Improving Application Performance by Dynamically Trading Frequency for Complexity in a GALS Microprocessor*

Greg Semeraro, David H. Albonesi, Steven Dropsho,
Grigorios Magklis, and Michael L. Scott

URCS Technical Report #832
February 2004

Abstract

Microprocessors are traditionally designed to provide "best overall" performance across a wide range of applications and operating environments. Several groups have proposed hardware techniques that save energy by "downsizing" hardware resources that are underutilized by the current application. We explore the converse: improving performance by "upsizing" resources for which the application has greater needs. Our proposal depends critically on the ability to change frequencies independently in separate domains of a globally asynchronous, locally synchronous (GALS) microprocessor.

We use a variant of a multiple clock domain (MCD) processor, with four independently clocked domains. Each domain is streamlined with modest hardware structures for very high clock frequency. Key structures can then be upsized on demand to exploit more distant parallelism, improve branch prediction, or increase cache capacity. Although doing so requires decreasing the associated domain frequency, other domain frequencies are unaffected. Measuring across a broad suite of application benchmarks, we find that configuring our MCD processor just once per application yields performance 17.6% better, on average, than that of the "best overall" fully synchronous design. By adapting automatically to application phases, we can increase this advantage to more than 20%.

*This work was supported in part by NSF grants CCR-9701915, CCR-9811929, CCR-9988361, EIA-0080124, and CCR-0204344; by DARPA/ITO under AFRL contract F29601-00-K-0182; by and IBM Faculty Partnership Award; and by equipment grants from IBM and Intel. Authors' current addresses: Greg Semeraro, Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY 14623, gpseec@ce.rit.edu; David Albonesi, Department of Electrical and Computer Engineering, University of Rochester, Rochester, NY 14627, albonesi@ece.rochester.edu; Steven Dropsho, EPFL–IC–LABOS, CH 1015 Lausanne, Switzerland, dropsho@cs.rochester.edu; Grigorios Magklis, Intel Labs Barcelona, Jordi Girona 29-3A, 08034 Barcelona, Spain, grigoriosx.magklis@intel.com; Michael Scott, Department of Computer Science, University of Rochester, Rochester, NY 14627, scott@cs.rochester.edu.

1. Introduction

Microprocessor design traditionally embodies a tradeoff between processor frequency and the hardware complexity required to exploit instruction-level parallelism (ILP). Any particular design is typically a compromise, because the tradeoff is highly application dependent. Optimal pipeline depth, for example, tends to vary across classes of applications, *e.g.* SPEC versus transaction processing [13, 20, 30]. While some applications benefit from a streamlined design in which hardware complexity is kept relatively modest, in favor of high clock rates, others benefit from a more complex design that uses larger hardware structures to exploit more ILP or to cache larger working sets, at the cost of a lower clock rate. Studies have even demonstrated significant variability in hardware requirements among the different phases of a given application [8, 28, 32]. Any processor designed for "best overall" performance inevitably makes inefficient use of its hardware resources when running a varied workload. This observation suggests the possibility of dynamically optimizing the tradeoff between clock rate and IPC (instructions per cycle) [1].

The simplest approach is to "downsize" underutilized structures in order to save energy. Proposals to date include adaptive cache hierarchies [2, 8, 24], adaptive issue queues [5, 11], and combinations including caches, issue queues, register files, and the reorder buffer [9, 23].

A downsized structure often operates at a higher speed. For instance, Buyuktosunoglu [5] demonstrated a 70% reduction in issue queue access time when downsizing from 32 entries to 8. Unfortunately, critical timing paths elsewhere in the chip typically prevent this speedup from translating into an overall increase in clock rate.

If one aims to improve performance, an alternative strategy [1] is to decrease global frequency when one or more structures are "upsized". This strategy preserves the correctness of critical timing paths, but succeeds only when the performance gained by upsizing some subset of the chip's hardware resources (*e.g.*, the data cache hierarchy) exceeds the performance lost by slowing down everything else, and

this is seldom the case. Most applications are sufficiently balanced in their hardware usage that the performance cost of a global slowdown easily exceeds the benefit of upsizing a subset of the hardware resources.

Globally-Asynchronous, Locally-Synchronous (GALS) microprocessors [15, 21, 27] offer a new opportunity to dynamically trade frequency for complexity within a *subset* of the processor. In a previously proposed Multiple Clock Domain (MCD) GALS design [27], the chip is broken down into separate domains for (1) front-end fetch / rename, (2) integer execution, (3) floating point execution, and (4) load / store (L1 data cache and L2 cache). Clock frequency can be varied in each domain independent of the others; synchronization circuits are used on all cross-domain communication paths. By varying voltage with frequency, significant energy can be saved with only a modest loss in performance [17, 27]. In this paper, we use a similar architecture in a quest to *improve* performance.

We begin with a baseline MCD processor in which each domain is optimized for high frequency, with relatively low hardware complexity. Dynamic upsizing of key hardware structures (with an accompanying decrease in domain frequency) then offers per-application tailoring of the frequency / complexity tradeoff. In effect, we trade frequency for complexity whenever an increase in potential ILP provides IPC benefits that override the frequency loss *in that one domain*. Our baseline processor has a higher branch mispredict penalty than is typical of fully synchronous designs, due to pipeline inefficiencies at lower frequencies. It also suffers a modest frequency penalty in some configurations due to the ability to adapt, and a modest increase in execution time due to the latency of cross-domain synchronization. Despite these handicaps, we demonstrate overall performance improvements of 17.6% with respect to the *best fully synchronous design* using only profile-based whole-program adaptation of the instruction cache, branch predictor, integer issue queue, floating-point issue queue, and data / L2 caches. When we add an adaptation algorithm that automatically detects phases of program behavior and sizes structures appropriately for each phase, average performance improvement reaches 20.4%. These results reflect an exhaustive exploration of the space of synchronous and whole-program adaptive processor configurations, embodying approximately 300 CPU months of simulation time.

The rest of this paper is organized as follows. In the next section, we describe the adaptive GALS microarchitecture, in which structures can be upsized while lowering domain frequency. We provide details of the structures that are resized in each domain and the timings for each. Our adaptive control algorithms are described in Section 3. Section 4 describes our experimental methodology; Section 5 presents results. We discuss related work in Section 6, and conclude in Section 7.

2. Adaptive GALS microarchitecture

The MCD architecture highlighted in Figure 1 has four independent clock domains, comprising the front end (L1 instruction cache, branch prediction, rename, reorder buffer and dispatch); integer processing core (issue queue, register file and execution units); floating-point processing core (issue queue, register file and execution units); and load / store unit (load / store queue, L1 data cache and unified L2 cache). The dynamic frequency control circuit within each of these domains is a PLL clocking circuit based on industrial circuits [7, 10]. The lock time in our experiments is normally distributed with a mean time of 15μs and a range of 10–20μs. As in the XScale processor [7], we assume that a domain is able to continue operating through a frequency change. Main memory can be thought of as a separate fifth domain, but it operates at a fixed base frequency and hence is non-adaptive.

Data generated in one domain and needed in another must cross a domain boundary, potentially incurring synchronization costs. The MCD simulator models synchronization circuitry based on the work of Sjogren and Myers [29]. 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. Both superscalar execution (which allows instructions to cross domains in groups) and out-of-order execution (which reduces the impact of individual instruction latencies) tend to hide synchronization costs, resulting in an average overall slowdown of less than 3%. Further details on the baseline MCD model, including a description of the inter-domain synchronization circuitry, can be found in prior papers [17, 27].

For this study we add adaptive structures to the baseline MCD domains. The resulting *adaptive MCD architecture* has a base configuration with small and simple structures running at a very high clock rate. For applications that perform better with additional resources, key structures can be upsized with a corresponding reduction in the clock rate of their domain. Unaffected domains still run at their base high clock rate.

Having adaptable structures and a variable clock means that structures may be safely oversized. The greater capacity (and lower domain frequency) is used only if an application attains a net benefit. Applications that do not require the extra capacity configure to a smaller size and run at a higher frequency. This approach permits the tradeoff between per-domain clock rate and complexity to be made for each application or application phase.

The resizable structures are shown in Figure 1. In the front end, the instruction cache and branch predictor are jointly resizable (*i.e.*, each cache configuration is paired with a branch predictor sized to operate at the frequency of the cache). This permits applications with larger instruc-

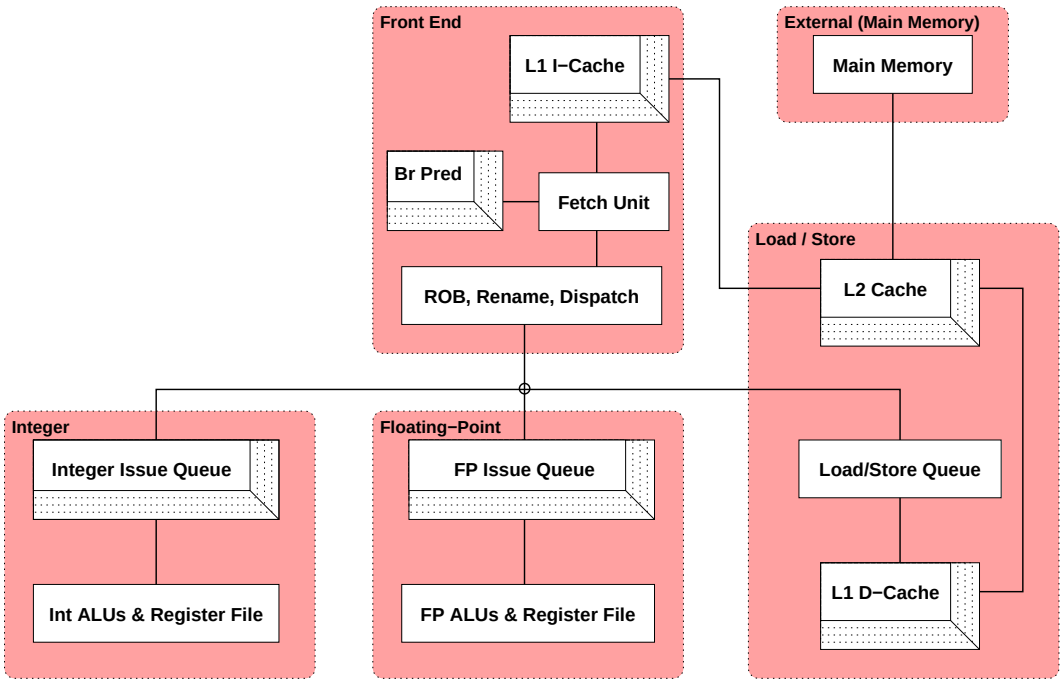


Figure 1. The clock domains of the MCD architecture [27]. Boxes with multiple borders indicate resizable structures.

tion footprints, or those that require more branch prediction resources, to be accommodated, albeit at the cost of a lower domain frequency. Similarly, in the load / store domain, the data cache and second level cache are resizable, also as matched pairs. In the integer and floating-point domains, issue queues can be resized to match available ILP. Additional structures could also conceivably be resized; we leave these options for future work.

There are costs associated with supporting adaptive structures. With the smallest sizings, per-domain pipe stage delays are balanced in order to attain the highest clock rate. When the clock frequency is lowered to accommodate the additional delay of an upsized structure, the resulting stage delay imbalance results in a design that is overpipelined with respect to the particular frequency. The cost is a longer branch mis-predict penalty. In our study, the adaptive MCD incurs two additional integer cycles and one additional front-end cycle for branch mispredictions. In addition, the base MCD configuration must have its structures designed for maximum performance in order to achieve the highest possible clock rate. But to support resizing, the smallest structure size must be a substructure of the larger sizings. Thus, structures may be suboptimal in their large configurations relative to the same size structure that has been optimized for a non-adaptable design.

Table 1. L1 data and L2 cache configurations.

L1-D		sub-banks		L2		sub-banks	
size	assoc	adapt	optimal	size	assoc	adapt	optimal
32 KB	1	32	32	256 KB	1	8	8
64 KB	2	32	8	512 KB	2	8	4
128 KB	4	32	16	1 MB	4	8	4
256 KB	8	32	4	2 MB	8	8	4

2.1. Load / store domain

In the load / store domain, the L1 data and L2 caches are 8-way associative, and resized by ways [9]. The base configuration (smallest size and highest clock rate) is a 32 KB direct-mapped data cache and a 256 KB direct-mapped L2 cache. The two caches are upsized together by increasing their associativity. The configurations we consider in our experiments are shown in Table 1. To limit the state space of possible configurations, we do not make use of the available 3, 5, 6, or 7-way associative configurations.

We use the CACTI modeling tool [31] (version 3.1) to obtain timings for all plausible cache configurations at a given size. The *optimal* columns in Table 1 describe the configurations that provide the fastest cycle time for the given capacity and associativity, without the ability to resize. The *adapt* columns were chosen by adopting the fastest configuration of the minimal-size structure and then replicating this configuration at higher levels of associativity to obtain the larger configurations. This strategy ensures the fastest clock frequency at the minimum configuration, but may not produce the fastest configuration when struc-

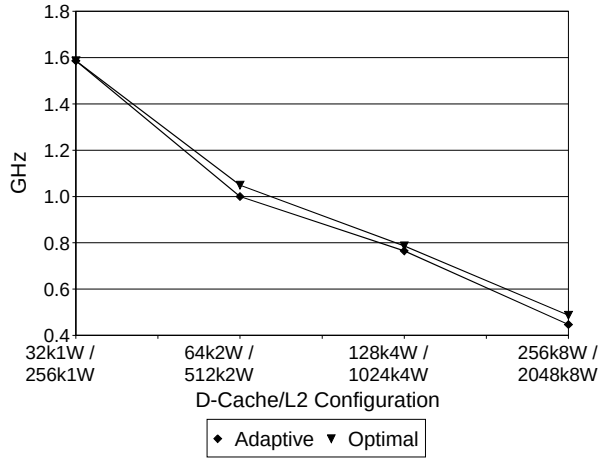


Figure 2. D-cache / L2 frequency versus configuration.

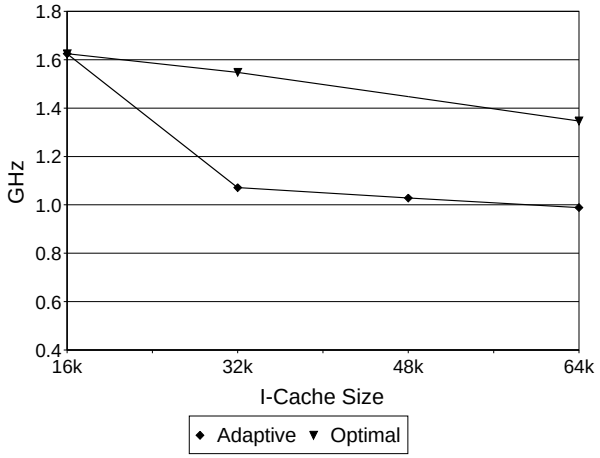


Figure 3. I-Cache Frequency versus Configuration.

tures are upsized. Since CACTI configures a 32KB direct-mapped cache as 32 sub-banks, each additional way in the adaptive L1-D cache is an identical 32 KB RAM. The reconfigurable L2, similarly, has 8 sub-banks per 256 KB way. In contrast, the number of sub-banks in an optimal fixed L1 varies with total capacity, and the optimal L2 structure has 4 sub-banks per way for all sizes larger than the minimum.

Frequencies for the various cache configurations, optimal and adaptive, are plotted in Figure 2. The difference between the optimized and adaptive configurations is small: approximately 5%.

2.2. Front end domain

In the front end both the instruction cache and the branch predictor are adaptive. Like the L1-D and L2 caches, they are always resized together. The configurations we consider are shown in Table 2. Like the data caches, the instruction cache adapts by ways, but with associativities of 1, 2, 3, and 4. The branch predictor is a hybrid design with a global shared history (*gshare*) component, a local history component, and a metapredictor to select which component to use [19]. The *gshare* component is a global branch history table (BHT) of 2^{h_g} two-bit counters indexed by the h_g-bit global history. The local history component is a pattern history table (PHT) holding the histories for different branches. The table is indexed by the branch PC. It returns an h_l-bit wide local history, which is used to index a local BHT of 2^{h_l} two-bit counters.

For the conventional fully synchronous design, we explored a wide range of different instruction cache configurations (Table 3), ranging in size from 4 to 64KB, and in associativity from 1 to 4, to find the best option. Associated with each instruction cache configuration is a branch predictor organization with a similar delay. Averaged across

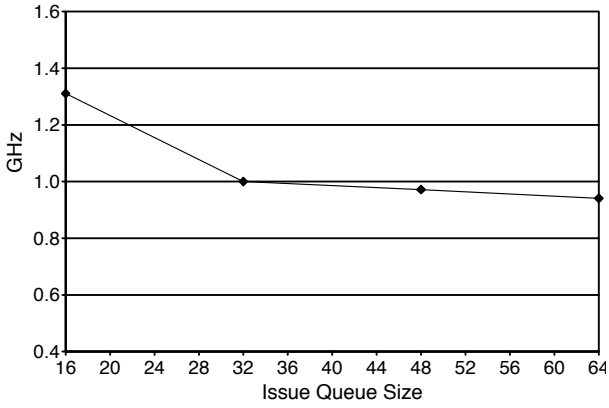


Figure 4. Issue queue frequency analysis.

our suite of 32 MediaBench, Olden, and SPEC2000 applications, a direct-mapped 64KB configuration and its associated branch predictor provide the best performance.

Figure 3 compares the operating frequency of an optimally configured direct-mapped cache to that of our adaptive configurations at various total cache sizes. As is clear on the adaptive curve, there is a large difference in frequency between direct-mapped and 2-way set associative configurations: approximately 31%. Since instruction streams tend to need little associativity, the optimal large instruction cache is direct-mapped and aggressively clocked. This is a clear advantage to the fully synchronous processor that the adaptive design must overcome.

2.3. Integer and floating point domains

In both the integer and floating point domains, the issue queues are resizable from 16 to 64 entries in four increments. The issue queue timings are derived as described by Palacharla *et al.* [22], using the same technology file

Table 2. Adaptive instruction cache / branch predictor configurations.

I-cache, dynamic			Branch predictor					
Size	**Assoc**	**Sub-banks**	h_g	***gshare* PHT**	**Meta-predictor**	h_l	***local* PHT**	***local* BHT**
16 KB	1	32	14 bits	16384	16384	11 bits	2048	1024
32 KB	2	32	15 bits	32768	32768	12 bits	4096	1024
48 KB	3	32	15 bits	32768	32768	12 bits	4096	1024
64 KB	4	32	16 bits	65536	65536	13 bits	8192	1024

Table 3. Optimized instruction cache / branch predictor configurations.

I-cache, optimized			Branch predictor					
Size	**Assoc**	**Sub-banks**	h_g	***gshare* PHT**	**Meta-predictor**	h_l	***local* PHT**	***local* BHT**
4 KB	1	2	12 bits	4096	4096	10 bits	1024	512
8 KB	1	4	13 bits	8192	8192	10 bits	1024	1024
16 KB	1	16	14 bits	16384	16384	11 bits	2048	1024
32 KB	1	32	15 bits	32768	32768	12 bits	4096	1024
64 KB	1	32	16 bits	65536	65536	13 bits	8192	1024
4 KB	2	8	12 bits	4096	4096	10 bits	1024	512
8 KB	2	16	13 bits	8192	8192	10 bits	1024	1024
16 KB	2	32	14 bits	16384	16384	11 bits	2048	1024
32 KB	2	32	15 bits	32768	32768	12 bits	4096	1024
64 KB	2	32	16 bits	65536	65536	13 bits	8192	1024
12 KB	3	16	13 bits	8192	8192	10 bits	1024	1024
16 KB	4	16	14 bits	16384	16384	11 bits	2048	1024
24 KB	3	32	14 bits	16384	16384	11 bits	2048	1024
32 KB	4	2	15 bits	32768	32768	12 bits	4096	1024
48 KB	3	32	15 bits	32768	32768	12 bits	4096	1024
64 KB	4	16	16 bits	65536	65536	13 bits	8192	1024

used by CACTI. Based on the results of Buyuktosunoglu *et al*. [5], we assume that a resizable issue queue suffers no access penalty over a non-resizable issue queue of the same size. A plot of queue frequencies is shown in Figure 4. Note that due to the $\log_4$ structure of the selection logic and the fact that the selection delay is much larger than the wake-up delay, we suffer a significant frequency decrease when moving from an issue queue with 16 entries (which has 2 levels of logic) to any larger issue queue up to 64 entries (all of which have 3 levels of logic). As will be explained more fully in Section 5, this frequency effect causes a 16-entry issue queue to be best for most applications.

3. Adaptive control algorithms

Both caches and issue queues require some sort of control algorithm to support on-line adaption to program phases. We describe our algorithms in this section. Note that they are not used in the whole-program experiments: for those we pick the configurations that show the best overall behavior based on exhaustive (off-line) exploration of the state space.

3.1. Phase adaptive caches

For our reconfigurable caches we employ the *Accounting Cache* [9] described in earlier work on energy efficiency. As described in Section 2, this cache adapts by ways. Even when running in a limited number of ways, however, it collects statistics in the remaining ways, permitting the calculation of the number of hits and misses that would have occurred over a given span of time for any of the possible configurations.

The four possible configurations of a 4-way set associative Accounting Cache are shown in Figure 5. In this example, the *A* partition can be 1, 2, 3, or 4 ways. The *B* partition is the remaining portion. The *A* partition is accessed first. If the data block is found it is returned. Otherwise a second access is made to the *B* partition, which causes blocks to be swapped. Note that the *B* partition is not considered a lower level of cache. All three caches of the adaptive MCD machine (L1I, L1D, and combined L2) have their own *A* and *B* partitions. When we simulate a fully synchronous processor, however, or when we choose a single adaptive configuration for the entire program run, we use only the *A*

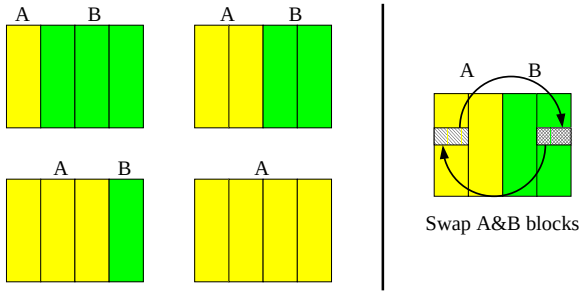


Figure 5. Partitioning options for a 4-way Accounting Cache [9].

partitions: a miss in *A* skips *B* and goes directly to the next lower level of the memory hierarchy.

A cache with a small *A* partition runs at a higher frequency than one with a larger *A* partition. Ideally one would make the *A* partition as small as possible without generating a significant number of *B* accesses. The *B* partition access latency is an integral number of cycles at the clock rate dictated by the size of the *A* partition.

As described in detail in previous work [9], the Accounting Cache maintains full most-recently-used (MRU) state on cache lines. Simple counts of the number of blocks accessed in each MRU state are sufficient to reconstruct the precise number of hits and misses to the *A* and *B* partitions *for all possible cache configurations, regardless of the current configuration*. The control algorithm resets the counts at the end of every 15K instruction interval, choosing a configuration for the next interval that would have minimized total access cost in the interval just ended. The hardware components required to perform this calculation are given in Table 4. The delay calculations are made with dedicated hardware—one for the instruction cache and one for the L1 data and L2 cache pair—using dedicated arithmetic circuits requiring an estimated 10k equivalent gates [33] (5K for the instruction cache and 5K for the L1 / L2 data caches). A complete reconfiguration decision requires approximately 32 cycles, based on binary addition trees and the generation of a single partial product per cycle; multi-operand adders and encoded multiplication would result in a faster calculation at the expense of additional circuitry.

3.2. Adaptive issue queues

A special property of the adaptive *Accounting Cache* design is that it avoids exploration of the configuration space when determining the best configuration. This property is also desirable for issue queue control. We introduce a new, deterministic algorithm to measure the inherent ILP of the currently running application, independent of microarchitectural features. The queue control algorithm uses this figure to choose among the four possible queue sizes the one that maximizes effective ILP, normalized to frequency.

A key observation is that the amount of inherent parallelism in the instruction stream can be calculated by immediate dependences of output registers on input registers. The earliest a result can be ready is the latest time of any of its input operands plus the latency of the operation. For this discussion let us assume that all ALU operations have a one cycle latency. As instructions are fetched, their source registers are renamed via the register rename mapping table. With tracking hardware initially reset, all input operands for the first instruction will have timestamps of zero and the destination register will receive a timestamp of '1'. If the next instruction uses that destination register as an input then its own destination register will be updated with a timestamp of '2', and so on. The maximum timestamp M is continuously recorded during this process.

Tracking continues until N instructions have been fetched, where N represents the queue size of 16, 32, 48, or 64. At that time an estimate of the application's ILP is N/M_N. Of course, the division is not actually performed because the numerator is a fixed quantity and the denominators can be compared directly in integer form. When all four estimates have been computed, the control algorithm scales them by the corresponding frequencies and compares them to determine which queue size would have led, in the very recent past, to the highest effective ILP. Dedicated hardware optimized for this task is modest.

The majority of the hardware is in extra storage to hold the timestamps: four bits per register to track the ILP for the 16 entry queue (ILP_{16}), five bits for ILP_{32}, and six bits each for ILP_{48} and ILP_{64}. The tracking intermingles integer and floating point operations, keeping a count of each. A tracking interval ends when either count, N_{INT} or N_{FP}, reaches N. This operation correctly stifles consideration of larger queue sizes that can never be filled for the less dominant instruction type because of resource limitations for the dominant instruction type. With 32 logical integer and 32 logical floating point registers, ILP tracking requires an additional 256 bits for ILP_{16} and 384 bits for ILP_{64}. In our experiments we track the ILP for all queues depths simultaneously, and consider the possibility of resizing as soon as all four counts are available. Alternatively, one could use one set of hardware counters and calculate the four ILP values serially by cycling through the values of N. Counters are reset at the end of each tracking interval.

Functional unit resource limitations can also be easily incorporated to guard against over-estimating potential ILP, but we found this additional precision had little practical impact on the algorithm's performance. This resizing control algorithm avoids search and the problem of local minima due to the (often) non-monotonic performance response relative to queue size.

Table 4. Estimate of hardware resources requirement to implement the Phase-Adaptive cache algorithm (per adaptable cache / cache pair).

Component	Estimate	Equivalent Gates
24 MRU and Hit Counters (15-bit)	$3n$ (Half-Adder) + $4n$ (D Flip-Flop) = $7n$ each	2,520
11 Adders (15-bit)	$7n$ (Full-Adder) = $7n$ each	1,155
2 8 $\times$28-bit Multipliers (36-bit Result)	$1n$ (Multiplier) + $4n$ (D Flip–Flop) = $5n$ each	360
1 Final Adder (36-bit)	$7n$ (Full-adder) = $7n$ each	252
Result Register (36-bit)	$4n$ (D Flip–Flop) = $4n$ each	144
Comparator (36-bit)	$6n$ (Comparator) = $6n$ each	216
	Total	4,647

Table 5. Architectural parameters for simulated processor.

Fetch queue: 16 entries
Branch mispredict penalty: 9 front-end + 7 integer cycles (10 + 9 for adaptive MCD)
Decode, issue, and retire widths: 8, 6, and 11 instructions
L1 cache latency (I and D): 2/8, 2/5, 2/2, or 2/– cycles, for A and (optionally) B partitions
L2 cache latency: 12/43, 12/27, 12/12, or 12/– cycles
Memory latency: 80 ns (1st access), 2 ns (subsequent)
Integer ALUs: 4 + 1 mult/div unit
FP ALUs: 4 + 1 mult/div/sqrt unit
Load/store queue: 64 entries
Physical register file: 96 integer, 96 FP
Reorder buffer: 256 entries

Table 6. MediaBench benchmark applications. All use the reference data sets.

Benchmark	Simulation window
`adpcm`	`encode` (6.6M) & `decode` (5.5M)
`epic`	`encode` (53M) & `decode` (6.7M)
`jpeg`	`compress` (15.5M) & `decompress` (4.6M)
`g721`	`encode` (0–200M) & `decode` (0–200M)
`gsm`	`encode` (0–200M) & `decode` (0–74M)
`ghostscript`	0–200M
`mesa`	`mipmap` (44.7M) & `osdemo` (7.6M) & `texgen` (75.8M)
`mpeg2`	`encode` (0–171M) & `decode` (0–200M)

Table 7. Olden benchmark applications.

Benchmark	Datasets	Simulation window
`bh`	2048 1	0–200M
`bisort`	65000 0	Entire program (127M)
`em3d`	4000 10	70M–178M (108M)
`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

4. Methodology

The simulation environment is based on the SimpleScalar toolset [4] with MCD processor extensions [27]. These extensions include modifications to model an aggressive superscalar processor, *e.g.*, the Register Update Unit (RUU) has been split into separate reorder buffer (ROB), issue queue, and physical register file structures. They also include a heavy re-write of the time management code to emulate separate clocks for each domain, complete with jitter, and to account for synchronization delays on all cross-domain communication. Table 5 contains a summary of the simulation parameters. These have been chosen, in general, to match the characteristics of the Alpha 21264. More detail on the MCD extensions can be found in prior MCD papers [26, 27].

Tables 6, 7 and 8 specify the benchmarks along with the instruction windows, total number of instructions, and input data sets or parameters for our simulation runs.

The evaluation of the adaptive MCD processor is performed by comparing the relative performance (program execution time) of the best configuration for each application against the fully synchronous processor that provides the best overall performance for our application suite. To find this "best overall" machine we explored a very wide design space: the cross-product of 4 integer issue queue sizes, 4 floating-point issue queue sizes, 4 data / L2 cache organizations, and 16 instruction cache / branch predictor organizations—a total of 1,024 options. (With 32 applications, this portion of our work alone consumed 160 CPU months of simulation time.) The sweep of i-cache configurations in particular ranged from 4KB to 64KB and 1, 2, 3, and 4-way associativity, including many (higher frequency) options not available in the adaptive MCD architecture. The overall best fully synchronous configuration for this suite of benchmarks has a 16-entry integer issue queue, a 16-entry floating-point issue queue, a 64KB direct-mapped instruction cache with a 16-bit *gshare* branch predictor, and a 32KB direct-mapped L1 data cache / 256KB direct-mapped L2 cache. The 64 KB direct-mapped instruction cache is

Table 8. Spec2000 Benchmark applications.

Benchmark	Datasets	Simulation window
Integer		
bzip2	source 58	1000M–1100M
crafty	ref	1000M–1100M
eon	ref	1000M–1100M
gcc	166.i	2000M–2100M
gzip	source 60	1000M–1100M
parser	ref	1000M–1100M
twolf	ref	1000M–1100M
vortex	ref	1000M–1100M
vpr	ref	1000M–1100M
Floating-Point		
apsi	ref	1000M–1100M
art	ref	300M–400M
equake	ref	1000M–1100M
galgel	ref	1000M–1100M
mesa	ref	1000M–1100M
wupwise	ref	1000M–1100M

Table 9. Distribution of adaptive architecture choices for Program-Adaptive.

Integer IQ		FP IQ		D-Cache		I-Cache	
16	85%	16	73%	32k1W/256k1W	50%	16k1W	55%
32	5%	32	15%	64k2W/512k2W	18%	32k2W	18%
48	5%	48	8%	128k4W/1024k4W	23%	48k3W	8%
64	5%	64	5%	256k8W/2048k8W	10%	64k4W	20%

27% faster than the same capacity in the adaptive MCD design, and the misprediction penalty is substantially lower. The data cache is the smallest and fastest configuration.

5. Results

Figure 6 shows the relative improvement in run time of the Program-Adaptive and Phase-Adaptive MCD processors over the best-overall fully synchronous processor. Program-Adaptive configurations are chosen by per-application exhaustive testing across all possible adaptive MCD configurations. Phase-Adaptive results employ the control algorithms described in Section 3. Many applications achieve a significant performance improvement with the adaptive MCD processor: for `gcc` the Program-Adaptive and Phase-Adaptive processors outperform the fully synchronous processor by 42% and 45%, respectively. For `em3d` the corresponding numbers are 45% and 49%.

Table 9 summarizes the distribution across the complete suite of benchmarks of the best MCD configurations for the Program-Adaptive mode (no dynamic phase adjustments). From these results it is clear that many applications perform best with the smallest / fastest domain configurations (first line of the table). At the same time, there are applications that benefit from larger / slower configurations. Often one structure dominates performance. `Gsm encode` and `decode`, for example, have similar performance for all configurations with a 64KB 4-Way instruction cache. While the very best results are achieved in conjunction with the smallest / fastest integer and load / store structures, the differences are minor. `Ghostscript`, similarly, performs well whenever the instruction cache is larger than 32KB; additional increases yield only marginal improvements.

The bias in Table 9 toward the smallest configuration is due to a number of applications having small computation kernels, high instruction level parallelism, and small data sets; *e.g.*, `adpcm encode`, `adpcm decode`, `bzip2`, and `mpeg2 encode`. For these applications the smallest configuration supplies sufficient capacity for efficient processing; thus, they opt for this configuration due to its higher clock frequency. On the other hand, memory intensive applications such as `em3d` perform best with larger / slower structures that significantly reduce the number of cache misses and avoid the associated long latencies.

There are also applications for which Program-Adaptive MCD performs worse than the fully synchronous processor (`jpeg decompress` 2.7%, `gsm encode` 0.1%, `ghostscript` 1.8%, `mesa mipmap` 4.9%, `apsi` 1.9%, `bzip2` 4.8% and `vpr` 6.6%). The most common source of trouble is an inability to deliver instructions to the execution units fast enough. This in turn stems from the inability to upsize the I-cache without increasing its associativity, which exacts a significant cost in clock rate. For applications that need the larger capacity (but not the increased associativity) there is no way to gain back the performance lost due to the decrease in frequency. For all other applications, the Program-Adaptive MCD improves performance relative to the fully synchronous processor, and in many cases it is a significant improvement (`adpcm decode` 30.5%, `em3d` 48.7%, `mst` 43.3%, `art` 32.2%, `gcc` 41.4% and `vortex` 33.1%). Overall, a 17.6% performance improvement is achieved.

5.1. Program- vs Phase-Adaptive

In general, Phase-Adaptive outperforms Program-Adaptive. There are no applications in our suite for which the performance of the Phase-Adaptive MCD is lower than that of the fully synchronous processor, and the overall performance improvement increases to 20.4%. In several applications the improvement is significant: `apsi`, `epic encode`, `ghostscript`, `crafty`, `eon`, `mesa`, `parser`, `twolf`, and `vortex`.

In Figure 7 we plot samples of adaptive behavior to highlight the significance of phases. The plots show structure configuration over time, as measured by the number of committed instructions. In Figure 7(a), `apsi` shows strong periodic phases for its data cache capacity needs. The data / L2

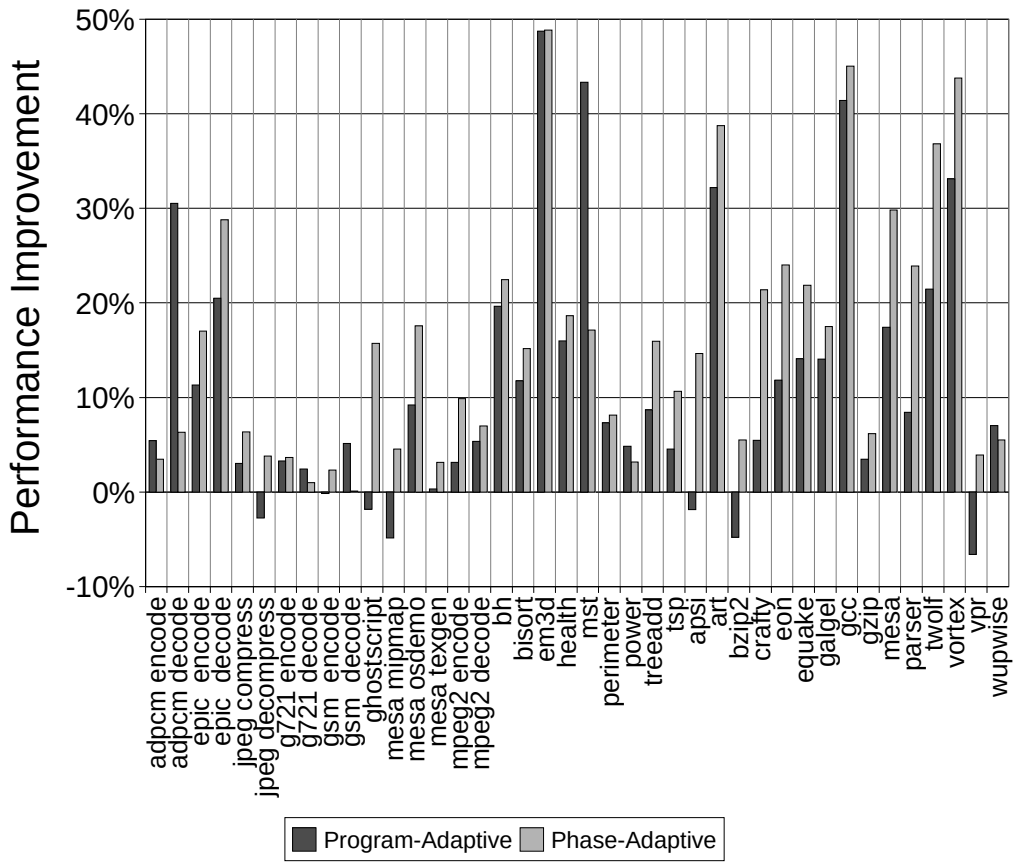


Figure 6. Performance improvement of Program- and Phase-adaptive MCD over fully synchronous.

caches adjust predominately between 32 KB/256 KB 1-way and 128 KB/1 MB 4-way. In Figure7(b), the issue queue for `art` cycles through the four configuration sizes in a regular pattern corresponding to available ILP.

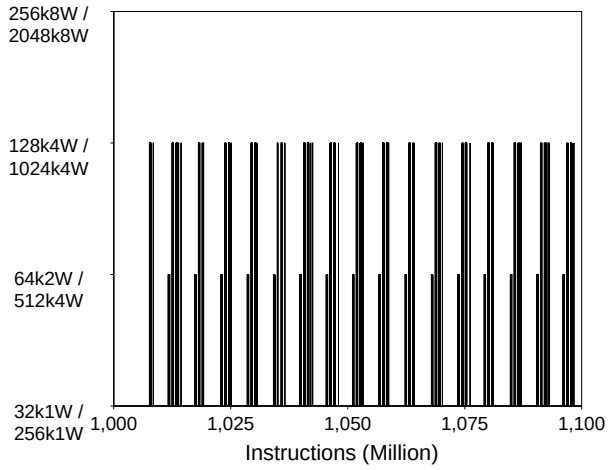
All instruction cache configurations prove useful for improving performance in the Program-Adaptive results: nearly half (45%) of the applications employ an instruction cache other than the smallest. Remarkably, however, in the Phase-Adaptive case only the smallest instruction cache configuration is used, yet the overall performance is *better* than that of Program-Adaptive. This seeming contradiction results from the fact that adapting at phase changes improves throughput enough to cause the processor to speculate through more branches. This in turn results in a larger number of mispredictions (though not necessarily a higher misprediction *rate*) and more overhead due to pipeline flushes. A smaller, faster instruction cache reduces refill overhead by responding faster, thereby increasing overall performance.

Faster pipeline refill is also why the best instruction cache for the fully synchronous processor is one that is both fast and large, at 64 KB direct-mapped. The frequency degradation in the adaptive MCD design to go to a larger instruction cache which is set-associative outweighs the performance benefits for these applications. This suggests, as a topic for future work, that an adaptive instruction cache design that resizes by sets rather than ways [24] might be able to gain additional performance benefits.

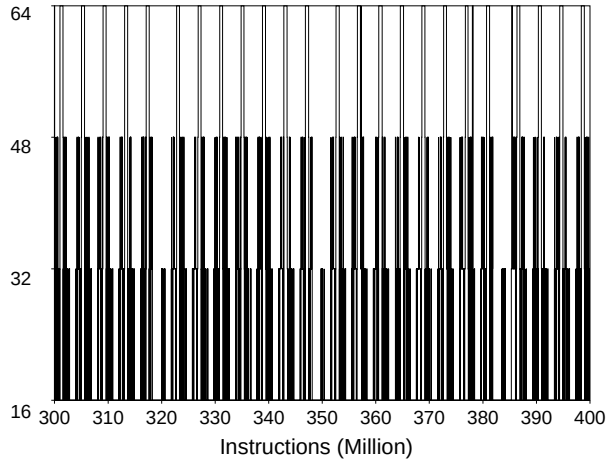
There are a few notable cases where Phase-Adaptive does not perform as well as Program-Adaptive. One such application is `adpcm decode`. The cause is the code kernel in the `adpcm_decoder()` function, which contains a series of data-dependent branches that are difficult to predict:

```
vpdiff = step >> 3;
if ( delta & 4 ) vpdiff += step;
if ( delta & 2 ) vpdiff += step>>1;
if ( delta & 1 ) vpdiff += step>>2;
```

The dynamic cache controller speeds up processing by configuring the data caches to their minimal and fastest sizing. The configuration is the proper configuration relative to the cache domain behavior. This results in a high rate of processing and causes more branches to be speculated,



(a) `apsi` D/L2 cache configurations



(b) `art` Integer IQ configurations

Figure 7. Sample reconfiguration traces

often incorrectly. By running with a larger cache the best synchronous design processes instructions more slowly and suffers fewer mispredictions, resulting in a dramatic decrease in pipeline flushes and significantly better performance. This advantage over the Phase-Adaptive case disappears if the problematic branches are replaced with predicated instructions.

Phase-Adaptive mode is less effective with the application *mst* because of periodic short bursts of cache conflicts in the *A* partition. The cache controller responds by increasing the associativity from direct-mapped to 2-way in order to avoid the costly *B* partition accesses. However, the change occurs in the next interval, after the burst, and the cache configuration ends up flipping back to direct-mapped. This pattern repeats, resulting in a slowdown compared to Program-Adaptive mode.

In summary, the adaptive MCD approach is able to exploit the needs of individual applications better than a globally constrained, fully synchronous processor. The Program-Adaptive MCD processor achieves a performance improvement of 17.6% over the best performing fully synchronous processor. Allowing the MCD processor to adapt to application phases further improves performance to 20.4% over the fully synchronous processor and, more significantly, Phase-Adaptive achieves this improvement without the application profiling of Program-Adaptive mode.

6. Related Work

Several manufacturers, notably Intel [16] and Transmeta [12], have processors capable of global frequency and voltage scaling. Global control works well for saving energy in applications with real-time constraints for which the processor as a whole is over-designed [14, 18]. The goal is to save energy with minimum performance *loss*. Albonesi [1] proposes a *complexity adaptive processor* that adjusts structure capacity while varying global clock frequency and/or access latencies (cycles) for performance optimization.

Childers *et al.* [6] propose trading IPC for clock frequency to save energy. The underlying assumption is that lowering the clock frequency will degrade performance. The user is allowed to select the level of performance degradation to be tolerated. This assumption is correct with a global clock, but in a GALS architecture with decoupled clock domains, decreasing the frequency can give higher performance if resources are scaled up accordingly. Multiple clock domain architectures [3, 15, 26] extend the work of Childers *et al.* by permitting the frequency of each domain to be set independently of the others. Semeraro *et al.* [26] adjust frequencies automatically at run time to reduce energy in domains that are not on the critical path of the instruction stream.

Powell *et al.* [24] incorporate a variable latency data cache. By predicting which way in a set associative cache holds the requested data, an access can be as fast as in a direct-mapped cache. A miss in the predicted set requires an additional access. Balasubramonian *et al.* [2] describe a reconfigurable data cache hierarchy whose access time adjusts with the configuration. They assume a globally synchronous processor. We extend this work by decoupling the clock domains and adjusting the data cache hierarchy, instruction cache, branch predictor, and integer and floating-point issue queues.

Reconfigurable issue queues [5, 9, 11, 23] have also been used to reduce energy. Results indicate that applications vary greatly in their resource requirements. The work of Dropsho *et al.* [9] includes the instruction, data, and L2

caches; issue queues; ROB; and register files. While the focus is energy efficiency, the results demonstrate that application resource requirements vary across these structures. Detailed timing of issue queues as it relates to queue size is explored by Buyuktosunoglu *et al.* [5] and Palacharla *et al.* [22].

Sasanka *et al.* [25] explore the combination of hardware adaptation (of issue queue size and issue width) and global-chip dynamic voltage scaling for multimedia applications. The goal is to meet frame rate processing requirements while minimizing energy dissipation. Our approach, by contrast, improves performance through adaptation and fine-grain dynamic frequency scaling. We argue that to achieve appreciable speedups, fine-grain dynamic frequency scaling using a GALS approach is necessary to limit the frequency effect of upsizing a structure to the local unit level. As demonstrated by Albonesi [1], dynamically trading frequency for complexity in a conventional, single clock design benefits only those applications with a severe performance bottleneck. This paper demonstrates how the adaptive MCD microarchitecture yields significant speedups across a broad range of applications.

7. Conclusions

General-purpose processors are designed to work well across a wide range of applications. The design point arrived at is, by necessity, a compromise since applications exercise the microarchitecture resources in widely varying degrees. The *adaptive MCD architecture* presented in this paper offers designers additional dimensions in which the microarchitecture can be optimized to further improve performance. By separating the major functionality into separate clock domains, clock rate and complexity tradeoffs can be made independently in each domain. By making the dominant structures in each domain adaptive, the same tradeoffs can be made dynamically for each application or application phase. Performance is enhanced by reducing each application phase's particular hardware bottlenecks, whether they are limited by frequency or by hardware resources.

We demonstrate that an adaptive MCD architecture provides performance improvements over any fixed combination of configurations in a fully synchronous design. For our suite of 32 standard benchmarks, the improvement is 17.6% over the best overall fully synchronous processor when adapting once for each application and 20.4% when adaptation is performed at application phases, despite the branch delay and frequency penalties inherent in adaptability. In future work we plan to explore a wider range of adaptive structures, examine the effects of branch predictor resizing in more detail, and consider resizing the instruction cache by sets instead of (or in addition to) ways.

References

[1] D. H. Albonesi. Dynamic IPC/clock rate optimization. In *25th Intl. Symp. on Computer Architecture*, June 1998.

[2] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *33rd Intl. Symp. on Microarchitecture*, Dec. 2000.

[3] L. Bengtsson and B. Svensson. A Globally Asynchronous, Locally Synchronous SIMD Processor. In *3rd Intl. Conf. on Massively Parallel Computing Systems*, Apr. 1998.

[4] D. Burger and T. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, U. Wisc.–Madison, June 1997.

[5] A. Buyuktosunoglu, D. H. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In *11th Great Lakes Symp. on VLSI*, Mar. 2001.

[6] B. R. Childers, H. Tang, and R. Melhem. Adapting Processor Supply Voltage to Instruction-Level Parallelism. In *Kool Chips Workshop*, Dec. 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*, June 2001.

[8] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *29th Intl. Symp. on Computer Architecture*, May 2002.

[9] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[10] M. Fleischmann. LongRun^{TM} Power Management. Technical report, Transmeta Corporation, Jan. 2001.

[11] D. Folegnani and A. Gonzalez. Energy-Efficient Issue Logic. In *28th Intl. Symp. on Computer Architecture*, June 2001.

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

[13] A. Hartstein and T. R. Puzak. The Optimum Pipeline Depth for a Microprocessor. In *29th Intl. Symp. on Computer Architecture*, May 2002.

[14] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scaling. In *Workshop on Power-Aware Computer Systems*, Nov. 2000.

[15] A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *29th Intl. Symp. on Computer Architecture*, May 2002.

[16] S. Leibson. XScale (StrongArm-2) Muscles In. *Microprocessor Report*, 14(9), Sept. 2000.

[17] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. G. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In *30th Intl. Symp. on Computer Architecture*, June 2003.

[18] D. Marculescu. On the Use of Microarchitecture-Driven Dynamic Voltage Scaling. In *Workshop on Complexity-Effective Design*, June 2000.

[19] S. McFarling. Combining Branch Predictors. Technical Report Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[20] V. Milutinovic, D. Fura, and W. Helbig. Pipeline design tradeoffs in a 32-bit Gallium Arsenide microprocessor. *IEEE Trans. on Computers*, 40(11), Nov. 1991.

[21] J. Muttersbach, T. Villager, H. Kaeslin, N. Felber, and W. Fichtner. Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems. In *12th IEEE Intl. ASIC/SOC Conf.*, Sept. 1999.

[22] S. Palacharla, N. Jouppi, and J. Smith. Quantifying the complexity of superscalar processors. Technical Report TR-96-1328, U. Wisc.–Madison, Nov. 1996.

[23] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *34th Intl. Symp. on Microarchitecture*, Dec. 2001.

[24] M. Powell, A. Agrawal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction. In *34th Intl. Symp. on Microarchitecture*, Dec. 2001.

[25] R. Sasanka, C. Hughes, and S. Adve. Joint Local and Global Hardware Adaptations for Energy. In *10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[26] 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 *35th Intl. Symp. on Microarchitecture*, Nov. 2002.

[27] 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 *8th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2002.

[28] T. Sherwood and B. Calder. Time Varying Behavior of Programs. Technical Report UCSD-CS99-630, U. Cal. San Diego, Aug. 1999.

[29] A. E. Sjogren and C. J. Myers. Interfacing Synchronous and Asynchronous Modules Within A High-Speed Pipeline. In *17th Conf. on Advanced Research in VLSI*, Sept. 1997.

[30] E. Sprangle and D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *29th Intl. Symp. on Computer Architecture*, May 2002.

[31] S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE J. of Solid-State Circuits*, May 1996.

[32] B. Xu and D. Albonesi. Runtime reconfiguration techniques for efficient general purpose computation. *IEEE Design and Test of Computers*, Jan. 2000.

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