Application Adaptive Energy Efficient Clustered Architectures

Diana Marculescu
Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA 15213-3890
dianam@ece.cmu.edu

ABSTRACT

As clock frequency and die area increase, achieving energy efficiency, while distributing a low skew, global clock signal becomes increasingly difficult. Challenges imposed by deep-submicron technologies can be alleviated by using a multiple voltage/multiple frequency island design style, or otherwise called, globally asynchronous, locally synchronous (GALS) design paradigm. This paper proposes a clustered architecture that enables application-adaptive energy efficiency through the use of dynamic voltage scaling for application code that is rendered non-critical for the overall performance, at run-time. As opposed to task scheduling using dynamic voltage scaling (DVS) that exploits workload variations across applications, our approach targets workload variations within the same application, while on-the fly classifying code as critical or non-critical and adapting to changes in the criticality of such code portions. Our results show that application adaptive variable voltage/variable frequency clustered architectures are up to **22%** better in energy and **11%** better in energy-delay product than their non-adaptive counterparts, while providing up to **31%** more energy savings when compared to DVS applied globally.

Categories and Subject Descriptors: C.1.3 **[Other Architecture Styles]:** Adaptable architectures, Pipeline Processors.

General Terms: Algorithms, Design, Performance, Measurement.

Keywords: Dynamic voltage scaling, Clustered architectures.

1 INTRODUCTION

Driven by technology scaling and increased complexity, achieving power efficiency has become an increasingly difficult challenge, especially in the presence of increasing die sizes, higher clock speeds and variability driven design issues. To cope with these challenges, a design style based on multiple voltage/frequency islands has been proposed recently [1]. In addition to controlling better local clock skews and allowing for local performance optimizations, a multiple voltage/multiple frequency island design style may enable application-driven adaptation for better energy efficiency.

In support of a frequency island design style, a globally asynchronous, locally synchronous (GALS) approach may serve as an intermediate step between fully synchronous and fully asynchronous designs, while at the same time providing local adaptation capabilities. In a GALS design, several independently clocked regions communicate with each other asynchronously. This approach reduces the problems associated with a global clock distribution network while allowing designers to use synchronous design techniques for each of the synchronous clock domains. Such an approach has been studied in the case of high-performance processors [2,3], with beneficial impact on power efficiency. At the same time, in the case of superscalar, out-of-order processors, the complexity of the dynamic scheduling hardware limits frequency growth [4]. In addition to limiting the performance of such an architecture, complex structures also consume more power than simpler equivalent structures. One way to avoid complexity is to partition or cluster resources into groups. Partitioning reduces the size of monolithic structures and the control logic associated with them. However, performance of a clustered architecture is sensitive to the steering mechanism it uses.

Combining the potential power savings of removing the global clock with savings due to reducing the complexity of a given design, while also allowing for local optimizations and workload-driven adaptation, is poised to yield power efficient architecture configurations. While several GALS designs have been proposed and have yielded reasonable power-performance tradeoffs if the different synchronous regions can be clocked at different speeds [2,3,5], previous work has looked at clustering hardware (and thus, executed instructions) based on type (i.e., integer, floating-point, memory), as opposed to clustering based on *importance* or *criticality*. Recent studies have shown that some instructions have less impact on execution time than others. These instructions, called *non-critical* instructions, may be run at a lower speed (and thus, lower power), without impacting performance significantly. However, the use of an *application adaptive* control mechanism for *dynamically* selecting the speed (or voltage) of the non-critical cluster has not been explored, nor implemented.

The contribution of this paper is twofold:

- Explore the impact of a GALS design style on a **clustered architecture** that uses an explicit *dynamic critical path prediction* mechanism to steer instructions to one of two existing clusters, depending on their criticality or importance to the overall performance.
- Propose **application adaptive**, dynamic control mechanisms able to adjust the speed/voltage of the cluster running non-critical code such that it matches the overall application profile better.

This paper is organized as follows. Section 2 overviews existing related work. In Section 3, the proposed architecture is described, with emphasis placed on the selection of clock domains, asynchronous communication mechanisms, and resource clustering parameters. Section 4 introduces several types of dynamic critical path predictors and power model for each of them, while also describing the dynamic control mechanisms. Section 5 describes the experimental methodology and an evaluation of the application adaptive variable voltage/frequency clustered architecture, when compared to the non-adaptive counterpart. Finally, Section 6 concludes the paper and outlines possible directions for future work.

2 RELATED WORK

Although the multiple voltage/frequency island design styles has been introduced recently [1], the GALS design paradigm has a history of a couple of decades [6]. The impact of using a GALS design style for power efficient ASICs has been explored in [7], while more recently the concept has been applied to high performance, superscalar processors [2,3,5,8]. There, the idea of using varying speeds and local voltages has been explored in the context of an Alpha-like architecture, with back-end clustering based on the type of instructions (e.g., integer, floating point, or memory), and not their importance or criticality to overall performance.

ISLPED'04, August 9–11, 2004, Newport Beach, California, USA.

Casmira and Grunwald [9] investigated applying scheduling slack to dynamic instruction issue. If an instruction had sufficient slack, then it was sent to a half-speed functional unit; otherwise, it went to a full-speed one. Instructions with sufficient slack can then be considered non-critical. Casmira and Grunwald's [9] determination of criticality is localized; it is computed within a 24–entry instruction window. The method used by Fields *et al.* [10], however, is global, which makes it more accurate and robust. [9] also only uses functional units of different speeds, while the architecture presented in this paper has the ability to run entire sections of the processor at different speeds and dynamically change them. Tune *et al.* [11] propose several heuristics that can be used to dynamically predict the critical path. The heuristics are localized, but the impact of this is mitigated by their sensitivity to machine resources. While the heuristics are less accurate and robust then the global method proposed by Fields *et al.* [10], their simplicity may cause them to be more power efficient to implement. Fields *et al.* [12] discuss slack as a measure of criticality. Criticality can only divide instructions into two classes: critical and non-critical. If a measure of criticality is known, however, multiple classes can be defined, leading to more powerful control strategies. Such a strategy could be used to steer instructions to more than two clusters. Later on, Tune *et al.* [13] introduce the concept of *tautness*: slack measures how much an instruction can be delayed before it becomes critical; *tautness* measures how much an instruction can be optimized before it becomes non-critical. Using these metrics, [13] compares several critical path predictors and comment on the accuracy of several prediction and training mechanisms.

3 BASELINE MICROARCHITECTURE

In this section, we describe the proposed clustered architecture that uses a dynamic explicit critical path prediction mechanism to steer instructions, a GALS clocking scheme and support for dynamic control of the non-critical cluster speed

3.1 Clock Domain Selection

In a standard pipelined out-of-order superscalar processor, a series of events takes place when handling an instruction. First, one or more instructions are fetched from the instruction cache. A branch predictor allows fetching to continue even when the next address to be fetched is not precisely known. Next, instructions are decoded to determine their purpose and registers are renamed to reduce false dependencies. If an operand register's content is valid, it is read from the register file and it travels through the pipeline with the instruction; otherwise, a flag is set and the instruction watches for the result to be produced. After register renaming, instructions are dispatched to an issue queue. When all operands of an instruction are ready and the appropriate functional unit is available, the instruction is woken up and is marked ready to execute. Based on a given policy, ready instructions are selected for execution during a given cycle. Once selected, the instruction issues, executes, and writes its result back to the register file. Results also traverse data bypass paths to waiting instructions in the issue window. Finally, instructions commit.

The above description suggests some natural boundaries in the pipeline. For instance, a decoupling buffer can often be inserted between the fetch and decode stages. The pipeline also has closely coupled regions that should not be separated. The issue and execution logic, for example, must be able to quickly share information back and forth. In addition to the boundaries suggested by the architecture's functionality, boundaries are also implied by current/future technology trends. As feature sizes shrink and die areas increase, it takes more cycles for a signal to propagate from one end of the die to the other. Matzke [14] studies this trend and suggests that, for a 0.13um process, only 33% of the die length will be reachable in a single clock period (assuming a 1.2GHz clock); for a 0.1um process, only about 16% of the die length is reachable in a single clock period. The trend of increasing wire delay has several implications. First, it suggests a minimum number of clock domains for a given technology, if a signal must be able to propagate throughout the entire clock domain in a single cycle. Second, it implies that architectures with distributed rather than centralized structures are likely to have better performance, as localization will be a key feature of future successful designs.

With these implied boundaries and limitations in mind, the clock domains were chosen as shown in Table 1. The selection of domains is similar to that made by Iyer *et al.* [2,8] and Semeraro *et al.* [3,5]. A significant difference, however, is that the back-end clock domains are clustered based on *criticality* rather than *type*.

Table 1. Stages and clock domains in the processor architecture

Stage	Operation	Domains
1, 2, 3	Fetch from I-cache. Predict branch outcomes.	1
4	Decode. Speculatively update branch predictor.	2
5, 6, 7	Rename registers. Read from register file.	2
8	Dispatch to clusters.	2
9, 10	Dispatch to issue queues.	3, 4
11	Issue to functional units.	3, 4
12	Execute.	3, 4
13	Wakeup. Writeback.	3, 4
14	Recover from branch mispredicts. Write to register file. Commit.	2

Figure 1 shows what resources are included in each clock domain of the GALS architecture and illustrates the flow of information within and between domains. The proposed architecture currently has fourteen stages to reflect current trends for increasingly deep pipelines (such as the Pentium 4 [15], which has more than 20 pipeline stages). The first clock domain consists of the fetch logic and takes three stages. The branch prediction unit and level 1 and level 2 instruction caches are located here; if a trace cache were present, as in the Pentium 4 [15], it would also be in this clock domain. The second clock domain contains the rest of the front end—decode, rename, and dispatch stages—as well as the commit stage. We justify grouping the decode stage with other stages by assuming a RISC architecture, or a CISC architecture with a trace cache, which greatly simplifies the decoding logic. The rename step takes three stages because trends suggest that additional cycles will be needed to access large data structures (such as renaming tables) in future designs. The register file is included in the second domain because it is closely coupled with the rename logic and the commit logic. Performing register read at rename, rather than at issue, also removes the need to have multiple copies of the register file in a two-cluster configuration.

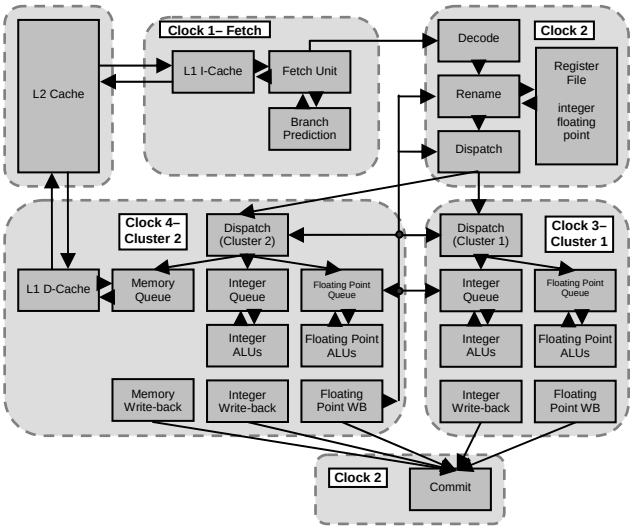

Figure 1. A clustered architecture with four clock domains. Cluster 1 runs non-critical code, while Cluster 2 runs critical code and includes the D-cache (memory operations are considered critical).

The dispatch stage is responsible for steering instructions based on criticality to one of the two remaining clock domains (Cluster 1 and Cluster 2, respectively). The third clock domain is the non-critical one

and does not handle any memory operations. Instead, the fourth clock domain, which handles critical instructions, becomes responsible for all memory operations. Placing all of the memory operations within a single clock domain removes the need to devise a way to have more than one domain share the data cache (such as having two copies of the data cache or using inter-cluster communication to handle non-critical memory accesses), which improves both power consumption and performance. Other than the changes to memory, the third and fourth clock domains in the two-cluster configuration are very similar to the third clock domain in the one-cluster configuration.

3.2 Asynchronous Communication Mechanism

In a design that uses a GALS clocking scheme, clocking domains use asynchronous communication mechanisms to communicate. These mechanisms have been the focus of a considerable amount of research and a wide variety are available, such as synchronization FIFOs [16], stretchable clocks [17], and pausable clocks [6,18]. In this paper, we assume an asynchronous FIFO queue, based on the low-latency token-ring implementation proposed by Chelcea and Nowick [19]. The FIFO is composed of multiple cells organized as a circular queue, with each cell implemented as a central latch and its associated control logic. Successive write operations are performed to successive cells at the head of the circular queue, until all of the cells are full. The read operations are serviced in the same way, starting from the tail of the structure. If the FIFO queue is neither full nor empty most of the time, it can provide the low latency offered by synchronizers while maintaining a high bandwidth through pipelining. A similar synchronization mechanism has been used before [2,3,5,8].

3.3 Resource Clustering

When the architecture is split into two-clusters, the same total number of resources is present, but distributed between the two clusters (Table 2). While there are a number of ways to distribute the resources, our work places three-quarters of the available resources in the first cluster (the non-critical cluster) and one-quarter of the available resources in the second cluster (the critical cluster). As discussed above, memory resources are an exception. The 3:1 division of resources reflects the fact that an estimated 75% of instructions are non-critical [10,11]. Results from previous work [20] also suggest this partitioning to be reasonable, while pointing the fact that critical code does not need to rely on dynamic scheduling for increasing performance due to tight dependencies on critical path.

Table 2. Processor configuration for the clustered architecture

Fetch/Decode/Rename /Dispatch/Commit	4
Issue	12 (4 int, 4 fp, 4 mem)
L1 Instruction Cache	32KB, 2-way SA; 32B blocks; 1 cycle latency
L1 Data Cache	32KB, 4-way SA; 32B blocks; 1 cycle latency
L2 Unified Cache	256KB, 2-way SA; 64B blocks; 6 cycles latency
Physical register file	72 integer, 72 floating-point
Branch predictor	Alpha 21264 like: Combined predictor which uses a bimodal and a 2-level PAg
Non-critical resources : Critical resources	
Integer ALUs	(3 + 1 mult/div units : 1 + 1 mult/div units)
Floating-point ALUs	(3 + 1 mult/div units : 1 + 1 mult/div units)
Load/Store Units	(0 : 2)
Integer issue queue	(15 entries : 5 entries)
FP issue queue	(12 entries : 4 entries)
Mem issue queue	(0 entries : 16 entries)

Instructions are distributed between the two clusters via *dynamic steering logic* during the first dispatch stage. A dynamic steering scheme can adapt to the state of the pipeline better than a static scheme, so it can minimize the number of inter-cluster communications and balance the workload better. This translates into improved performance. Several dynamic steering algorithms have been explored in the literature, including random steering and register dependence (or register mapped) steering, which has a number of variations [21]. Our work uses *pure criticality* to steer instructions. Criticality is dynamically predicted using the algorithms discussed in Section 4.

Often, load balancing is necessary in clustered architectures to be sure that resources are not underutilized. Our results show that only in 10-15% of the cases a load balancing mechanism would help alleviate potential bottlenecks, hence we decided against using a load balancing technique.

4 DYNAMIC CRITICAL PATH PREDICTION

A critical instruction is an instruction whose data dependencies and/or resource requirements induce performance bottlenecks. Since the criticality of an instruction in case of dynamically scheduled processors is partly determined by the characteristics of the microarchitecture, criticality cannot be determined at compile time. The dynamic critical path prediction mechanisms used in this paper are similar to the work done by Fields *et al.* [10], in addition to a lower cost, but sufficiently accurate version that we propose. The token-based predictor [10] was chosen because of its accuracy and robustness. As it will be seen later, however, to be accurate, its size and complexity become prohibitive due to its power overhead.

4.1 Critical vs. Non-Critical Instructions

To characterize the criticality of instructions, a dependency graph can be used. In such a directed graph, instructions constitute the nodes, while edges show the dependency information. The weight of each edge is the time needed to resolve the dependence. The challenge in the case of superscalar processors is that these weights are dependent on the run-time conditions, and, for better results, dynamic versions of such graphs should be constructed and managed on the fly [10]. Given such a graph, the critical path is the longest weighted path from the first dispatch node to the last commit node; any instructions on the critical path are considered critical

To be able to classify instructions in critical or non-critical, the impact of delaying each one of them on the overall execution time should be assessed. Given the complexity of typical dependency graphs, especially for dynamically scheduled processors, heuristics can be used to keep track of the most relevant information. For example, the approach in [10] assumes that the execution time of all instructions is increased by one cycle. Then, the latency of the critical and non-critical instructions is decreased in turn and the change in performance is compared. Decreasing the latency of critical instructions should have a much larger impact on performance than decreasing the latency of the non-critical instructions. As one might expect, however, there is not a one-to-one correspondence between the total number of cycles removed from execution time and the total number of latency cycles removed from critical instructions. Instead, this ratio provides a measure of how dominant the critical path is. Near critical paths may emerge if the dominant critical path is optimized, limiting the performance gain. Additionally, some performance improvement may be seen by reducing the latency of the non-critical instructions. This characteristic exists because the model does not capture all possible architectural dependencies, allowing some critical instructions to escape notice.

Two additional key observations, that reduce the complexity of hardware based predictors, should be considered. **First**, the weights of the edges in the dependency graph are irrelevant – only the order in which the edges arrive is meaningful. What matters is when an instruction completes, not how long it takes to do so. A multi-cycle multiply instruction may be non-critical, for example, if it produces a required result before a single-cycle addition instruction does. If an edge is on the critical path, then it must be last-arriving. Otherwise, the edge could be delayed without penalty, which contradicts the definition of criticality. Likewise, if an edge is not a last-arriving edge, then it is not critical. A long chain of last-arriving edges is therefore likely to be part of the critical path, and instructions in the chain are likely to be critical. **Second**, if criticality is to be analyzed and detected in hardware, it is not possible for a dependency to span more than the number of instructions in flight between decode and commit as it would exceed the available state space in the dynamically scheduled

processor (this number is usually characterized by *rob_size*, the size of the reorder buffer that holds instructions from the time they are decoded until they have committed). These two observations have proven to be useful for reducing the complexity of critical path predictors. In addition, critical instructions are typically tightly coupled in a dependency chain that is unlikely to benefit from an out-of-order, dynamically scheduled microarchitecture [20]. Hence, an in-order issue for the critical cluster is enough, without any significant performance penalty.

4.2 Token-Based Critical Path Prediction

The dynamic critical path prediction hardware as defined in [10] can be divided into two parts: prediction and training. The prediction hardware is simply an array of hysteresis counters indexed by the program counter and accessed during the fetch stage. If the hysteresis counter is above a certain threshold, the instruction is predicted critical; otherwise, it is predicted non-critical. The information is stored in a Critical Path Prediction table which is similar in functionality to the Branch Target Buffer.

The training hardware is more complicated and is implemented as a special array [10]. Training is performed in the commit stage via sampling. As each instruction commits, it is added to the token array in FIFO fashion. Next, a token is planted into the entry corresponding to the committing instruction, if possible. The bit of the training array entry corresponding to the available token must then be set for the committing instruction. Then, as other instructions commit, the token is propagated forward along all last-arriving edges. When an instruction commits, it knows which instructions in the dependency graph are the sources for its last-arriving edges. To propagate tokens, the token trainer array entry for the given source instruction is read and any tokens in the corresponding entry are copied to the entry of the destination instruction for the currently committing instruction. Once the processor has committed a certain number of instructions, with respect to the planting of the token, the token is checked for liveness. If the token is still alive, then it is likely that the instruction that it was planted in is critical; otherwise, the instruction is likely to be non-critical. Finally, the prediction array is updated to reflect the training decision, and the token is freed.

In our case, a coarse grain approach—instead of a fine grain one—was used to roughly estimate the accuracy of the predictor. Instead of changing the latency of individual instructions, the speed of an entire cluster was changed. Doubling the speed of the critical cluster (which has a third of the functional units), created, on average, a performance increase of 9.9%. Doubling the speed of the non-critical cluster (which has three functional units and is responsible for roughly three quarters of the total number of instructions executed) created a performance increase of only 8.0%. This result implies that the dynamic critical path predictor is working as intended.

4.3 Dependency-Based Critical Path Prediction

Although very accurate, the token-based predictor described before has the potential of becoming a significant source of additional power overhead due to the extra arrays needed to store token information. On the other hand, if structures are downsized for increased power efficiency, the prediction accuracy decreases. As suggested before [20], an alternative, less expensive predictor may be employed, based on (for example): "age" of instructions in the issue window; dependency chain length; dependency sub-tree size (i.e., the number of dependent instructions); wake-up information (i.e., the number of instructions woken-up when bypassing results). Results indicate that marking the oldest, not ready to issue instructions in the issue window as critical fares the best [20]. The only hardware support needed for this (in addition to the Critical Path Prediction table storing information about criticality of instructions) is a set of counters whose bitwidth depends on the issue window size. One downside of such predictors is that they see a smaller window of the dependency graph (i.e., only instructions present in the issue window, and not the entire set of in-flight instructions).

We propose a modified version of this heuristic predictor which turns out to be similar in performance and accuracy with the token based predictor. In addition to marking instructions at the bottom of the issue queue as critical, our proposed heuristic also marks as critical instructions that wake-up instructions already scheduled for running in the critical cluster. A similar, but not identical, policy has been used before [20], in that instructions waking up the oldest instructions in the issue window were also marked as critical.

Our results show that, for the pipeline considered in this paper, the proposed dependency based heuristic fares slightly better on average than the token based predictor, and without the extra hardware structures imposed by it.

4.4 Practical Considerations

The Critical Path Prediction table used in both token-based and dependency-based prediction works much like the branch target buffer (BTB), so its power consumption is modeled in the same way, as a simple array structure. Six bit counters with hysteresis are used to follow the criticality of a given instruction (Table 3).

While Fields *et al.* [10] use a standard monolithic reorder buffer, in our case, however, *rob_size* is the total number of pipeline registers after the decode stage, including the issue queues. It should be noted that some pipeline registers are part of a larger queue and that all entries of the queue are included in the total.

Because of token management, the token training array is too complex to be treated as a simple array and is therefore handled as a content addressable memory (CAM). Thus, the token training array has (*number of nodes per instruction* x *commit width*) read ports and (*commit width*) write ports. Assuming that clock gating is used on a per cycle basis, the token training array's power consumption can be scaled by the number of instructions committed in a cycle, yielding a significant reduction in power. Also, the number of ports can be chosen so as to handle only the average commit width, dropping extra information or caching it for delayed processing; such a step would understandably reduce the accuracy of the predictor. Since the number of read and write ports is proportional to the commit width, the power consumption is still quite large and scales poorly, even if these power saving techniques are employed.

Table 3. Dynamic critical path prediction hardware characteristics

Token- and dependency-based predictors	Critical Path Prediction Table	1.5KB in size (2K entries x 6b per counter x direct mapped); 1 read/write port
	Hysteresis	6 bit counters (saturate at 0 and 63) Increment by 8 when trained as critical Decrement by 1 when trained as non-critical Predict as critical if hysteresis is above 8
Token-based predictor	Token Array	416B in size (104 entries x 32b per entry x direct mapped); 12 read ports; 4 write ports
	Token information	128B in size (8 entries x 128b per entry x direct mapped); 4 read/write ports
	No. tokens used	8
	Planting of tokens	If a token is available and the instruction does not have a token already, then plant a token in the execute node.
	Liveness check for tokens	After 185 committed instructions (1.75 x *rob_size*)

A final piece of hardware needed for dynamic critical path prediction (which was not mentioned in [10]), is the token management hardware, which tracks free tokens and liveness counters. We have considered this piece of logic as a four-ported array that is accessed every cycle that an instruction commits, with no clock or power gating capabilities. A complete summary of the dynamic critical path prediction hardware's characteristics is provided in Table 3.

4.5 Non-Critical Cluster Dynamic Control Algorithms

Previous work on GALS superscalar processors has explored the possibility of achieving better power efficiency through the use of

dynamic control algorithms for local speeds and voltages, so as to match a given application's profile. A threshold based control algorithm triggered by issue queue occupancy has been proposed in [8], while an aggressive attack-decay algorithm based on monitoring overall performance counters has been used in [5]. The idea of queue occupancy monitoring for voltage scaling is not new – in a real design case [22] such a control mechanism has been successfully used in conjunction with self-timed interface-based GALS systems.

For the purpose of providing adaptability depending on the application profile, we have considered two control algorithms (Figure 2). The **relative**-threshold based control algorithm checks the *ratio* of issue queue occupancy in the current and previous monitoring interval. While this is similar to the attack-decay algorithm presented in [5], it does not check for limits on performance degradation as the non-critical cluster produces a 0.08% overall performance degradation, for each 1% slowdown factor (as described in Section 4.2), so the impact on overall performance can be quantified easily, without keeping another performance counter. The main idea is to slowdown progressively if the issue queue occupancy is within a certain relative factor (*down*), while ramping up the speed by more than one power state when occupancy increases significantly (by a relative factor of *up*).

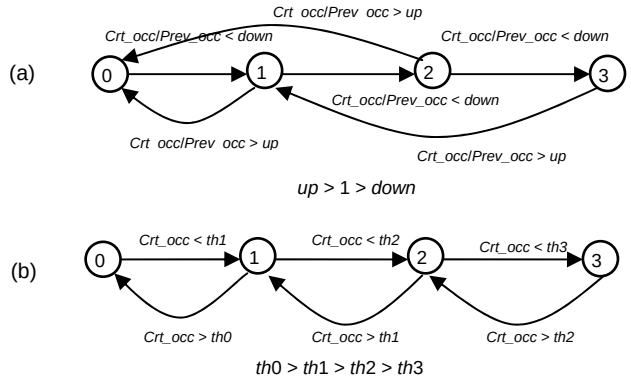

Figure 2. The relative- (a) and absolute-threshold (b) based control algorithms. In both cases, state zero is the highest speed/largest voltage, while state three is the lowest speed/lowest voltage ($V_{dd,0} > V_{dd,1} > V_{dd,2} > V_{dd,3}$)

The **absolute**-threshold based algorithm checks the average issue queue occupancy and, if below a certain *low* threshold, the non-critical cluster is put in the next low power state. Otherwise, if the average occupancy is larger than a high threshold, the non-critical cluster is ramped up to next high power state.

Table 4. Dynamic control algorithms settings

Adaptation interval	10K instructions
Slowdown factors	1, 1.5, 2, 2.5
Relative difference factors for the relative threshold-based alg.	1.1 (*up*) and 0.8 (*down*)
Threshold values for the absolute-threshold based alg.	Int: 4, 7, 10, 12 FP: 2, 3, 5, 6

For the results presented in this paper, we have assumed four power states characterized by relative slowdown factors as shown in Table 4. The voltages corresponding to each state have been determined based on the delay-voltage dependency: $Delay \propto V_{dd}/(V_{dd}-V_t)^{\alpha}$ (for a baseline value of V_{dd} = 3.3V, V_t = 0.55V and α = 1.4 in a 0.18um technology) and assuming slowdown factors as in Table 4. The threshold values (*th*0, *th*1, *th*2, *th*3) have been computed based on the threshold values for the integer and FP issue queues given in Table 4 (e.g., *th*0 is the sum of the highest thresholds for integer and FP, or 12+6 = 18; similarly, *th*3 is the sum of the lowest threshold values for integer and FP issue queues or 4+2 = 6.) As assumed before [2,3,5,8], we assume that the system does not stop working while the non-critical cluster voltage or speed are changing.

5 EXPERIMENTAL RESULTS

The SimpleScalar toolset [23] was used to create a cycle-accurate model of a fourteen stage superscalar pipeline with out-of-order execution. Out-of-order execution was accomplished through the use of issue queues and a register file instead of the default register update unit/load store queue (RUU/LSQ) combination. The configuration of the proposed architecture has been summarized in Tables 2-4.

The GALS clustered architecture was simulated using an event-driven simulation engine, similar to the one described previously [2,3,5,8]. For the results presented in this paper, each clocking domain in the GALS processor is run at the same frequency, except for the non-critical cluster clock domain. Asynchronous FIFO models based on work done by Chelcea and Nowick [19] were used to model the synchronization penalties between the different clock domains contained in the GALS architecture.

The Wattch [24] power estimation extensions were also included so that the power dissipation of the proposed architecture could be evaluated. The power consumption of the critical path prediction hardware is modeled as discussed in Section 4.4. A subset of the SPEC2000 benchmark suite (*gzip*, *vpr*, *mesa*, *equake*, *vortex*, *bzip2*) was used to assess the performance and power dissipation of the proposed architecture. Reference input sets were used and the benchmarks were run for 100M instructions, after fast-forwarding over 500M instructions.

To assess the feasibility of our proposed approach, we have investigated:

- The impact of the **critical path predictor** on the overall energy cost and performance of the clustered GALS architecture.
- The impact of the dynamic **control algorithm** for the non-critical cluster in terms of both performance and energy cost.

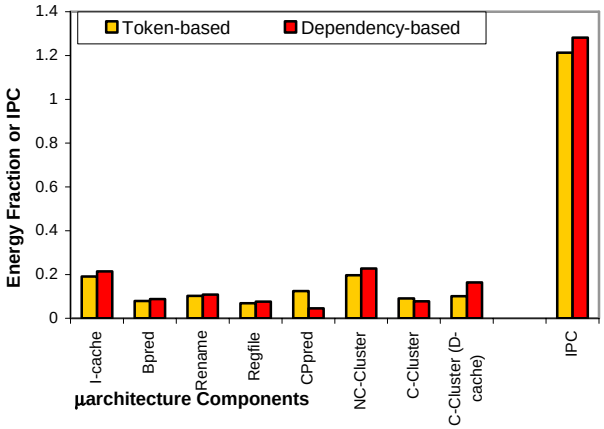

Figure 3. The impact of the critical path predictor implementation on the energy cost and overall performance

We show in Figure 3 the breakdown of the total energy budget into major components for the two hardware based critical path predictors used in this paper: **token-based** and the newly proposed one, **dependency-based**. The configuration used for the two predictors has been included in Table 3. As it can be seen, while performance (IPC or Instructions Committed per Cycle) is on average about the same (with the dependency-based case slightly better), the power cost associated to the token-based predictor is much higher: 12% vs. 4% (for the dependency-based predictor) of the overall energy budget. Furthermore, the total energy consumed across all benchmarks considered is slightly better in the case of dependency-based than for the token-based criticality predictor. For the microarchitecture considered in this paper, the token-based predictor does not seem to be justified in terms of cost and a simple, dependency-based predictor is sufficiently good, with only the overhead of the Critical Path Predictor

table (Table 3). For this reason, the next set of results assumes a dependency-based critical path predictor.

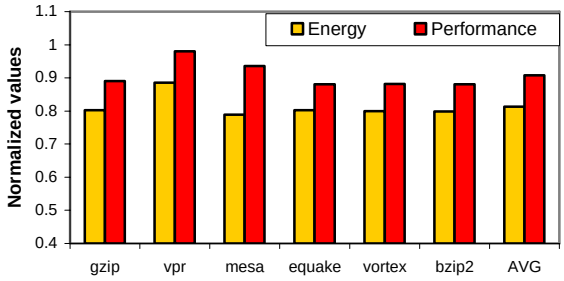

Figure 4. Normalized energy and performance for the application adaptive clustered architecture using *relative threshold*-based control. The baseline is the original GALS clustered architecture with dependency-based criticality predictor, without control.

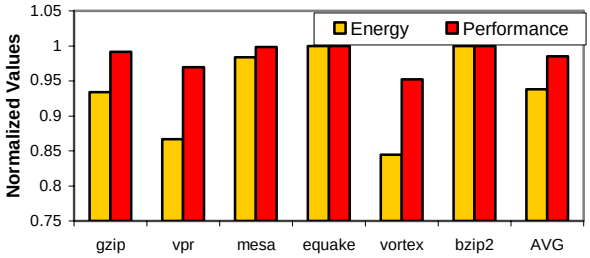

Figure 5. Normalized energy and performance for the application adaptive clustered architecture using *absolute threshold*-based control. The baseline is the original GALS clustered architecture with dependency-based criticality predictor, without control.

We show in Figures 4 and 5 the results of analyzing the two proposed control algorithms described in Section 4.5. The parameters considered in the two cases have been described in Table 4. As it can be seen, the relative threshold based algorithm is more aggressive and keeps the non-critical cluster for most applications in states characterized by a large slowdown factor, yielding energy savings of up to **22%** (in case of *mesa*, for example), with performance loss of **10%** on average. On the other hand, the absolute-threshold based control algorithm is more conservative and it does not allow the non-critical cluster to go into a state characterized by a large slowdown value. In this case, the energy savings reaches **17%** (in case of *vortex*), with an average performance penalty of only **2%**. In terms of energy-delay product, the application adaptive clustered architecture with relative threshold based control is **11%** *better* than the non-adaptive clustered architecture, while the absolute-threshold based case is **5%** *better* than the non-adaptive counterpart. Although not pictured, when compared to globally applied DVS, our application adaptive, *localized* version is up to **31%** *better* in terms of overall energy savings.

6 CONCLUSION

In this paper we have proposed an application adaptive clustered architecture based on a GALS design style, able to dynamically match the speed (and thus voltage) of various portions of code with the application profile. Based on hardware based critical path prediction, instructions rendered non-critical for the overall performance of the application are steered to a non-critical cluster whose speed and voltage can be varied dynamically depending on several runtime factors. Results show that such an architecture is up to 22% more energy efficient than its non-adaptive counterpart and has 11% better energy delay product.

7 ACKNOWLEDGEMENTS

This research was supported in part by SRC Grant No. 2001-HJ-898 and by NSF CAREER Award No. CCR-008479. The author would like to thank Katrina Zwicker and Adam Stoler for contributing to an earlier version of this work.

8 REFERENCES

[1] D. Lackey, *et al.*, "Managing Power and Performance for System-on-Chip Designs using Voltage Islands,' in Proc. Intl. Conf. on Computer-Aided Design (ICCAD), pp. 195-202, Nov. 2002.

[2] A. Iyer and D. Marculescu, "Power and performance evaluation of globally asynchronous, locally synchronous processors," in Proc. Intl. Symp. on Computer Architecture (ISCA), pp. 158-170, May 2002.

[3] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," in Proc. Intl. Symposium on High-Performance Computer Architecture (HPCA), pp. 29-42, Feb. 2002.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in Proc. Intl. Symposium on Computer Architecture, ACM Press, pp. 206-218, June 1997.

[5] G. Semeraro, D.H. Albonesi, S.G. Dropsho, G. Magklis, S. Dwarkadas, M.L. Scott, "Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture," in Proc. Intl. Symposium on Microarchitecture (MICRO), pp. 356-367, Nov. 2002.

[6] D. M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems", PhD Thesis, Stanford University, Oct. 1984.

[7] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist, "Lower Power Consumption in Clock By Using Globally Asynchronous Locally Synchronous Design Style," in Proc. Design Automation Conference (DAC), pp. 873-878, June 1999.

[8] A. Iyer and D. Marculescu, "Power Efficiency of Multiple Clock, Multiple Voltage Cores," in Proc. IEEE/ACM Intl. Conference on Computer-Aided Design (ICCAD), pp. 379-386, San Jose, CA, Nov. 2002

[9] J. Casmira and D. Grunwald, "Dynamic Scheduling Slack," in Proc. Kool Chips Workshop, in conjunction with MICRO 33, Dec. 2000.

[10] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction," in Proc. Intl. Symp. on Computer Architecture (ISCA), pp. 74-85, July 2001.

[11] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," in Proc. Intl. Symposium on High Performance Computer Architecture (HPCA), pp. 185-196, Jan. 2001

[12] B. Fields, R. Bodik, and M. D. Hill, "Slack: Maximizing Performance under Technological Constraints," in Proc. Intl. Symposium on Computer Architecture (ISCA), pp. 47-58, May 2002.

[13] E. Tune, D. Tullsen, and B. Calder, "Quantifying Instruction Criticality," in Proc. Intl. Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 104-116, Sept. 2002.

[14] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," in IEEE Computer, 30(9):37-39, Sept. 1997.

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium4 Processor," in Intel Technology Journal, Q1 2001.

[16] R. Kol and R. Ginosar, "Adaptive Synchronization for Multi-Synchronous Systems", 1998 IEEE Intl. Conference on Computer Design (ICCD'98), pp. 188-189, Oct. 1998.

[17] D. S. Bormann and P. Y. K. Cheung, "Asynchronous Wrapper for Heterogeneous Systems", Proc. Intl. Conference on Computer Design (ICCD), IEEE Computer Society Press, pp. 307-314, Oct. 1997.

[18] J. Seizovic, "Pipeline Synchronization", Proc. Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 87-96, November 1994.

[19] T. Chelcea and S. M. Nowick, "A Low-Latency FIFO for Mixed-Clock Systems," in Proc. of the IEEE Computer Society Annual Workshop on VLSI (WVLSI'00), pp. 119-126, April 2000.

[20] J. Seng, E. Tune and D. Tullsen, "Reducing Power with Dynamic Critical Path Information," in Proc. Intl. Symposium on Microarchitecture (MICRO-34), pp. 114-123, Dec. 2001.

[21] R. Canal, J.M. Parcerisa, and A. Gonzalez, "A Cost-Effective Clustered Architecture," Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99), pp. 160-168, Oct. 1999

[22] L. S. Nielsen, C. Niessen, J. Sparso, and K. van Berkel, "Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage," in IEEE Transactions on Very Large Scale Integration Systems (TVLSI), December 1994.

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

[24] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in Proc. Intl. Symposium on Computer Architecture, pp. 83-94, June 2000.