Results

Our modifications to SimpleScalar and Wattch reflect the synchronization penalties and architectural differences of an MCD processor, and support dynamic voltage and frequency scaling in each domain. We have also modified the simulator to emulate our instrumentation code. Our architectural parameters (Table 1) have been chosen, to the extent possible, to match the Alpha 21264 processor.

We report results for six applications from the MediaBench [24] suite, each comprising "encode" and "decode" operations, and seven applications from the SPEC CPU2000 suite [19] (four floating-point and three integer), resulting in a total of nineteen benchmarks. MediaBench is distributed with two input sets, a smaller one, which we call the *training* set, and a larger one, which we call the *reference* set. For the SPEC benchmarks we used the provided

Branch predictor: comb. of bimodal and 2-level PAg	
Level1	1024 entries, history 10
Level2	1024 entries
Bimodal predictor size	1024
Combining predictor size	4096
BTB	4096 sets, 2-way
Branch Mispredict Penalty	7
Decode / Issue / Retire Width	4 / 6 / 11
L1 Data Cache	64KB, 2-way set associative
L1 Instruction Cache	64KB, 2-way set associative
L2 Unified Cache	1MB, direct mapped
Cache Access Time	2 cycles L1, 12 cycles L2
Integer ALUs	4 + 1 mult/div unit
Floating-Point ALUs	2 + 1 mult/div/sqrt unit
Issue Queue Size	20 int, 15 fp, 64 ld/st
Reorder Buffer Size	80
Physical Register File Size	72 integer, 72 floating-point
Domain Frequency Range	250 MHz – 1.0 GHz
Domain Voltage Range	0.65 V – 1.20 V
Frequency Change Speed	73.3 ns/MHz
Domain Clock Jitter	±110 ps, normally distributed
Inter-domain Synchronization Window	300 ps

Table 1. Simplescalar configuration.

Benchmark	Training	Reference
adpcm_decode	entire program (7.1M)	entire program (11.2M)
adpcm_encode	entire program (8.3M)	entire program (13.3M)
epic_decode	entire program (9.6M)	entire program (10.6M)
epic_encode	entire program (52.9M)	entire program (54.1M)
g721_decode	0 – 200M	0 – 200M
g721_encode	0 – 200M	0 – 200M
gsm_decode	entire program (77.1M)	entire program (122.1M)
gsm_encode	0 – 200M	0 – 200M
jpeg_compress	entire program (19.3M)	entire program (153.4M)
jpeg_decompress	entire program (4.6M)	entire program (36.5M)
mpeg2_decode	entire program (152.3M)	0 – 200M
mpeg2_encode	0 – 200M	0 – 200M
gzip	20,518 – 20,718M	21,185 – 21,385M
vpr	335 – 535M	1,600 – 1,800M
mcf	590 – 790M	1,325 – 1,525M
swim	84 – 284M	575 – 775M
applu	36 – 236M	650 – 850M
art	6,865 – 7,065M	13,398 – 13,598M
equake	958 – 1,158M	4,266 – 4,466M

Table 2. Instruction windows for both the training and reference input sets.

training and reference input sets. Table 2 shows the instruction windows used in our simulations.

4.1. Slowdown and Energy Savings

Our principal results appear in Figures 4, 5, and 6. These show performance degradation, energy savings, and energy×delay improvement, respectively, for the applications in our benchmark suite. All numbers are shown with respect to a baseline MCD processor. Experimentation with additional processor models indicates that the MCD processor has an inherent performance penalty of about 1.3% (max 3.6%) compared to its globally-clocked counterpart, and an energy penalty of 0.8% (max 2.1%). We have not attempted to quantify any performance or energy gains that might accrue from the lack of global clock skew constraints.

The "off-line" and "on-line" bars represent results obtained with perfect future knowledge and with hardware-based reconfiguration, as described in [30] and [29], respectively. The third set of bars represents the **L+F** profiling scheme as described in Section 3. All the simulations were run using the reference input set. The profiling-based cases were trained using the smaller input set. The profiling bars include the performance and energy cost of instrumentation instructions.

Our results indicate that the potential for energy savings from dynamic voltage scaling is quite high. The off-line algorithm achieves about 31% energy savings on average, with 7% slowdown. The savings achieved by the on-line algorithm for the same average slowdown are about 22%, which is roughly 70% of what the off-line achieves. Profile-based reconfiguration achieves almost identical results to the off-line algorithm. This is very promising, because it shows that we can—with some extra overhead at application development time—expect results very close to what an omniscient algorithm with future knowledge can achieve.

Figure 7 summarizes—in the form of "error" bars—the minimum, maximum and average performance degradation, energy savings and energy×delay improvement for the different reconfiguration methods. The "global" numbers correspond to a single-clock processor that employs global dynamic voltage scaling for each benchmark, so as to achieve approximately the same total run time as the off-line algorithm. For example, if the application runs for 100s with the off-line algorithm, but takes only 95s on a single-clock processor running at maximum frequency, the equivalent "global" result assumes that we will run the single-clock processor at 95% of its maximum frequency. As we can see, all MCD reconfiguration methods achieve significantly higher energy savings than "global" does: 82% higher for off-line and **L+F**; 29% higher for on-line (attack-decay).

Figure 7 also highlights the difference in the stability/predictability of the reconfiguration methods. With profile-driven reconfiguration, performance degradation for all applications remains between 1% and 12% (the numbers plotted are for **L+F**, but all the other options stay in this range as well). The on-line algorithm, by contrast, ranges from 1% to 28%. As a result, the energy×delay results for **L+F** are all between 17% and 41%, while those for the on-line algorithm range from −4% to 34%.

4.2. Sensitivity to Calling Context

For our application suite we see relatively little variation due to the different definitions of context discussed in Section 3.1; methods that do not use call chains generally perform as well as the more complicated methods. The ap-

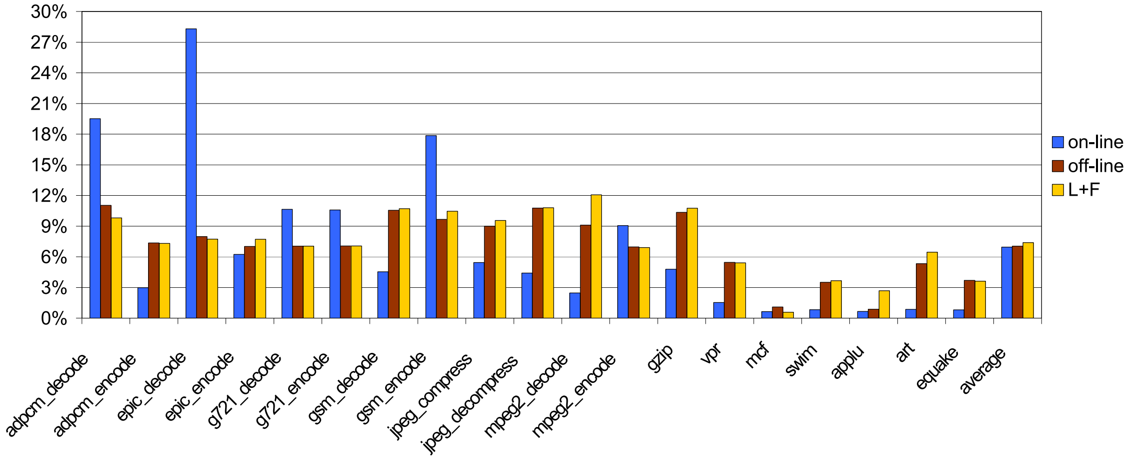


Figure 4. Performance degradation results.

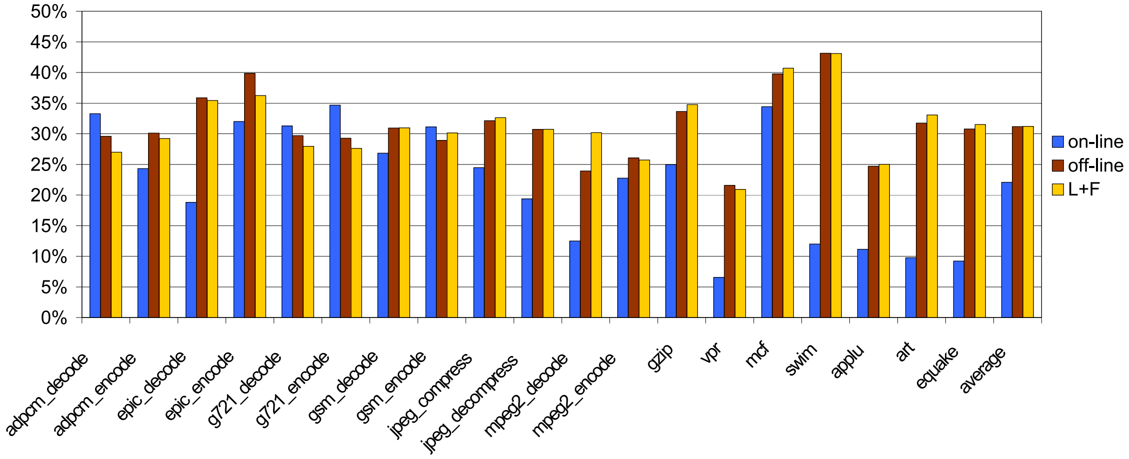


Figure 5. Energy savings results.

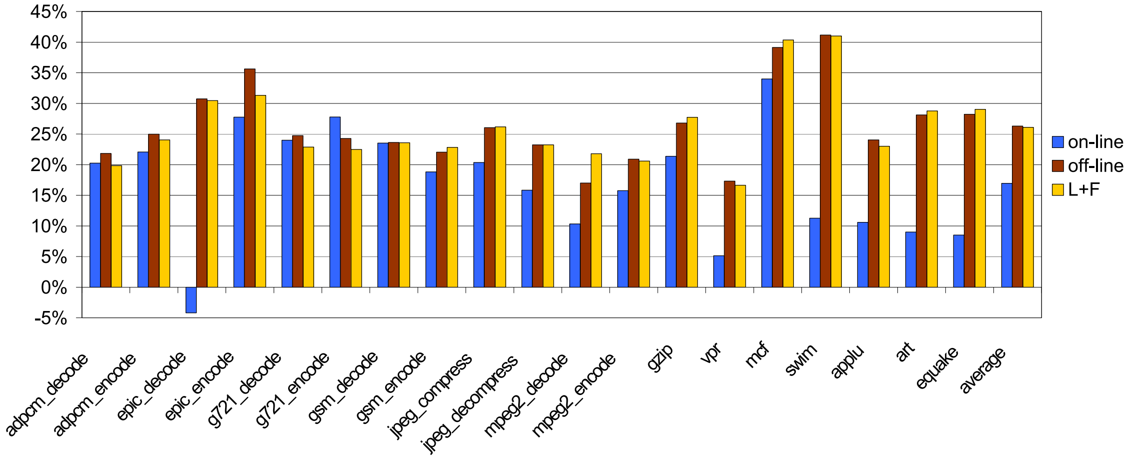


Figure 6. Energy × delay improvement results.

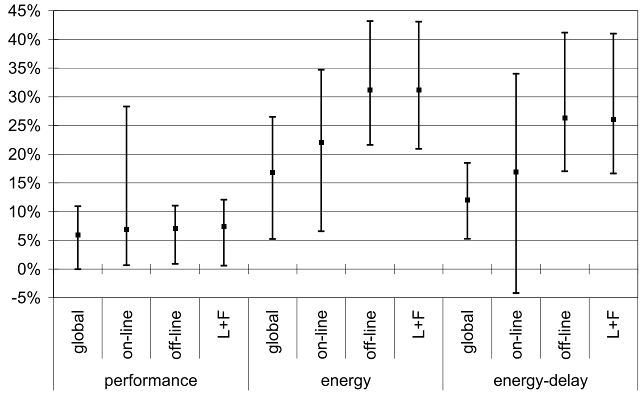


Figure 7. Minimum, maximum and average slowdown, energy savings, and energy×delay improvement.

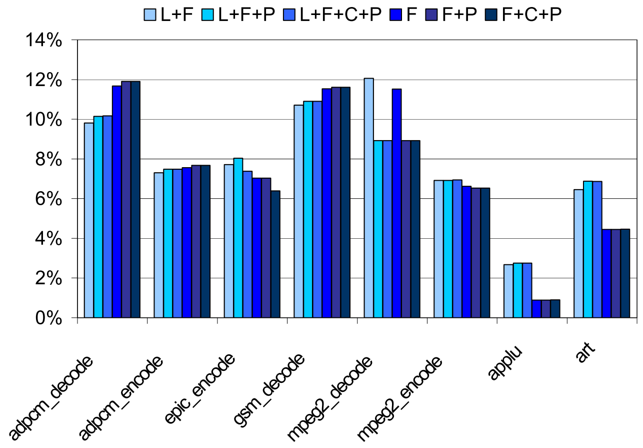


Figure 8. Performance degradation.

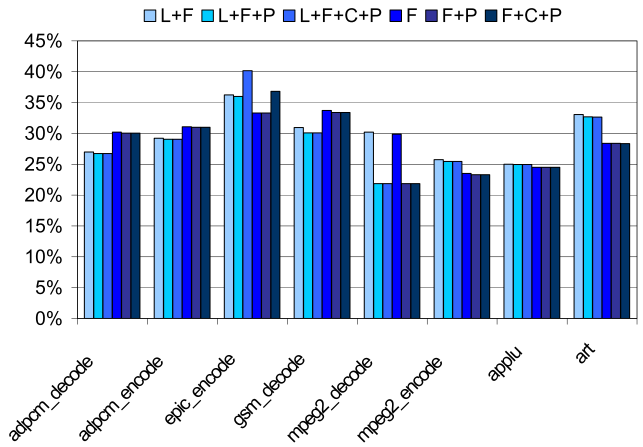


Figure 9. Energy savings.

plications that show some variation are presented in Figures 8 and 9. For *mpeg2_decode*, not having call chains leads to higher performance degradation (and respective energy savings). Investigation reveals different behavior for this application in the training and production runs. Specifically, there are functions that can be reached over multiple code paths, some of which do not arise during training. Mechanisms that track call chains will not reconfigure the processor on these paths at run time. The **L+F** and **F** mechanisms, however, will always change frequencies when they encounter a node that was long-running in the training runs, even when they reach it over a different path.

Instrumentation of call sites produces a noticeable difference in results for only one application: *epic_encode*. Here tracking the site of calls (in **L+F+C+P** and **F+C+P**) leads to slightly lower performance degradation (less than 1%), but *higher* energy savings (approximately 4%) than in **L+F+P** and **F+P**. The difference stems from a subroutine (`internal_filter`) that is called from six different places from inside its parent (`build_level`). Each invocation of `internal_filter` has different behavior, and follows slightly different code paths. Tracking of call sites allows the reconfiguration mechanism to choose different frequencies for the different invocations. When we ignore the call sites we inevitably choose the average frequency of all instances, which, though it yields similar performance degradation, is not as energy efficient.

For most of the MediaBench programs the code is split into a large number of small and medium sized subroutines. A reconfiguration algorithm can do reasonably well by ignoring loops and reconfiguring at the subroutines that include them. *Epic_encode* and *mpeg2_encode* include subroutines with more than one long-running loop. Reconfiguring these loops individually leads to a small increase in performance degradation and a corresponding increase in energy savings. By contrast, in *adpcm_decode*, *gsm_decode* and, to a lesser extent, *adpcm_encode*, loop reconfiguration leads to a *decrease* in performance degradation and a corresponding decrease in energy savings.

One might expect loops to matter more in scientific applications, in which subroutines often contain more than one time-consuming loop nest. We can see examples in two of our floating-point SPEC benchmarks. In *applu*, reconfiguring at loop boundaries increases performance degradation by about 2%, with a 1% gain in energy savings. The increase in performance degradation appears to be due to instrumentation overhead: when not considering loops we change frequency fewer than 10 times in our simulation window (200M instructions); with loop reconfiguration this number rises to about 8,000. In *art*, the core computation resides inside a loop with seven sub-loops. Reconfiguring at loop boundaries again increases performance degradation by about 2%, but with somewhat better energy savings:

roughly 5%.

Based on these results, we recommend the **L+F** method, *i.e.*, reconfiguring at loop and function boundaries, but without including call chain information in the program state. It produces energy and performance results comparable to those of the more complicated algorithms, with lower instrumentation overhead. Using call chains as part of the program state may be a more appropriate method when application behavior changes significantly between the training and production runs, but this situation does not arise in our application suite.

4.3. Sensitivity to Slowdown Threshold

Figures 10 and 11 show the energy savings and energy×delay improvement achieved by the on-line, off-line and **L+F** algorithms relative to achieved slowdown. Several things are apparent from these figures. First, profile-based reconfiguration achieves almost the same energy savings for equivalent performance degradation as the off-line algorithm. The on-line algorithm on the other hand, although it is close to the off-line at low performance degradation targets, starts to tail off with increased slowdown. Beyond a slowdown of 8% the on-line algorithm continues to save energy, but its energy×delay improvement stays the same. Beyond about 15% (not shown here), energy×delay improvement actually begins to decrease. By contrast, the off-line and profile-based reconfiguration methods show a much more linear relationship between performance degradation and energy×delay. We would expect it to tail off eventually, but much farther out on the curve.

4.4. Instrumentation Overhead

Table 3 shows the number of long-running nodes identified by our profiling tool, as well as the total number of nodes in the call tree, for both the data sets, under the most aggressive (**L+F+C+P**) definition of calling context. It indicates the extent to which code paths identified in training runs match those that appear in production runs. (Our profiling mechanism, of course, does not construct call trees for production runs. The numbers in Tables 3 and 4 were collected for comparison purposes only.) The numbers in the "Common" column indicate the number of tree nodes (long-running and total) that appear (with the same ancestors) in the trees of both the training and reference sets. The last column ("Coverage") presents these same numbers as a fraction of the number of nodes in the tree for the reference set. It is clear from this column that the training and reference sets run most applications through the same code paths. The notable exception is *mpeg2_decode*, in which the training set produces only 63% of the tree nodes produced by the reference set. Moreover only 57% of the long-running nodes identified under the training input set are the same as those identified under the reference input set. When run with the reference input set, *swim* also produces more reconfiguration points, because some loops in the code run for more iterations and thus are classified as long running. Unlike *mpeg2_decode* though, all reconfiguration points found with the training set are also found with the reference set.

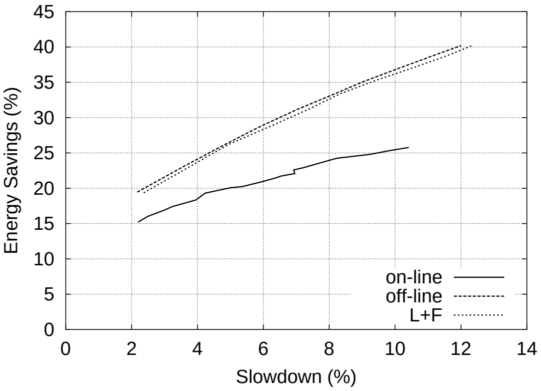


Figure 10. Energy savings for the on-line, off-line and L+F algorithms.

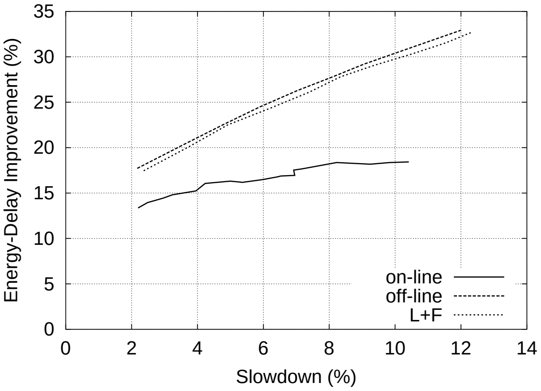


Figure 11. Energy×delay improvement for the on-line, off-line and L+F algorithms.

Table 4 addresses the cost of instrumentation, again for **L+F+C+P**. The second column ("Static") shows the number of static reconfiguration and instrumentation points in the code. These numbers are in some cases smaller than the corresponding numbers in Table 3 because a subroutine or loop may correspond to multiple nodes in the call tree. Numbers are identical between the training and reference data sets, again with the exception of *mpeg2_decode*. The third column of the table ("Dynamic") shows how many times we executed reconfiguration and instrumentation code at run time (profiling with the training set and running with the reference set). These numbers are generally much higher than the static ones, since we execute each subroutine or loop many times. The only exception is

Benchmark	TRAIN		REF		Common		Coverage	
adpcm_decode	2	4	2	4	2	4	1.00	1.00
adpcm_encode	2	4	2	4	2	4	1.00	1.00
epic_decode	18	25	18	25	18	25	1.00	1.00
epic_encode	65	91	65	91	65	91	1.00	1.00
g721_decode	1	1	1	1	1	1	1.00	1.00
g721_encode	1	1	1	1	1	1	1.00	1.00
gsm_decode	3	5	3	5	3	5	1.00	1.00
gsm_encode	6	9	6	9	6	9	1.00	1.00
jpeg_compress	11	17	11	17	11	17	1.00	1.00
jpeg_decompress	4	6	4	6	4	6	1.00	1.00
mpeg2_decode	11	15	14	19	8	12	0.57	0.63
mpeg2_encode	30	40	30	40	30	40	1.00	1.00
gzip	78	224	70	196	65	182	0.93	0.93
vpr	67	92	84	119	7	12	0.08	0.10
mcf	26	41	26	41	26	41	1.00	1.00
swim	16	23	25	32	16	23	0.64	0.78
applu	61	77	68	85	60	77	0.98	0.91
art	65	98	68	100	65	98	0.96	0.98
equake	30	35	30	35	30	35	1.00	1.00

Table 3. Number of reconfiguration nodes and total number of nodes in the call tree when profiling with the reference and training sets.

Benchmark	Static		Dynamic		Overhead
adpcm_decode	2	4	470	939	0.33%
adpcm_encode	2	4	470	939	0.17%
epic_decode	18	25	106	149	0.03%
epic_encode	27	40	4270	4441	0.29%
g721_decode	1	1	1	1	0.00%
g721_encode	1	1	1	1	0.00%
gsm_decode	3	5	5841	11681	0.30%
gsm_encode	6	9	12057	16579	0.35%
jpeg_compress	7	11	40	45	0.00%
jpeg_decompress	4	6	1411	1415	0.17%
mpeg2_decode	4	7	1	2	0.00%
mpeg2_encode	30	40	7264	7283	0.18%
gzip	19	56	153	170	0.00%
vpr	56	75	3303	3307	0.03%
mcf	24	37	10477	16030	0.03%
swim	16	23	5344	5352	0.03%
applu	49	62	7968	7968	0.06%
art	43	64	63080	65984	0.39%
equake	30	35	34	41	0.00%

Table 4. Static and dynamic reconfiguration and instrumentation points, and estimated run-time overhead for L+F+C+P.

mpeg2_decode, where the number of dynamic reconfiguration/instrumentation points is smaller than the static. This happens because we use ATOM to profile the application and so the static numbers correspond to the whole program execution. The production runs, however, are executed on SimpleScalar, and use an instruction window of only 200M instructions—much less than the entire program.

The final column of Table 4 shows the cost, as a percentage of total application run time, of the extra instructions injected in the binary, as estimated by our simulator. These numbers, of course, are for the **L+F+C+P** worst case. Figure 12 shows the number of instrumentation points, and associated run-time overhead, of the simpler alternatives, normalized to the overhead of **L+F+C+P**. The first two sets of bars compare the number of static reconfiguration and instrumentation points in the code for the different profiling algorithms, averaged across all applications. The **L+F** and **F** methods, of course, have no static instrumentation points, only reconfiguration points. Note also that the number of static instrumentation points is independent of whether we track call chain information at run time; all that varies is the *cost* of those instrumentation points. Thus, for example, **L+F+P** will always have the same number of long-running subroutines and loops as **L+F**. The last set of bars in Figure 12 compares the run-time overhead of the different profiling methods. As expected, **L+F+C+P** has the highest overhead. Interestingly, the number of instrumentation instructions for **L+F** and **F** is so small that there is almost perfect scheduling with the rest of the program instructions, and the overhead is virtually zero.

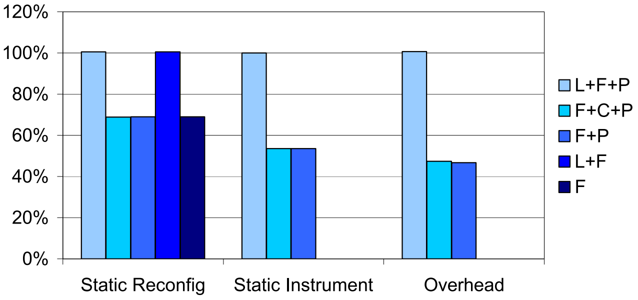


Figure 12. Number of static reconfiguration and instrumentation points and run-time overhead compared to L+F+C+P.

Tables 3 and 4 also allow us to estimate the size of our lookup tables. In the worst case (*gzip*) we need a 225×56-entry table to keep track of the current node, and a 225-entry table to store all the domain frequencies: a total of about 13KB. All other benchmarks need less than 4KB.

5. Related Work

Microprocessor manufacturers such as Intel [25] and Transmeta [18] offer processors capable of global dynamic frequency and voltage scaling. Marculescu [26] and Hsu et al. [20] evaluated the use of whole-chip dynamic voltage scaling (DVS) with minimal loss of performance using cache misses as the trigger. Following the lead of Weiser et al. [32], many groups have proposed OS-level mechanisms to "squeeze out the idle time" in underloaded systems via whole-chip DVS. Childers et al. [10] propose to trade IPC for clock frequency, to achieve a user-requested quality of service from the system (expressed in MIPS). Processes that can achieve higher MIPS than the current QoS setting are slowed to reduce energy consumption. By contrast, our work aims to stay as close as possible to the maximum performance of each individual application.

Other work [7, 28] proposes to steer instructions to pipelines or functional units running statically at different speeds so as to exploit scheduling slack in the program to save energy. Fields et al. [15] use a dependence graph similar to ours, but constructed on the fly, to identify the critical path of an application. Their goal is to improve instruction steering in clustered architectures and to improve value prediction by selectively applying it to critical instructions only. Fields et al. [14] also introduce an on-line "slack" predictor, based on the application's recent history, in order to steer instructions between a fast and a slow pipeline.

Huang et al. [21] also use profiling to reduce energy, but they do not consider dynamic voltage and frequency scaling. Their profiler runs every important function with every combination of four different power-saving techniques to see which combination uses the least energy with negligible slowdown. Our work minimizes performance degradation by scaling only the portions of the processor that are not on the critical path. We also consider reconfiguration based on loop and call chain information, and require only a single profiling run for each set of training data.

Several groups have used basic block or edge profiling and heuristics to identify heavily executed program paths [8, 12, 16, 33]. Ball and Larus [3] first introduced an efficient technique for *path profiling*. In a follow-up study, Ammons et al. [1] describe a technique for context sensitive profiling, introducing the notion of a *calling context tree (CCT)*, on which our *call trees* are based. Ammons et al. also describe a mechanism to construct the CCT, and to associate runtime statistics with tree nodes. They demonstrate that context sensitive profiling can expose different behavior for functions called in different contexts. We borrow heavily from this work, extending it to include loops as nodes of the CCT, and differentiating among calls to the same routine from different places within a single caller. The reason we need the CCT is also different in our case. Instead of the most frequently executed paths, we need the minimum number of large-enough calling contexts that cover the whole execution of the program.

6. Conclusions

We have described and evaluated a profile-driven reconfiguration mechanism for a Multiple Clock Domain microprocessor. Using data obtained during profiling runs, we modify applications to scale frequencies and voltages at appropriate points during later production runs. Our results suggest that this mechanism provides a practical and effective way to save significant amounts of energy in many applications, with acceptable performance degradation.

In comparison to a baseline MCD processor, we demonstrate average energy savings of approximately 31%, with performance degradation of 7%, on 19 multimedia and SPEC benchmarks. These results rival those obtained by an off-line analysis tool with perfect future knowledge, and are significantly better than those obtained using a previously published hardware-based on-line reconfiguration algorithm. Our results also indicate that profile-driven reconfiguration is significantly more stable than the on-line alternative. The downside is the need for training runs, which may be infeasible in some environments.

We believe that profile-driven application editing can be used for additional forms of architectural adaptation, e.g. the reconfiguration of CAM/RAM structures [2, 6]. We also hope to develop our profiling and simulation infrastructure into a general-purpose system for context-sensitive analysis of application performance and energy use.

Acknowledgements

We are grateful to Sandhya Dwarkadas for her many contributions to the MCD design and suggestions on this study.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, June 1997.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 245–257, Dec. 2000.

[3] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Dec. 1996.

[4] 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.

[5] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, Computer Science Department, University of Wisconsin, June 1997.

[6] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. H. Albonesi. An Adaptive Issue Queue for Reduced Power at High Performance. In *Proceedings of the Workshop on Power-Aware Computer Systems, in conjunction with ASPLOS-IX*, Nov. 2000.

[7] J. Casmira and D. Grunwald. Dynamic Instruction Scheduling Slack. In *Proceedings of the Kool Chips Workshop, in conjunction with MICRO-33*, Dec. 2000.

[8] P. P. Chang. Trace Selection for Compiling Large C Application Programs to Microcode. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO 21)*, pages 21–29, Nov. 1988.

[9] D. M. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. PhD thesis, Stanford University, 1984.

[10] B. R. Childers, H. Tang, and R. Melhem. Adapting Processor Supply Voltage to Instruction-Level Parallelism. In *Proceedings of the Kool Chips Workshop, in conjunction with MICRO-34*, Dec. 2001.

[11] 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*, June 2001.

[12] J. R. Ellis. A Compiler for VLIW Architectures. Technical Report YALEU/DCS/RR-364, Yale University, Department of Computer Science, Feb. 1985.

[13] A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the USENIX 1995 Technical Conference*, Jan. 1995.

[14] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing Performance Under Technological Constraints. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 47–58, May 2002.

[15] B. Fields, S. Rubin, and R. Bodík. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[16] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[17] M. Fleischmann. Crusoe Power Management – Reducing the Operating Power with LongRun. In *Proceedings of the HOT CHIPS Symposium XII*, Aug. 2000.

[18] T. R. Halfhill. Transmeta breaks x86 low power barrier. *Microprocessor Report*, 14(2), Feb. 2000.

[19] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, pages 28–35, July 2000.

[20] C.-H. 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 ASPLOS-IX*, Nov. 2000.

[21] M. Huang, J. Renau, and J. Torrellas. Profile-Based Energy Reduction in High-Performance Processors. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[22] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[23] 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*, May 2002.

[24] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[25] S. Leibson. XScale (StrongArm-2) Muscles In. *Microprocessor Report*, 14(9):7–12, Sept. 2000.

[26] D. Marculescu. On the Use of Microarchitecture-Driven Dynamic Voltage Scaling. In *Proceedings of the Workshop on Complexity-Effective Design, in conjunction with ISCA-27*, June 2000.

[27] Intel Corp. Datasheet: Intel® Pentium® 4 Processor with 512-KB L2 cache on 0.13 Micron Process at 2 GHz–3.06 GHz. Available at http://www.intel.com/design/pentium4/-datashts/298643.htm, Nov. 2002.

[28] R. Pyreddy and G. Tyson. Evaluating Design Tradeoffs in Dual Speed Pipelines. In *Proceedings of the Workshop on Complexity-Effective Design, in conjunction with ISCA-28*, June 2001.

[29] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.

[30] 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*, Feb. 2002.

[31] 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, Sept. 1997.

[32] 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*, Nov. 1994.

[33] C. Young and M. D. Smith. Improving the Accuracy of Static Branch Prediction Using Branch Correlation. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, Oct. 1994.