Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors *

Anoop Iyer Diana Marculescu

Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh, PA 15213
Email: {aiyer, dianam}@ece.cmu.edu

Abstract

Due to shrinking technologies and increasing design sizes, it is becoming more difficult and expensive to distribute a global clock signal with low skew throughout a processor die. Asynchronous processor designs do not suffer from this problem since they do not have a global clock. However, a paradigm shift from synchronous to asynchronous is unlikely to happen in the processor industry in the near future. Hence the study of Globally Asynchronous Locally Synchronous (or GALS) systems is relevant. In this paper we use a cycle-accurate simulation environment to study the impact of asynchrony in a superscalar processor architecture. Our results show that as expected, going from a synchronous to a GALS design causes a drop in performance, but elimination of the global clock does not lead to drastic power reductions. From a power perspective, GALS designs are inherently less efficient when compared to synchronous architectures. However, the flexibility offered by the independently controllable local clocks enables the effective use of other energy conservation techniques like dynamic voltage scaling. Our results show that for a 5-clock domain GALS processor, the drop in performance ranges between 5-15%, while power consumption is reduced by 10% on the average. Fine-grained voltage scaling reduces the gap between fully synchronous and GALS implementations, allowing for better power efficiency.

1 Introduction

Most conventional microprocessor designs are synchronous in their construction; that is, they have a global clock signal which provides a common timing reference for the operation of all the circuitry on the chip. On the other hand, fully asynchronous designs built using self-timed circuits do not have any global timing reference; examples of this design style are given in Sutherland's work on *Micropipelines* [1]. **G**lobally **A**synchronous **L**ocally **S**ynchronous systems (which we refer to as GALS systems in this paper) are an intermediate style of design between these two. GALS systems contain several independent synchronous blocks which operate with their own local clocks and communicate asynchronously with each other. The main feature of these systems is the absence of a global timing reference and the use of several distinct local clocks (or clock domains), possibly running at different frequencies.

1.1 Motivation

The idea of GALS system design is in itself not new [2]. Interest in GALS design is now growing due to the following reasons:

- **Global clock distribution:** Trends of increasing die sizes and rising transistor counts may soon lead to a situation in which distributing a high-frequency global clock signal with low skew throughout a large die is prohibitively expensive in terms of design effort, die area, and power dissipation. GALS systems eliminate the need for careful design and fine-tuning of a global clock distribution network.
- **Design reuse:** Designers are now seriously exploring opportunities for reusing IP cores, and system-on-chip design is gaining popularity. Integrating several cores on one chip may not always be possible with a single clock system; different cores may have different clock requirements and operating frequencies. GALS systems with standardized asynchronous interfaces will facilitate design reuse.
- **Inertia:** While a fully asynchronous design style promises to solve both the above problems, a complete migration from synchronous to asynchronous systems is

*This work was supported in part by IBM Corp. SUR Grant No. 4901B10170 and by SRC Grant No. 2001-HJ-898.

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 IEEE

not likely to happen in the immediate future; CAD tools for asynchronous design are mature, but not commercially strong yet.

In the microprocessor industry, global clock distribution issues (further discussed in section 2) are perhaps the best motivating factor for the study of GALS systems. However since products in this arena are highly performance-driven, we need to evaluate the impact of asynchronous communication on performance and power. We describe in this paper the development of a modeling and simulation framework and the results of some experiments with a hypothetical superscalar GALS processor design. We have attempted to address the following issues:

- If we design a microprocessor in a GALS style with multiple clock domains, how much performance overhead will it incur over a fully synchronous processor?
- Will the elimination of the global clock network help in reducing power in a microprocessor, as other works have claimed?
- How can we exploit the extra flexibility offered by independent clock domains in a GALS processor?

In this work, we show that GALS processors are *not* necessarily more power efficient than fully synchronous designs, as it has been previously claimed, but they *may* become so if clock speed and supply voltage are tuned for each synchronous block. Eventually, fine adaptation can be extended to support application-driven, multiple-domain dynamic clock/voltage scaling.

1.2 Related Work

Sutherland's paper on *Micropipelines* [1] contains a good introduction to asynchronous design. Asynchronous processor cores have been in development for over a decade now; for example, the Amulet processor core developed at Manchester, which implements the ARM instruction set, is in its third generation and is commercially viable and competitive [3]. GALS systems were studied in detail by Chapiro in his 1984 PhD thesis [2]. His work covers metastability issues in GALS systems and outlines a stretchable clocking strategy which provides a mechanism for asynchronous communication. Chelcea and Nowick propose in [4, 5] the use of FIFOs as a low-latency asynchronous communication mechanism between synchronous blocks. Hemani *et al.* estimated in [6] the clock power savings in GALS designs compared to synchronous designs. However, their work targets a regular ASIC design flow with simpler clocking strategies rather than the aggressive clock distribution networks used in microprocessors. Muttersbach *et al.* have implemented asynchronous wrappers around synchronous blocks [7]; they have used these wrappers along with asynchronous memory blocks to implement an ASIC and have thus proved the feasibility of GALS design in silicon. However they have not provided any direct performance comparisons between GALS systems and synchronous systems. A similar system has been proposed by Moore *et al.* in [8]; pausible clocking for GALS systems has been described by Yun and Dooply in [9]. The work of Semeraro *et al.* [10] is the closest to our GALS study. They show the effect of voltage scaling by using off-line profiling of the application.

1.3 Organization of this Paper

The rest of this paper is organized as follows:

- In section 2 we discuss global clock distribution methods and the challenges it poses, and thus motivate the study of GALS systems.
- In section 3 we describe some of the issues involved in GALS processor design.
- In section 4 we outline an architecture for a hypothetical GALS processor and describe the simulation and modeling setup which we used to study power and performance trends in this processor.
- In section 5 we show some results on power and performance trends.
- Finally in section 6 we summarize our contributions and conclude with some future directions for research on GALS processors.

2 Clock Distribution

2.1 Design Practices

Generating a high frequency clock signal and distributing it across a large die with low skew is a challenging task demanding a lot of design effort, die area and power. Restle *et al.* [11] and Bailey and Benschneider [12] give a good overview of clocking system design for high-performance processors.

In most processors, a phase lock loop (PLL) generates a high frequency clock signal from a slower external clock. A combination of a metal grid and a tree of buffers is used to distribute the clock throughout the chip. Trees have low latency, dissipate less power and use less wiring; but they need to be rerouted whenever the logic is modified even slightly, and in a custom-designed processor, this requires a lot of effort. Trees work well if the clock loading is uniform across the chip area; unfortunately, most microprocessors have widely varying clock loads. Metal grids provide a regular structure to facilitate the early design and characterization of the clock

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 IEEE

network. They also minimize local skew by providing more direct interconnections between clock pins.

Moreover, clocking in most processors today is hierarchical. Figure 1 shows an example of a hierarchical distribution network; several major clocks are derived from a global clock grid, and local clocks are in turn derived from the major clocks. This approach serves to modularize the overall design and to minimize the local skew inside a block. It also has the advantage that clock drivers for each functional block can be customized to the skew and drive requirements of that block; thus the drive on the global clock grid need not be designed for the worst-case clock loading.

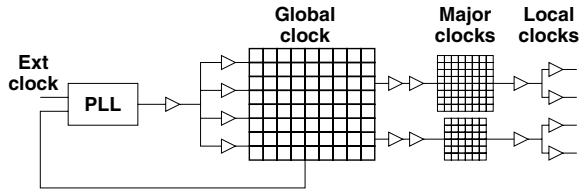


Figure 1. An example of a hierarchical clock distribution network

2.2 Case Study

Restle *et al.* have argued in [11] that clock skew arises mainly due to process variations in the tree of buffers driving the clock. Since device geometries will continue to shrink and clock frequencies and die sizes will continue to increase, global clock skew induced by such process variations can only get worse. Hence we argue that we will reach a point where skew will thus eat up a significant proportion of the cycle time and thus will directly affect performance.

This point may already have been reached. Table 1 shows a case study of a few processor designs spanning four major CMOS technology generations which entered the market during the last decade. The numbers in the table clearly show that technology scaling has led to a dramatic increase in design size and speed. However, since interconnects do not scale as well as transistor gate lengths do, these numbers indicate that the complexity of the clock distribution task has increased even more dramatically; we now have to clock many more registers with much smaller skew budgets than before.

Designers have handled this increased design complexity using complicated hierarchical distribution systems like the one shown in Figure 1. However, even a complex system of multiple grids and H-trees is not sufficient for today's Gigahertz clocks. For instance, the 800-MHz prototype of the Itanium chip has a projected skew of 110 ps using a hierarchical distribution scheme with multiple grids and trees. This skew is almost 10% of the total cycle time. The Itanium designers have added a network of 32 active deskewing circuits [13] which connect multiple local clock grids together and help in bringing down the overall skew to 28 ps.

While techniques like active deskewing help to push the envelope for clocked systems further, they come at a significant cost in terms of die area and power dissipation. At some point, pushing the limits of clock distribution networks will lead to diminishing marginal returns. At that stage, GALS design techniques will come in useful.

3 Globally Asynchronous Locally Synchronous Processor Design

In this section we discuss some architectural issues involved in the design of a globally asynchronous locally synchronous processor, with focus on performance and power evaluation. Since our primary focus is at the architecture level, we choose to omit several lower-level issues in our study. Some areas which have been dealt with in detail elsewhere are:

- **Metastability resolution:** The problem of metastable signals and techniques for metastability resolution using synchronizers and arbiters are discussed in [14]. Our approach uses asynchronous FIFOs [4, 5] between clock domains and this in turn relies on synchronizers.
- **Local clock generation:** Each clock domain in a GALS system needs its own local clock generator; ring oscillators have been proposed as a viable clock generation scheme [2, 7]. We assume that we can use ring oscillators in each synchronous block in the GALS processor.
- **Failure modeling:** A system with multiple clock domains is prone to synchronization failures; we do not attempt to model these since their probabilities are miniscule (but non-zero) [14] and our work does not target mission-critical systems.

3.1 Defining Synchronous Blocks

Hemani *et al.* have described an automated strategy for defining locally synchronous blocks in a GALS design [6]. Starting from a hierarchical RTL description of the system, their method uses iterative refinement to get an optimal partitioning of the system into a number of synchronous blocks, using clock power as an objective function for optimization. In a custom-designed system like a microprocessor, performance requirements justify manual intervention in the partitioning phase. Since the primary motivation behind GALS design is to avoid distributing a common clock signal over large areas, the strategy for partitioning the design into synchronous blocks will largely be dictated by physical design aspects. However, since asynchrony can lead to higher latencies, it is crucial to take architecture issues into account when partitioning the design.

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 IEEE

Design	Technology	Device count	Cycle time	Skew	Remarks
Alpha 21064	0.8 μm (1992)	1.6M	5 ns	200 ps	Single line of drivers for clock grid
Alpha 21164	0.5 μm (1995)	9.3M	3.3 ns	80 ps	Two lines of drivers for clock grid
Alpha 21264	0.35 μm (1998)	15.2M	1.7 ns	65 ps	16 distributed lines of drivers
Itanium (with active deskewing)	0.18 μm (2001)	25.4M	1.25 ns	28 ps	32 active deskewing circuits
Itanium (without active deskewing)	0.18 μm (2001)	25.4M	1.25 ns	110 ps	Projected skew without deskewing

Table 1. Trends in global clock skew for microprocessor designs across process generations

In the traditional superscalar out-of-order processor model the *instruction flow* consists of fetching instructions from the instruction cache, using the branch predictor for successive fetch addresses. The *register dataflow* consists of issuing instructions out of the instruction window and forwarding results to dependent instructions. The *memory dataflow* consists of issuing loads to the data cache and forwarding data to dependent instructions. Introducing high latencies in any of these three crucial flows will have an impact on the processor's performance.

The level 1 instruction cache and the branch predictor taken together are a good candidate for one synchronous block corresponding to the front-end of the pipeline. In some architectures, notably in CISC architectures like Intel's IA-32, the decode logic occupies a large area and consists of several pipe stages; in such cases, decode would be a good candidate for another synchronous block.

Inside the out-of-order execution core, it is difficult to make generalizations and say which parts of the core may be decoupled without much overhead and which may not; such decisions are very specific to the microarchitecture and the instruction set of the processor. Area and clock distribution considerations obviously suggest this partitioning to some extent. For instance in the 21264 Alpha the 'major clocks' (tapped from the global clock and distributed locally) are defined this way, based mostly on the top-level hierarchy of the design; they suggest a partitioning system for that specific implementation. The 21264 has the following major clocks [12]: (1) instruction fetch and branch predict (2) bus interface unit (3) integer issue and execution units (4) floating point issue and execution units (5) load/store unit (6) pad ring. We shall revisit this implementation in section 4 where we describe our proposed GALS architecture.

3.2 Asynchronous Communication Mechanisms

Many methods have been proposed for clocking GALS systems with *stretchable clocks* [2, 7, 8]. Such clocking systems manage asynchronous communication between two clock domains by stretching one phase of both the clocks while the handshaking and data transfer takes place. This is typically done using an arbiter element inside the loop of a ring oscillator. While this mechanism provides an elegant and fail-safe method of communication, it also stalls both the synchronous blocks during the transaction. In a processor pipeline, transactions occur practically during every cycle. Stretching the clock every cycle would lead to a situation where the effective clock frequency is determined not by the clock generator but by the rate of communication with other synchronous modules.[1] This is not desirable, especially in systems where the frequencies of the different clocks have been chosen to meet performance and power requirements.

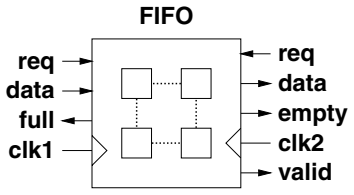


Figure 2. Asynchronous FIFO for interfacing two clock domains

Chelcea and Nowick have presented in [4, 5] a design for a low-latency token-ring based FIFO which can be used for asynchronous communication between synchronous blocks. The interfaces to the FIFO are shown in Figure 2. Their design uses *full* and *empty* signals to indicate the occupancy of the FIFO. The *empty* signal is controlled by the producer of data into the FIFO and is synchronized to the consumer's clock; similarly, the *full* signal is controlled by the consumer and is synchronized to the producer's clock. A few modifications are made to the circuit to account for latencies in synchronization and to prevent deadlock. In addition to providing high throughput in the steady state, the design has low latency when compared to other methods we tested. Since the focus of our work is at a higher level of abstraction, we shall not go into further details; a complete description of the operation of the circuit is given in [4, 5]. We shall refer back to this FIFO structure when describing our experiments with GALS design.

3.3 Multiple Supply Voltages

An interesting possibility with the use of multiple local clocks with potentially different speeds is the use of multiple

[1] To an extent, this behavior is rather like the timing behavior of Sutherland's *Micropipelines*, where the rate of forward communication in the pipeline makes the system *self-timed*.

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

local supply voltages in a dynamic or application-dependent manner. Since applications vary in their usage of processor resources, intelligent selection of clock frequencies can give us significant power savings with minimal impact on performance. The simplest example of this is slowing down or shutting off the floating-point units while running integer applications. Selectively slowing down certain regions of the processor is more easily achieved in a GALS design than in a synchronous design because different subsystems run on different clocks and these clocks can be independently controlled.

If some parts of the core are slowed down, they can be operated at a lower supply voltage too. In such a system, the asynchronous communication interfaces between synchronous blocks will need to have level-conversion circuits. The amount by which we can reduce the voltage depends on the slowdown of the clock. Since energy consumption is dependent on the *square* of the supply voltage, reducing the supply voltage will lead to significant energy benefits.

The relationship between logic delay D and supply voltage V_{dd} is given by the following equation [15]:

$$D \propto \frac{V_{dd}}{(V_{dd} - V_t)^{\alpha}} \quad (1)$$

where V_t is the threshold voltage of the transistor and α is a technology-dependent factor. For a 0.35 μm technology, α is 2; for smaller technologies, the value of α is between 1 and 2. This implies that savings arising out of dynamic voltage scaling for a given delay value are higher for smaller technology generations.

4 A GALS Architecture

We have studied a superscalar processor model and have attempted to build a GALS model which duplicates its pipeline structure for the most part, so that we can compare GALS processors with synchronous processors in terms of power and performance. The architecture that we chose for our study is a hypothetical processor resembling the 21264 Alpha in some ways.

4.1 The Architecture

After a detailed look at the architecture, we chose to have five clock domains in the GALS version of the design. Figure 3 shows the pipeline structure of both the synchronous (base) processor and the GALS processor we designed. The boundaries between clock domains in the GALS processor are indicated by dotted lines. In the base (synchronous) model, all the logic runs off the same clock. In the GALS model, various regions are clocked using different clock signals independent of each other. The first stage of the pipeline consists of an instruction cache and branch prediction unit (clock domain 1). The next stages are instruction decode and register rename (clock domain 2). There are three issue queues in the design: one for integer instructions (clock domain 3), one for floating-point instructions (clock domain 4) and one for loads and stores (clock domain 5). In the GALS processor, the integer ALUs and the integer issue queue are in the same clocking region. This ensures that dependent instructions within the integer issue queue can be issued back-to-back as soon as operands are available. Similarly, floating-point ALUs and the floating-point issue queue share one clock, and the data-cache, the level-2 cache and memory issue queue share one clock.

In the synchronous version, communication between successive logic blocks is done using regular pipe stages. In the present version of the GALS model, asynchronous FIFOs described in section 3.2 have been used.

Table 2 gives a summary of the pipeline stages in the processor models we developed for our experiments, along with a listing of the clock domains of the GALS processor which are involved in each pipe stage. Table 3 describes the microarchitecture in some detail.

Stage	Operation	Domains
1	Fetch from I-cache	1
2	Decode	2
3	Register rename, Regfile read	2
4	Dispatch into issue queue	2, 3/4/5
5	Issue to functional unit	3/4/5
6	Execute	3/4/5
7	Wakeup, Writeback	3/4/5
8	Regfile write, Commit	3/4/5, 2

Table 2. Pipeline stages in our processor models

Fetch and decode rate	4 inst/cycle
Integer issue queue size	20
FP issue queue size	16
Memory issue queue size	16
Integer registers	72
FP registers	72
L1 data cache	16KB 4-way 1 cycle latency
L1 instruction cache	16KB direct-mapped 1 cycle latency
L2 unified cache	256KB 4-way 6 cycles latency
ALUs	4 integer, 4 FP

Table 3. Microarchitecture details of our processor models

4.2 A GALS Simulation Framework

Building a cycle-accurate simulator for a single-clock pipelined system is simple; in C, we only need to call various pipe-stage functions in the reverse order of their occurrence in the pipeline. However, to simulate a multiple-clock

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

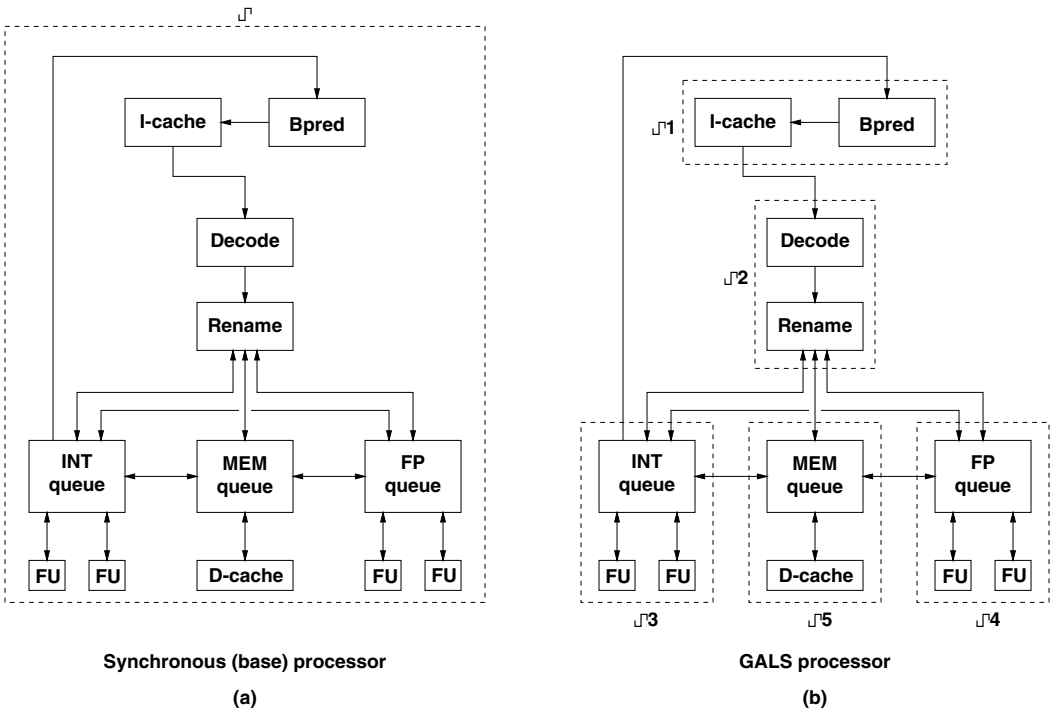


Figure 3. Pipeline of the simulated architecture

system where the different clocks have entirely independent frequency and phase, we need a more detailed simulation infrastructure.

We have written a general-purpose event-driven simulation engine which can be used to simulate any asynchronous system, synchronous (clocked) system, or a system which contains both asynchronous and synchronous components. The guts of this event-driven simulation engine consist of an event queue and a global timer. The event queue is implemented as a singly linked list in C. Each node of the queue contains the following fields:

- a function to call at each occurrence of the event;
- a parameter to call the function with;
- a time at which the event is scheduled to occur;
- a priority number to determine the order of execution of events which are scheduled occur at the same time instant;
- for periodic events, a time period of repetition (for simulation of clocked systems), and
- a pointer to the next queue item.

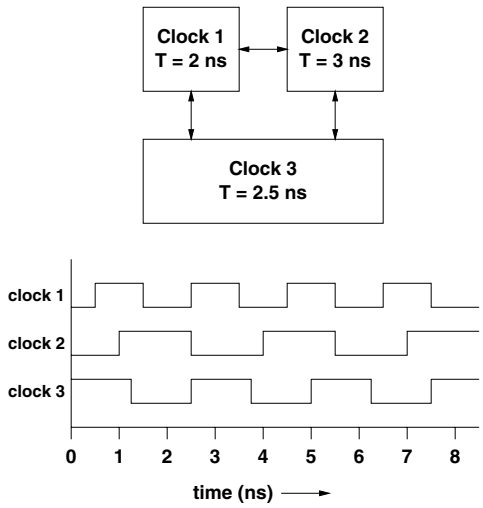
To set the system in motion, we need to insert one or more starting events into the event queue. The queue contains events sorted in increasing order of their scheduled times. Hence, processing the event queue for running the simulation is easy; we only need to read successive events from the head of the queue and execute them by calling the appropriate execution functions. To simulate clocked systems, we need to insert one event for each clock domain; for each such event, we need to specify a time period. When the execution engine processes such a periodic event, it schedules another instance of the same event into the queue, thus representing the next cycle of execution of the clocked system.

Figure 4 (a) shows an example of a system with three clock domains, each of which has a different clock frequency. To simulate this system, we need to add three starting events into the event queue, all of which are periodic, to represent the three clock domains. Figure 4 (b) shows the C code which models the system.

4.3 Performance and Power Models

To evaluate the above architecture, we wrote models of both the synchronous and the GALS processors using the Simplescalar toolset [16]. Simplescalar provides a comprehensive infrastructure for modeling and simulation of microarchitecture features. To simulate the GALS processor, we made use of the event-driven simulation engine described earlier in section 4.2. We have set up five clock domains in our simulator and in the first set of experiments, had all the clocks running at the same speed. The starting phase of each clock was set to a random value at runtime.

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**



(a)

```
init_event_queue ();
add_event (/* start time */ 0.5,
           /* function */ &clock1_logic,
           /* param */ NULL,
           /* period */ 2.0);
add_event (/* start time */ 1.0,
           /* function */ &clock2_logic,
           /* param */ NULL,
           /* period */ 3.0);
add_event (/* start time */ 0.0,
           /* function */ &clock3_logic,
           /* param */ NULL,
           /* period */ 2.5);
process_event_queue ();
```

(b)

Figure 4. Event-driven GALS system simulation. (a) An example system. (b) C code for simulating this system.

We used the Wattch framework [17] to add power models to our processor simulation. Wattch provides switching capacitance modeling for structures like ALUs, caches, arrays and buses in a processor. These are integrated into our base and GALS simulators to provide energy statistics. To account for overheads arising from clock-gating and leakage currents, we modeled unused modules as consuming 10% of their full power. We also modeled power consumed by the FIFOs used for communication between domains.

In addition to modeling the switching capacitance of memories and buses inside the processor, we have also modeled the switching capacitance of clock grids. For the synchronous base processor model, we assumed a clock distribution hierarchy resembling that of the 21264 Alpha processor. We modeled one global clock grid and five local clock grids corresponding to the five clock domains discussed in section 3.1. The areas and metal densities of each clock grid were approximated by the numbers published for the 21264 processor. For the GALS processor, since there is no global clock, we eliminated the switching capacitance of the global clock grid and retained the five major clock grids, corresponding to the distribution networks for each of the synchronous blocks.

5 Experimental Results

To assess the performance and power of our proposed GALS processor design, we tested the base and the GALS simulators with a set of benchmarks taken from the Spec95 [18] and the Mediabench [19] benchmark suites. We have performed two sets of experiments:

1. Base versus GALS performance and power analysis with all synchronous blocks running at the same clock frequency and supply voltage.
2. Base versus a multiple-clock, multiple-voltage GALS design.

5.1 Power and Performance Analysis

Performance

Not surprisingly, the GALS processor is slowed down by asynchronous communication and does not perform as well as the synchronous processor. Figure 5 shows the relative slowdown of various benchmarks running on the GALS processor when compared to the synchronous processor. On an average, the benchmarks we ran on GALS were slower by 10% when compared to base. As expected, the *fpppp* benchmark had the lowest performance hit. This is due to the application's exceptionally small proportion of branch instructions; on an average only one in every 67 instructions is a branch in this benchmark, while most other applications have one branch for every five to six instructions. This indicates that the asynchronous FIFO models used in our design have good throughput in the steady state when there are no branch mispredictions. This also suggests that branch mispredictions will prove more expensive in the GALS model due to its longer recovery pipeline.

We have also observed that the performance of the GALS processor varies with the relative phase of the various clocks, especially in the case where all the clocks are of the same frequency. This variation is of the order of 0.5%.

Instruction Latencies

On close examination of other statistics in the processor pipeline, we can see that the introduction of asynchronous

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

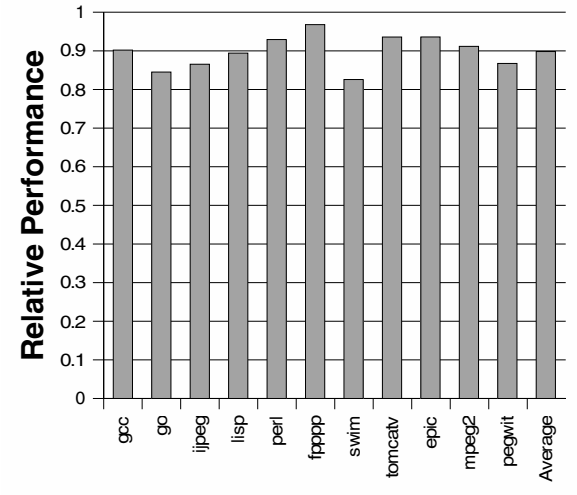


Figure 5. Performance of the GALS model relative to the base model

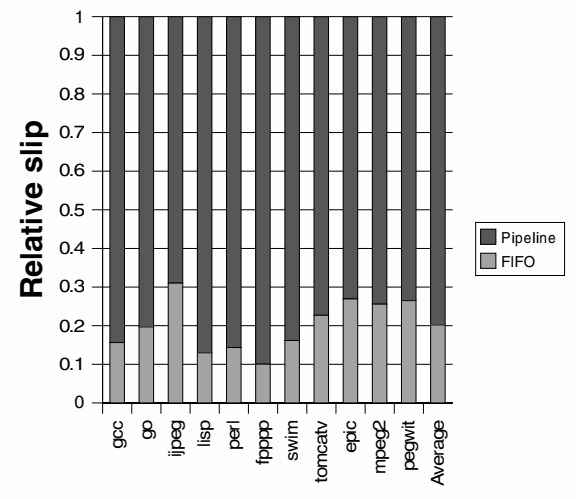


Figure 7. Relative Slip

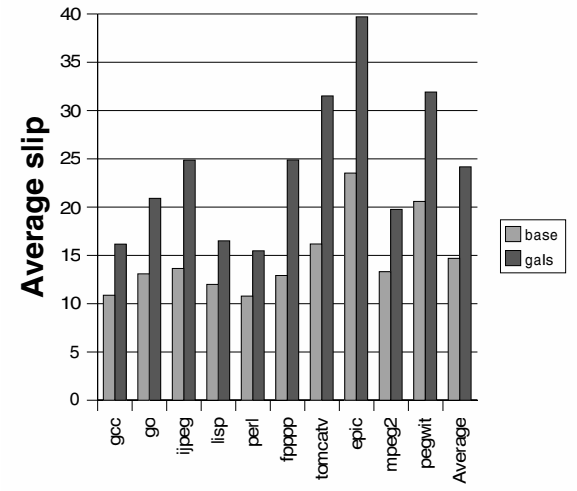


Figure 6. Average slip of an instruction in the base and GALS designs

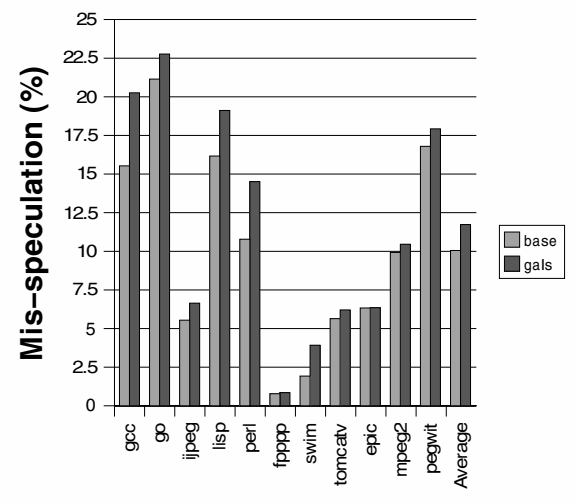


Figure 8. Percentage of mis-speculated instructions in the base and GALS processors

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

communication latencies inside the design has led to various other overheads which in some cases offset the power gains due to the absence of global clock. For instance, the *slip* (the average time taken by each instruction from the fetch to the commit stage) increases by 65% on average for all benchmarks in the GALS processor, as seen in Figure 6. This is because the addition of asynchronous communication channels leads to an increase in the effective length of the pipeline. Figure 7 shows the proportion of this slip time which is spent in the FIFOs (marked "FIFO" in the graph) versus the proportion of time spent in execution units, issue queues, etc. (marked "pipeline" in the graph). As we expect, the difference in slip between the GALS and the base versions is due in part to the time spent in the FIFOs. However, there is still an increase in the slip which cannot be accounted for by the time spent in FIFOs alone; this is caused by the latency in forwarding results from one queue to another through FIFOs. Note that this delay is caused by the FIFO latency of forwarding results and not by the latency in the instruction flow.

Speculation

This increase in pipeline length in the GALS processor also leads to higher speculative execution, as shown in Figure 8. This is most marked for the integer applications we tested, where the percentage of mis-speculated instructions goes up from 13.8 percent in the base processor to 16.7 percent in the GALS processor. Increase in speculation is less for applications containing many long-latency instructions. Similarly, we have observed that the average number of in-flight instructions in the pipeline is higher in the GALS model; so is the average occupancy of the register allocation tables and issue queues. For instance the integer register allocation table occupancy went up from 15 in base to 24 in GALS for the *ijpeg* benchmark.

Power

Figure 9 shows the relative total energy and average power consumption of the GALS processor, normalized to the respective measures of the base processor. In most benchmarks, the elimination of the global clock has resulted in some savings in the per-cycle power dissipation. But due to the extra switching activity inside the core, higher occupancies of the issue queues and register allocation tables, increased speculation and higher execution times, the total energy needed for execution is not necessarily lower, but is higher for the GALS processor in some cases. For the benchmarks we tested, this increase in energy is 1% on average.

Figure 10 shows the breakdown of the base and GALS model power consumption into various macro blocks. From the figure, we can see that power gains arising from elimination of the global clock are offset by the increased power consumption of other blocks.

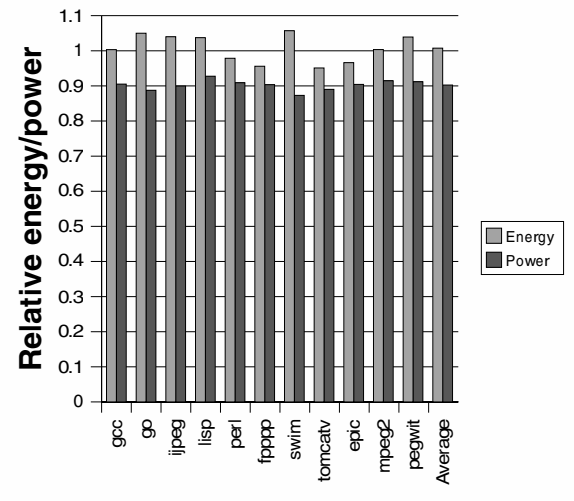


Figure 9. Energy and power consumption of the GALS processor normalized to those of the base processor

5.2 Multiple-Clock, Multiple-Voltage Processors

In a second set of experiments, we tried to determine which parts of the processor could be slowed down in an application-dependent manner without affecting performance. The technique of multiple supply voltages described in section 3.3 was used to determine an optimal supply voltage for lowest operating power, using equation 1 with a value of $\alpha = 1.6$ which is appropriate for today's 0.13 μm devices. The voltage thus determined is of course the ideal case; in practice, there will be an overhead due to DC-DC level conversion circuits.

Figure 11 shows the results of slowing down some clock domains in a generic fashion; the fetch clock and memory clock were slowed down by 10% and the floating point clock was slowed by 50%. The energy and power benefits are decent but performance losses are substantial (about 18%). From this graph, we see that we can apply clock slowdown only on a selective basis, after studying the application's characteristics.

- **perl:** Since there are virtually no floating-point instructions in this integer benchmark, we slowed down the FP clock by a factor of 3. The performance drop was 9% over the base version; the total energy was reduced by 10.8% and the average power by 18%.
- **ijpeg:** In this case, we have considered simultaneous slowing down the fetch, floating point and memory clocks (domains 1, 4 and 5 in Figure 3 (b)). We chose to study the impact of slowing down the memory clock on the power and performance of *ijpeg* since this benchmark has a very low proportion of memory accesses. In

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

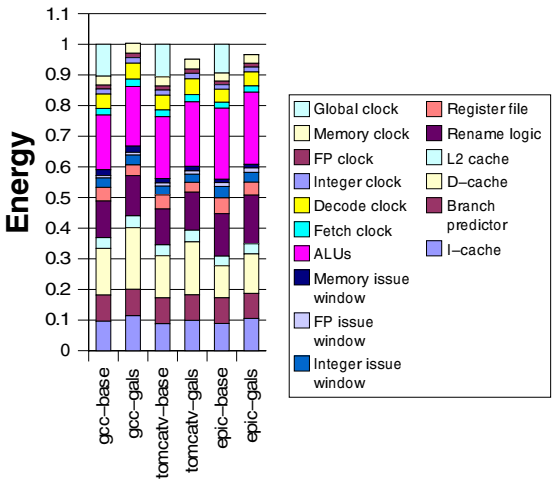


Figure 10. Breakdown of energy into various macro blocks

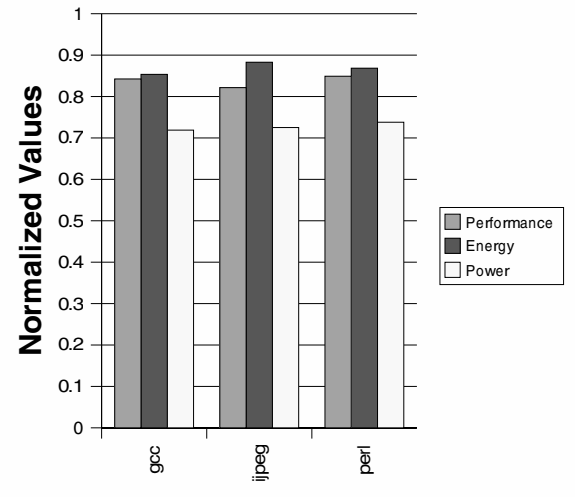


Figure 11. Results from selective slowdown applied on three benchmarks

all cases reported in Figure 12, the fetch clock has been slowed down by 10% and the FP clock by 20%, while for the memory clock we have considered four cases: no slowdown (gals-00), slowdown of 10% (gals-10), 20% (gals-20) and 50% (gals-50). Figure 12 shows that we can trade off performance for energy savings for this benchmark. Energy savings vary between 4 and 13% with a performance drop between 15 and 25% when compared to the fully synchronous processor.

- **gcc:** We chose this integer benchmark to apply a slower clock to the floating-point queue and units. Since the instruction bandwidth of this benchmark is also low, we slowed down the fetch unit by 10%. Figure 13 shows the results for performance, power and energy, normalized to the base case. The numbers marked "gals-1" are from the case where the floating-point clock is slower by 50% and the numbers marked "gals-2" are from the case where it slower by a factor of 3. The graph shows that *gcc* can afford to have a slower floating point unit without too much performance hit. Given scaleable voltage supplies, this technique also provides energy savings of 11% and power savings of 21% with a performance loss of 13% when compared to the fully synchronous processor.

To compare the capability of the GALS processor to trade off power for performance, we have also provided the normalized energy of the base (synchronous) processor when run at a slower clock (and lower voltage) that would exhibit an equivalent performance penalty (the column labeled "ideal" in Figures 12 and 13). It can be seen that by slowing down the floating-point clock domain, the GALS processor is able to trade off performance for energy in case of the *gcc* benchmark. Figure 12 shows that slowing down the memory clock does not lead to a good performance-energy tradeoff for the *ijpeg* benchmark. Hence the extent of the tradeoff we can achieve by slowing down various clock domains is dictated by the nature of the application.

Overall, our experimental evidence shows that naive GALS implementations (with all clocks running at the same frequency) may not necessarily be very energy efficient as claimed previously. Instead, the increased flexibility of running local clocks at different speeds (and thus different voltages) offers a viable solution for energy aware computing under the increasing pressure of handling clock skew and distribution issues.

6 Conclusion

Our modeling and simulation setup has given direct comparisons of power and performance of GALS systems against those of synchronous systems. Our experimental evidence shows that the overhead associated with GALS processors renders them inefficient; hence eliminating the global clock is not in itself a solution for low power. However, combined with intelligent fine-tuning of clock frequency and supply voltage, GALS systems can provide some power benefits. Clocking smaller areas will mean smaller skew values and hence faster clocks; we have not modeled such effects in this work because skew estimates require extensive physical design. Besides, having independent clock domains eliminates the need for balanced pipelines and could provide more avenues for fine-tuning performance.

Since clock distribution issues may necessitate the prac-

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

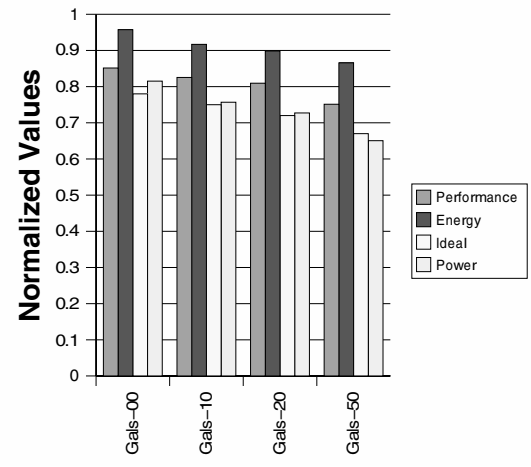


Figure 12. Impact of selective fetch, memory, and FP clock slowdown (ijpeg benchmark)

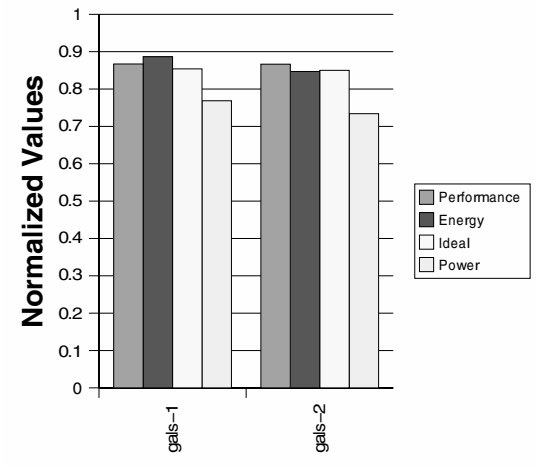


Figure 13. Impact of selective fetch and FP clock slowdown (gcc benchmark)

tice of GALS design in the future, studies on performance enhancement in GALS systems are worthwhile. Further studies in this direction could involve latency-hiding techniques like multithreaded execution in hardware.

References

[1] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, June 1989.

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

[3] S. B. Furber, D. A. Edwards, and J. D. Garside, "AMULET3: A 100 MIPS Asynchronous Embedded Processor," in *Proc. Intl. Conference on Computer Design (ICCD)*, 2000.

[4] T. Chelcea and S. M. Nowick, "A Low-Latency FIFO for Mixed-Clock Systems," in *Proc. IEEE Computer Society Workshop on VLSI*, 2000.

[5] T. Chelcea and S. M. Nowick, "Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols," in *Proc. Design Automation Conference (DAC)*, 2001.

[6] 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)*, 1999.

[7] J. Muttersbach, T. Villiger, and W. Fitchner, "Practical Design of Globally Asynchronous Locally Synchronous Systems," in *Proc. Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2000.

[8] S. W. Moore, G. S. Taylor, P. A. Cunningham, R. D. Mullins, and P. Robinson, "Self Calibrating Clocks for Globally Asynchronous Locally Synchronous Circuits," in *Proc. Intl. Conference on Computer Design (ICCD)*, 2000.

[9] K. Y. Yun and A. E. Dooply, "Pausible Clocking-Based Heterogeneous Systems," *IEEE Transactions on VLSI Systems*, December 1999.

[10] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarakadas, and M. L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," in *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2002.

[11] P. J. Restle *et al.*, "A Clock Distribution Network for Microprocessors," *IEEE Journal of Solid State Circuits (JSSC)*, May 2001.

[12] D. W. Bailey and B. J. Benschneider, "Clocking Design and Analysis for a 600-MHz Alpha Microprocessor," *IEEE Journal of Solid State Circuits (JSSC)*, Nov 1998.

[13] S. Tam, S. Rusu, U. N. Desai, R. Kim, J. Zhang, and I. Young, "Clock Generation and Distribution for the First IA-64 Microprocessor," *IEEE Journal of Solid State Circuits (JSSC)*, November 2000.

[14] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 1996.

[15] K. Chen and C. Hu, "Performance and Vdd Scaling in Deep Submicrometer CMOS," *IEEE Journal of Solid State Circuits (JSSC)*, October 1998.

[16] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, version 2.0," Tech. Rep. 1342, University of Wisconsin-Madison, CS Department, June 1997.

[17] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-level Power Analysis and Optimizations," in *Proc. Intl Symp on Computer Architecture (ISCA)*, 2000.

[18] "Spec95 Benchmarks." http://www.spec.org.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *International Symposium on Microarchitecture (MICRO)*, 1997.

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**