10th Intl. Symp. on Asynchronous Circuits and Systems (ASYNC),
Herakleion, Crete, Apr. 2004

Hiding Synchronization Delays in a GALS Processor Microarchitecture*

Greg Semeraro⋆, David H. Albonesi‡,
Grigorios Magklis†, Michael L. Scott†, Steven G. Dropsho† and Sandhya Dwarkadas†

⋆Department of Computer Engineering
Rochester Institute of Technology
Rochester, NY 14623

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

Abstract

We analyze an Alpha 21264-like Globally–Asynchronous, Locally–Synchronous (GALS) processor organized as a Multiple Clock Domain (MCD) microarchitecture and identify the architectural features of the processor that influence the limited performance degradation measured. We show that the out-of-order superscalar execution features of a processor, which allow traditional instruction execution latency to be hidden, are the same features that reduce the performance degradation impact of the synchronization costs of an MCD processor. In the case of our Alpha 21264-like processor, up to 94% of the MCD synchronization delays are hidden and do not impact overall performance. In addition, we show that by adding out-of-order superscalar execution capabilities to a simpler microarchitecture, such as an Intel StrongARM-like processor, as much as 62% of the performance degradation caused by synchronization delays can be eliminated.

1 Introduction

Globally Asynchronous, Locally Synchronous (GALS) designs are an intermediate approach between fully asynchronous and fully synchronous clocking styles. A GALS design has the advantage in that it eliminates the timing and cost overhead of global clock distribution while maintaining a synchronous design style within each clock domain. One such GALS processor approach, which we call MCD (Multiple Clock Domain), provides the capability of independently configuring each domain to execute at frequency/voltage settings at or below the maximum values [20]. This allows domains that are not executing operations critical to performance to be configured at a lower frequency, and consequently, an MCD processor has the advantage that energy can be saved [14, 19]. However, an MCD processor has the disadvantage that inter-domain communication may incur a synchronization penalty resulting in performance degradation.

In this paper, we analyze the MCD microarchitecture and describe how the processor architecture influences the performance degradation due to synchronization. We describe the synchronization points within MCD and characterize the relative synchronization costs of each. We further demonstrate how the latency tolerating features of a dynamic superscalar processor can hide a significant fraction of the synchronization delay, leading to a very modest performance overhead. Combined with our previous results [14, 19], which show the energy saving advantages of our GALS MCD design, these results demonstrate the potential viability of our MCD processor.

*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 an IBM Faculty Partnership Award; and by equipment grants from IBM and Intel.

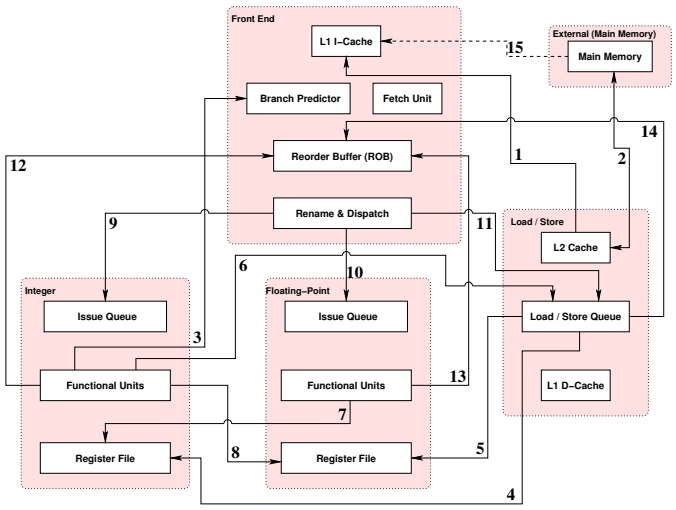


Figure 1. MCD block diagram, showing inter-domain communication.

2 MCD Microarchitecture

Our MCD architecture [20] uses four clock domains, highlighted in Figure 1, comprising the front end; integer processing core; floating-point processing core; and load/store unit. The main memory can also be considered a separate clock domain since it is independently clocked but it is not controllable by the processor. As a result of partitioning the processor in this way, there were no explicit changes to the pipeline organization of the machine. This limits the scope of changes required by the MCD microarchitecture and isolates those changes to the interfaces. The processing cores within the domains remain unchanged as fully synchronous units.

In choosing the boundaries between domains, we attempted to identify points where i) there already existed a queue struc-

Table 1. MCD-specific processor configuration parameters.

Parameter	Value(s)
Clock Domains	Fetch/Dispatch, Integer, Floating-Point and Load/Store
Domain Voltage	0.65V – 1.20V
Voltage Change Rate	66.9 $\frac{\text{nsec}}{\text{mV}}$
Domain Frequency	250MHz – 1.0GHz
Frequency Change Rate	49.1 $\frac{\text{nsec}}{\text{MHz}}$
Domain Clock Jitter	110ps variance, normally distributed about zero
Synchronization Window	30% of 1.0GHz clock (300ps)

ture that served to decouple different pipeline functions, or ii) there was relatively little inter-function communication. The baseline microarchitecture upon which our MCD processor is built is a four-way dynamic superscalar processor with speculative execution similar in organization to the Compaq Alpha 21264 [11, 12, 13].

The MCD microarchitecture is characterized by a number of parameters, which define the unique features of the architecture. The values chosen represent present and near-term state-of-the-art circuit capabilities. These MCD-specific parameters are summarized in Table 1 and described in detail in the following sections.

3 Domain Clocking Style

We investigated two models for dynamic voltage and frequency scaling: an *XScale* model and a *Transmeta* model, both of which are based on published information from their respective companies [8, 9]. For both of these models, we assume that the frequency change can be initiated immediately when transitioning to a lower frequency and voltage, while the desired voltage must be reached first before increasing frequency. For the Transmeta model, we assume a total of 32 separate voltage steps, at 28.6mV intervals, with a voltage adjustment time of 20μs per step. Frequency changes require the PLL to re-lock. Until it does, the domain remains idle. The PLL locking circuit is assumed to require a lock time that is normally distributed with a mean time of 15μs and a range of 10–20μs. For the XScale model, we assume that frequency changes occur as soon as the voltage changes and circuits operate through the change.

Although we investigated both models, it became clear early on in the investigation that the Transmeta model is not appropriate for an MCD microarchitecture. The reason for this is that the Transmeta model requires the PLL to re-lock after each frequency change, stopping the domain during that time. With the tight interaction between the domains, the suspension of one domain quickly cascades to other domains. The end result is that whenever *any* domain frequency is changed in the Transmeta model, *all* domains are stalled for nearly all of the time required to re-lock the PLL. As one can imagine, this has a profound impact on overall performance. For these reasons the Transmeta model was not investigated further and is not included in any of the analysis that follows.

3.1 Clock Design

In this work we assume a system clock of 1.0GHz, derived from an external 100MHz source using on-chip PLLs where all domains frequencies are 1.0GHz but all are independent. These PLLs produce jitter. Moreover, since the external clock is derived from a lower frequency crystal using an external PLL, the external clock will also have jitter, and these jitters are additive, *i.e.*, $Jitter_{Total} = Jitter_{Crystal} + Jitter_{PLL}^{External} + Jitter_{PLL}^{Internal}$. The $Jitter_{Crystal}$ can be expected to be extremely small and due entirely to thermal changes; we assume a value of zero. The $Jitter_{PLL}^{External}$ is governed by the quality and design of the PLL chip used. A survey of available ICs reveals that most devices are specified as 100ps jitter; we use this number in our study. The $Jitter_{PLL}^{Internal}$ is also governed by the quality and design of the PLL. We assume a circuit of the same caliber as the external PLL, which implies a worst case jitter of 10ps (comparable error on a 10$\times$ shorter cycle). Since clock jitter varies over time with a long-term average of zero, the $\mid Jitter_{Total} \mid \leq$ 110ps ($\leq$ 11% of a 1.0GHz clock period).

Starting with the clock design of the Alpha 21264 [1], we derive an MCD clock design by dividing the clock grid into regions corresponding to the MCD domain partitions. Each domain requires a separate PLL and clock grid driver circuit, all of which are fed by the 1.0GHz clock source. Although each of these domain PLLs derives its timing from a common source, we do not assume any phase relationship between the PLL outputs. We do this because we assume that the skew requirements for the 1.0GHz clock have been relaxed and we cannot guarantee that the clocks are in phase when they arrive at the domain PLLs.

4 Domain Interface Circuits

Fundamental to the operation of an MCD processor is the inter-domain communication. There are two types of communication that must be modeled: FIFO queue structures and issue queue structures. FIFO queues have the advantage that the inter-domain synchronization penalty can be hidden whenever the FIFO is neither full nor empty. The mechanism by which this is achieved and the precise definitions of `Full` and `Empty` are described in Section 4.1 and are similar to mixed clock FIFOs proposed by others [3, 5, 6, 7, 21]. The disadvantage of FIFO queue structures is that they can only be used where strict First-In-First-Out queue organization is applicable. Furthermore, if the communication is such that the FIFO is almost always empty (or full), the bulk of the synchronization penalty will not be hidden. The issue queue structure is very similar to the FIFO queue except that the inter-domain synchronization penalty must be assessed as each entry is put into the queue rather than only if the FIFO is empty or full. This is necessary since with an issue queue structure, the entries are consumed in an undefined order. Therefore, it is important to accurately model precisely when an entry becomes "visible" to the consumer.

4.1 FIFO Queue Structures

What follows is an analysis of the timing and characteristics of queues used to buffer control and data signals between different clock domains. Fundamental to this analysis are the following assumptions:

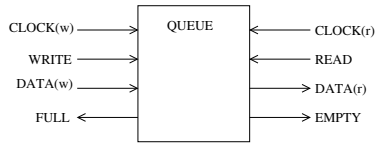


Figure 2. FIFO queue interfaces.

- Each interface is synchronous; *i.e.*, information is always read and written on the rising edge of the respective clock.
- Status signals which emanate from the interface are generated synchronously to the respective interface clock.
- Restrictions regarding the ratio of the maximum to minimum frequency are necessary but no other assumptions are made regarding the relative frequency or phase of the interface clocks.

The general queue structure that we use for inter-domain communication is shown in Figure 2 and is the same as in [5], with the following minor extensions. First, the queue design is such that the `Full` and `Empty` flags always reflect what the state of the queue will be *after* subsequent read/write cycles. In other words, the `Full` and `Empty` flags need to be generated prior to the queue actually being full or empty, using conditions that take into account the differences in the clock speed and potential skew of the two domains. The worst-case situation occurs when the producer is operating at the maximum frequency and the consumer at the minimum. There are several possible approaches to handling this problem. In this paper, we assume additional queue entries to absorb writes from the producer in order to recognize the potential delay of actually determining that the queue is full. In other words, the `Full` signal is generated using the condition that the queue length is within $\frac{\texttt{max\_freq}}{\texttt{min\_freq}} + 1$ of the maximum queue size. Note that our results do not account for the potential performance advantage of these additional entries. Second, the synchronization time of the clock arbitration circuit, T_S, represents the minimum time required between the source and destination clocks in order for the signal to be successfully latched *and seen* at the destination. It is this T_S which defines the conditions under which data which has been written into a queue is prevented from being read by the destination domain. Although the logical structure of our queues is similar to [5], we assume the arbitration and synchronization overhead described in [22] to define T_S, *i.e.*, we assume a T_S of 30% of the period of the highest frequency.

Even with completely independent clocks for each interface, the queue structure is able to operate at full speed for both reading and writing under certain conditions. A logic-level schematic of the FIFO queue structure is shown in Figure 3 for a 4-entry queue. Note that writing to and reading from the structure are independent operations with synchronization occurring in the generation of the `Full` and `Empty` status signals which are based on the `Valid` bits associated with each queue entry. Although the combinatorial logic associated with the generation of `Full` and `Empty` has been simplified in this figure for demonstration purposes (the logic shown does not take into account early status indication required for worst-case frequency differences), the synchronization circuits shown are as required in all cases. That is, the `Valid` bits for the queue entries must be synchronized to *both* the read and write clock do-

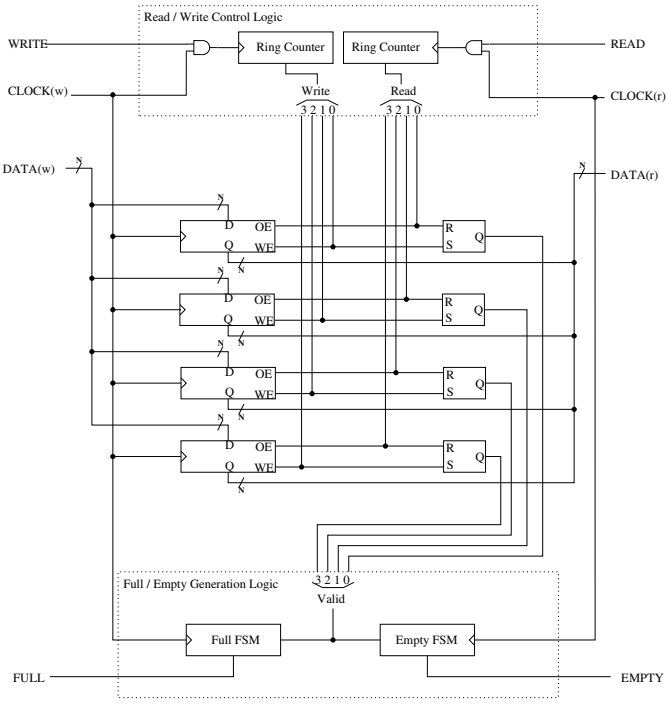


Figure 3. FIFO queue structure details.

mains independently. There is no chance of improper enqueuing/dequeuing as long as the synchronization circuit is glitch-free (see Section 4.3) and the access protocol described in the beginning of this section is applied.

Figure 4 shows a sample switch-level timing diagram of the FIFO queue of Figure 3. The critical points occur at the four labeled timings T_1, T_2, T_3, and T_4. T_1 occurs as the first entry is written into the queue. In this case, `Valid`$[0]$ is generated too closely to the falling edge of the read clock (cycle R_0) for it to be synchronized during that cycle (*i.e.*, $T_1 < T_S$); therefore, $\overline{\texttt{Empty}}$ is generated one cycle later. The next critical timing occurs as the fourth entry is written and the queue becomes full. In this case, `Valid`$[3]$ is generated on the rising edge of the write clock (cycle W_4) and synchronization is complete before the corresponding falling edge of the write clock (*i.e.*, $T_2 > T_S$). Note that in this case, the synchronization will always be complete before the falling edge because *both* the originating event (`Valid`$[3]$ generated on the rising edge of the write clock) and the synchronization event (`Full` status signal seen on the source domain) are in fact the same domain. The same situation exists for T_4 with respect to the destination domain. These conditions are guaranteed as long as the synchronization circuit requires less than half the period of the fastest interface to perform the synchronization. The queue remains full until a read occurs (cycle R_4) and `Valid`$[0]$ becomes false. The transition on `Valid`$[0]$ occurs well before the falling edge of the write clock (cycle W_7), which ensures that the synchronization is complete before that falling edge (*i.e.*, $T_3 > T_S$). The last critical timing occurs when the fourth entry is read and the queue becomes empty. Again, the transition of the valid signal (`Valid`$[3]$) occurs well before the falling edge of the read clock (cycle R_7); therefore, the synchronization is complete and no additional cycle of synchronization is required

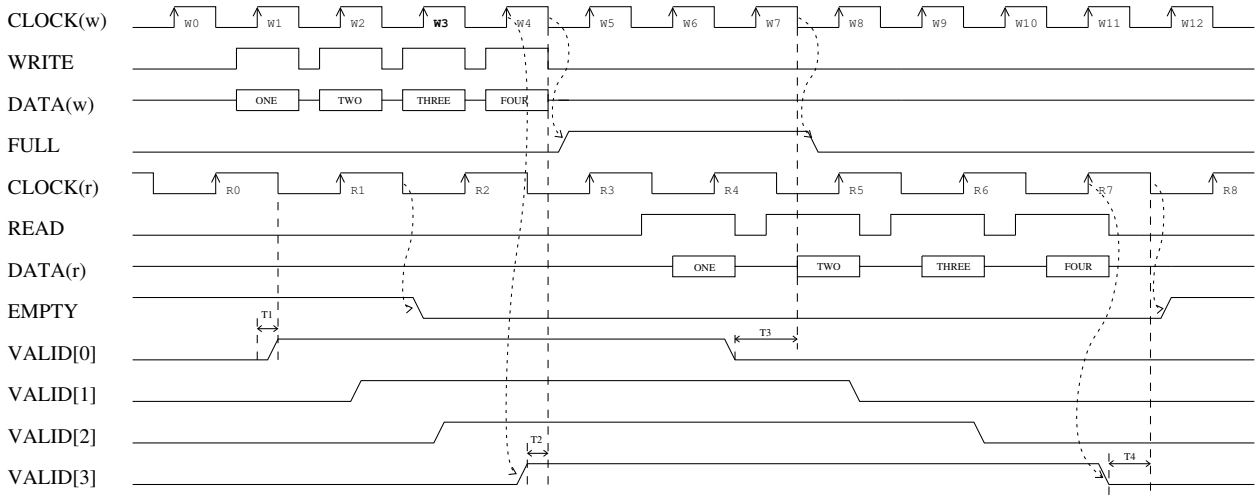


Figure 4. Queue timing diagram.

(*i.e.*, $T_4 > T_S$). This sample timing diagram illustrates how the synchronization parameter, T_S, manifests itself as read and/or write cycle penalties on respective clock domain interfaces.

4.2 Issue Queue Structures

Many of the queues that we use as synchronization points have a different interface than that described above. With an issue queue, each entry has `Valid` and `Ready` flags that the scheduler uses to determine if an entry should be read (issued). By design, the scheduler will never issue more than the number of valid and ready entries in the queue. Note, however, that due to synchronization, there may be a delay before the scheduler sees newly written queue data.

The issue queue structure design follows directly from the FIFO design but must be modified in an important way. The ability to hide the synchronization penalty when the FIFO is not full or not empty does not exist for issue queue type structures. The reason for this is that the information put into the queue is needed on the output of the queue as soon as possible (because the order in which entries are put into the queue does not dictate the order in which they may be removed).

4.3 Synchronization With Queues

The delay associated with crossing a clock domain interface is a function of the following:

- The synchronization time of the clock arbitration circuit, T_S, which represents the minimum time required between the source and destination clocks in order for the signal to be successfully latched at the destination. We assume the arbitration and synchronization circuits developed by Sjogren and Myers [22] that detect whether the source and destination clock edges are sufficiently far apart (at minimum, T_S) such that a source-generated signal can be successfully clocked at the destination.
- The ratio of the frequencies of the interface clocks.
- The relative phases of the interface clocks.

This delay can best be understood by examining a timing diagram of the two clocks (Figure 5). Without loss of generality, the following discussion assumes F_1 is the source and F_2 is the destination domain. Consider the case when the queue is initially empty. Data is written into the queue on the rising edge of F_1 (edge 1). Data can be read out of the queue as early as the second rising edge of F_2 (edge 3), if and only if $T > T_S$, *i.e.*, `Empty` has become false on the F_2 interface before the next rising edge of F_2 (edge 2). This two-cycle delay is necessary for the reading interface to recognize that the queue is non-empty and enter a state where data can be read from the queue. Therefore, the delay would be seen by F_2, and would be two clock cycles when $T > T_S$ (one rising edge to recognize $\overline{\texttt{Empty}}$ and another rising edge to begin reading) and three cycles when $T \leq T_S$ (the additional rising edge occurring during the interface metastable region). The value of T is determined by the relative frequency and phases of F_1 and F_2 and may change over time. The cost of synchronization is controlled by the relationship between T and T_S.

An optimal design would minimize T_S in order to allow wide frequency and/or phase variations between F_1 and F_2 and increase the probability of a two-cycle delay. Alternatively, controlling the relative frequencies and phases of F_1 and F_2 would allow a two-cycle delay to be guaranteed. Note that this analysis assumes $T_S < \frac{1}{F_1}$ and $T_S < \frac{1}{F_2}$. The analogous situation exists when the queue is `Full`, replacing `Empty` with `Full`, edge 1 with edge 2, and edge 3 with edge 4, in the above discussion.

Figure 6 shows a transistor-level schematic of a synchronization circuit adapted from [17]. The most salient characteristic of this circuit (and others like it [18]) is that it is guaranteed to be glitch-free. This is the case because the internal signals $\texttt{R}_0$ and $\texttt{R}_1$ are integral voltages — note that the voltage source for this stage is not V_{dd} but rather the output nodes of the previous stage. Because these signals are integrals they are guaranteed to be monotonically changing signals. Therefore, $\texttt{R}_0$ and $\texttt{R}_1$ comprise the synchronized input in dual-rail logic. These signals

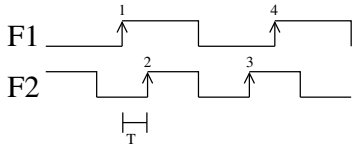


Figure 5. Synchronization timing.

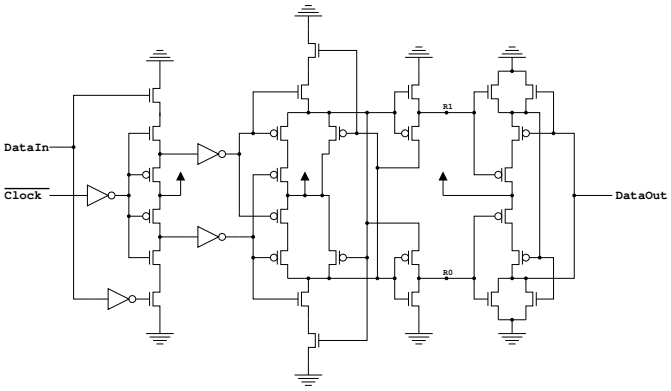


Figure 6. Synchronization circuit.

are then used to drive the last stage of the synchronizer, which is a simple RS latch, to produce a single-rail logic output. The circuit synchronizes the `DataIn` signal on the falling edge of the `Clock`, producing a guaranteed glitch-free and synchronized signal on `DataOut` (recall that the `Valid`[n] signals in Figure 3 are synchronized on the falling edges of the clocks). Figure 7 shows the results of `SPICE` simulation of the synchronization circuit of Figure 6 with a number of falling clock edges coinciding with transitions of the input data signal. Figure 8 shows one such transition at 4.8nsec in detail where it can be seen that the output signal is glitch-free and monotonic. The circuit was simulated using level-8 0.25μm TSMC SCN025 transistor models. This synchronizer circuit provides the necessary component to the synchronizing FIFO of Figure 3. With the data stored in the FIFO regardless of the synchronizer outcome, and the `Valid` signals in the FIFO properly synchronized to *both* domain clocks, the effect is that data is possibly delayed by one clock in the receiving domain (since the receiver cannot 'see' the data until the `Valid` signal has been synchronized).

Since the queue structure proposed does not represent a significant departure from queues already in use in modern microprocessors, we do not expect that the size of these queues would be appreciably impacted by the changes required for synchronization. For the circuit shown in Figure 6, 36 transistors are required per bit of synchronization. Although this is not inconsequential it is also not a significant increase in the overall size requirements of the queue. Notice that the number of synchronization circuits required is a function of the depth of the queue and not the width (Figure 3, one `Valid` bit per data word). Others [5, 6] have demonstrated that the addition of clock synchronization/arbitration circuits to the FIFO structure results in a negligible impact on the total structure area.

5 MCD Synchronization Points

The manner in which the processor is partitioned as part of transforming it into an MCD processor is of critical importance since the performance degradation that will be imposed as a result of the inter-domain communication penalties directly follows from the processor partitions.

The following list identifies all modeled inter-domain communication channels. These communication channels were determined by analyzing the microarchitecture of the MCD partitioning and are a direct consequence of that partitioning. For each communication channel, we discuss the data being transferred between domains and classify the synchronization mechanism required to ensure reliable communication. In addition, each channel is identified as using an existing queue structure or as requiring the addition of the queue entirely due to inter-domain synchronization. Each communication channel is identified by number in Figure 1.

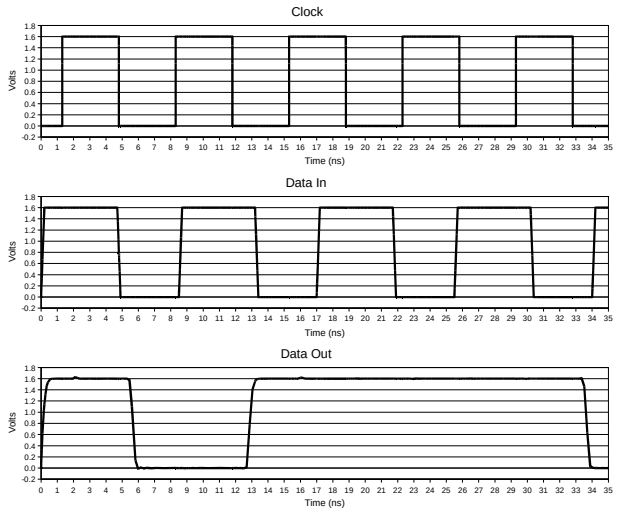


Figure 7. Synchronization timing `SPICE` results.

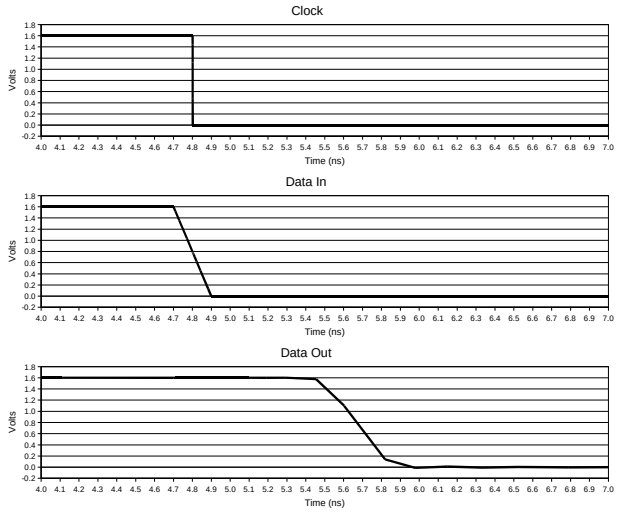


Figure 8. Synchronization timing `SPICE` results (4–7nsec).

Communication Channel 1

Information: L1 Cache Line
Transfer: L1 Instruction Cache $\Leftarrow$ L2 Unified Cache
Domains: Fetch/Dispatch $\Leftarrow$ Load/Store
Reason: L1 Instruction Cache Miss
Synchronization Type: FIFO Queue Structure
Discussion: New queue structure. When a level-1 instruction cache miss occurs, a cache line must be brought in from the level-2 cache. The fetch/dispatch domain only initiates an L2 cache request when there is an L1 I-cache miss, and since the front end *must* stall until that request is satisfied, we know that there can only be one such outstanding request at a time. Therefore, any inter-domain penalty that is to be assessed *cannot* be hidden behind a partially full FIFO. Hence, the synchro-

nization penalty must be determined based solely on the relative frequency and phase of the domain clocks for each communication transaction. Although communication channels such as this one, which can only contain one element at a time, would be implemented as a single synchronization register, we prefer to maintain the generalized FIFO model since the performance characteristics of both are identical.

Communication Channel 2

Information: L2 Cache Line
Transfer: L2 Unified Cache ⇔ Main Memory
Domains: Load/Store ⇔ Memory
Reason: L2 Cache Miss (Data Reference *or* Instruction)
Synchronization Type: FIFO Queue Structure
Discussion: Existing queue structure. This communication path, although logically bidirectional, is composed of two independent unidirectional channels. Since they are similar and related, they will be addressed together. Notice also that the reason for this communication can be either an L2 cache miss due to a data reference, *or* an L2 cache miss due to an instruction fetch. The characteristics of this interface are the same as the previous description with the exception that multiple outstanding requests to the L2 unified cache are supported. That fact alone makes it *possible* that some of the inter-domain synchronization penalties can be hidden. As it turns out, there are few times when there are enough outstanding requests such that the penalty would actually be hidden. For our MCD processor, there would have to be more than 5 outstanding requests (See Section 4.1) to ensure that the penalty did not impact memory access time. Fortunately, the inter-domain penalty is only one cycle. Given that the memory access time for the first L2 cache line is 80 cycles (subsequent, contiguous lines have an access time of 2 cycles), the inter-domain synchronization penalty alone is not likely to have an appreciable impact (the penalty represents an increase in memory access time of 1.06–1.25%, based on 8-word cache line fill transactions).

Communication Channel 3

Information: Branch Prediction Outcome
Transfer: Branch Predictor ⇐ Integer ALU
Domains: Fetch/Dispatch ⇐ Integer
Reason: Branch Instruction
Synchronization Type: FIFO Queue Structure
Discussion: New queue structure. When a branch instruction is committed, the machine state must be updated with the branch outcome (*i.e.*, `Taken`/$\overline{\texttt{Taken}}$). In addition, the fetch unit needs to know if the prediction was correct or not since it must change fetch paths if the prediction was not correct. This communication path is, by its very nature, sequential. That is, branch outcomes need to be processed by the fetch/dispatch domain in a strict first-in-first-out manner. This is also a communication path that is not likely to benefit from the penalty hiding property of the FIFO queue structure since doing so would require 5 or more branch outcomes to be buffered waiting to be processed by the fetch/dispatch domain.

There is another aspect of a branch mis-prediction that must be properly handled. That is, when a branch is mis-predicted, all speculative instructions must be squashed from the machine pipeline. This can happen as soon as the branch outcome is known and does not need to occur after the branch predictor is updated. The communication in this case occurs from the integer domain (where the branch outcome is determined) to all other domains. To handle this case, we chose to squash speculative instructions on the rising edge of each domain clock *after* the synchronization has occurred in the fetch/dispatch domain. Although this represents a conservative approach, it simplifies the design considerably since it eliminates the synchronization FIFO between the integer domain and the floating-point and load/store domains. The effect of implementing speculative instruction squashing in this manner is that resources used by those soon-to-be-squashed instructions are used longer than they would be otherwise. This *may* have a negative impact on performance if instructions cannot be issued because processor resources are in use by one of these instructions.

Communication Channel 4

Information: Integer Load Result
Transfer: Integer Register File ⇐ L1 Data Cache
Domains: Integer ⇐ Load/Store
Reason: Load Instruction (Integer only)
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. The load value may originate in the Load/Store Queue (LSQ), the L1 Data Cache, the L2 Cache or Main Memory; any synchronization that is required to get the value to the LSQ is assumed to have already been assessed. This communication channel handles only the synchronization between the LSQ and the integer register file. Since loads can complete out-of-order (a load that is found in the LSQ could have issued after a load that is waiting on a cache miss), the synchronization mechanism in this case must be of the issue queue structure type since the load result is useful to the integer domain as soon as possible after becoming available.

Communication Channel 5

Information: Floating-Point Load Result
Transfer: Floating-Point Register File ⇐ L1 Data Cache
Domains: Floating-Point ⇐ Load/Store
Reason: Load Instruction (Floating-Point only)
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. Floating-point load instructions are identical to integer loads with one exception: the destination of the load is the floating-point register file in the floating-point domain. All synchronization characteristics are the same as integer loads, and as such, the required interface is an issue queue type structure.

Communication Channel 6

Information: Effective Address
Transfer: Load/Store Queue ⇐ Integer Result Bus
Domains: Load/Store ⇐ Integer
Reason: Load or Store Instruction
Synchronization Type: Issue Queue Structure

Discussion: Existing queue structure. Each load and store instruction is broken into two operations: i) the load/store operation, which accesses the L1 data cache, and ii) the addition operation, which computes the effective address for the memory reference. The load/store operation is issued to the load/store domain, and the effective address calculation is issued to the integer domain. The load/store operation is dependent on the address calculation since the memory reference cannot be performed until the address is known. Since both integer operations in the integer issue queue (IIQ) and load/store operations in the load/store queue (LSQ) are free to proceed out-of-order, the effective address calculation can *potentially* be used by the LSQ as soon as the calculation is complete, regardless of the program order in which the memory references were made. These characteristics require that the effective address result that is transferred from the integer domain to the load/store domain be stored in an issue queue structure.

Communication Channel 7

Information: Integer Value
Transfer: Integer Register File ⇐ Floating-Point Result Bus
Domains: Integer ⇐ Floating-Point
Reason: `FloatCvt` Instruction (FP to Integer Conversion)
Synchronization Type: FIFO Queue Structure
Discussion: New queue structure. Converting a floating-point value into an integer value is performed by the floating-point hardware, but the result is written to the integer register file. In order to ensure that the converted value is properly received in the integer domain, a simple first-in-first-out structure is necessary since there is no requirement that the operations occur out of order. That being the case, it is also unlikely that the synchronization penalty can be hidden by the occupancy of the FIFO unless a stream of conversion operations is performed. Although this situation is rare, it may occur for vector floating-point applications. In those cases, the existence of the FIFO is likely to reduce the effective synchronization penalty seen by the application.

Communication Channel 8

Information: Floating-Point Value
Transfer: Floating-Point Register File ⇐ Integer Result Bus
Domains: Floating-Point ⇐ Integer
Reason: `FloatCvt` Instruction (Integer to FP Conversion)
Synchronization Type: FIFO Queue Structure
Discussion: New queue structure. Same as previous, except that the conversion is now from integer to floating-point. The same application characteristics, *i.e.*, vector floating-point conversions, would be required in order to see the benefit of the FIFO queue structure.

Communication Channel 9

Information: Integer Instruction
Transfer: Integer Issue Queue ⇐ Integer Instruction
Domains: Integer ⇐ Fetch/Dispatch
Reason: Integer Instruction Issue
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. When instructions are dispatched into an issue queue, the scheduler must have access to those instructions as soon as possible in a superscalar processor. Since instructions issue out-of-order, scheduling decisions are based on the instructions present in the issue queue, regardless of the order in which those instructions were inserted. It is clear that an issue queue structure is required and that the synchronization penalty associated with the dispatch of an instruction cannot be hidden, regardless of the number of entries in the queue. Note that as long as there are instructions in the queue the scheduler is able to proceed, albeit without knowledge of instructions which are not visible due to synchronization delays.

Communication Channel 10

Information: Floating-Point Instruction
Transfer: Floating-Point Issue Queue ⇐ Floating-Point Instruction
Domains: Floating-Point ⇐ Fetch/Dispatch
Reason: Floating-Point Instruction Issue
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. Same as previous, but for floating-point instructions.

Communication Channel 11

Information: Load/Store Operation
Transfer: Load/Store Queue ⇐ Load/Store Operation
Domains: Load/Store ⇐ Fetch/Dispatch
Reason: Load/Store Instruction Issue
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. Same as previous, but for load/store instructions.

Communication Channel 12

Information: Integer Instruction Commit
Transfer: Reorder Buffer ⇐ Integer Instruction Completion
Domains: Fetch/Dispatch ⇐ Integer
Reason: Integer Instruction Completion
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. When instructions complete execution, the machine state must be updated. Although the update of machine state occurs in program order, the completion of instructions is out-of-order. The Reorder Buffer (ROB) is the structure that commits instructions in program order. The ROB does this by examining all instructions that have completed execution. When an instruction has completed execution and it is the next instruction in program order, it is committed. If only one instruction could be committed in a given cycle, then instruction completion information could be transferred from the various domains in-order (*i.e.*, in order of completion, *not* in program order). Committing only one instruction per cycle would significantly degrade the performance of a superscalar processor that can issue and execute more than one instruction in parallel. Therefore, the ROB must be capable of committing more than one instruction per cycle. The ability to commit more than one instruction per cycle requires that the synchronization mechanism used for the completion information be of the issue queue structure type since the ROB can use the completion information in any order. Note that the synchronization delay may result in increased pressure (*i.e.*, higher average occupancy) on the ROB structure.

Communication Channel 13

Information: Floating-Point Instruction Commit
Transfer: Reorder Buffer ⇐ FP Instruction Completion
Domains: Fetch/Dispatch ⇐ Floating-Point
Reason: Floating-Point Instruction Completion
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. Same as previous, but for floating-point instructions.

Communication Channel 14

Information: Load/Store Operation Commit
Transfer: Reorder Buffer ⇐ Ld/St Operation Completion
Domains: Fetch/Dispatch ⇐ Load/Store
Reason: Load/Store Instruction Completion
Synchronization Type: Issue Queue Structure
Discussion: Existing queue structure. Same as previous, but for load/store instructions.

Communication Channel 15

Information: L1 Cache Line
Transfer: L1 Instruction Cache ⇐ Main Memory
Domains: Fetch/Dispatch ⇐ Memory
Reason: L1 Instruction Cache Miss
Synchronization Type: FIFO Queue Structure
Discussion: New queue structure. If the processor is configured *without* a level-2 cache, then L1 I-cache misses must be filled from main memory. This interface is a simple FIFO since the front end must wait for L1 I-cache misses to complete, so there is no benefit to allowing out-of-order transactions. Note that this channel only exists in processor configurations *without* an L2-cache.

6 Simulation Methodology

Our simulation environment is based on the SimpleScalar toolset [4] with the Wattch [2] power estimation extension and the MCD processor extensions [20]. The MCD extensions include modifications to more closely model the microarchitecture of the Alpha 21264 microprocessor [12]. A summary of our simulation parameters for the Alpha 21264-like processor appears in Table 2. For comparison to an in-order processor, we also simulate a StrongARM SA–1110-like processor. A summary of the simulation parameters for that processor is given in Table 3.

We selected a broad mix of compute bound, memory bound, and multimedia applications from the MediaBench, Olden, and Spec2000 benchmark suites (shown in Figure 9).

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

Table 2. Architectural parameters for simulated Alpha 21264-like processor.

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

Table 3. Architectural parameters for simulated StrongARM SA–1110-like processor.

Configuration Parameter	Value
Instruction Issue	In-order
Branch predictor	None (assume not-taken)
Branch Mispredict Penalty	4 cycles
Decode Width	1 instruction
Issue Width	2 instructions
Retire Width	2 instructions
L1 Data Cache	8KB, 32-way set associative
L1 Instruction Cache	16KB, 32-way set associative
L2 Cache	None
L1 cache latency	1 cycle
Integer ALUs	1 + 1 mult/div unit
Floating-Point ALUs	1 + 1 mult/div/sqrt unit
Integer Issue Queue	16 entries
Floating-Point Issue Queue	16 entries
Load/Store Queue	12 entries (4 load / 8 store)
Physical Register File	32 integer, 32 floating-point

Synchronization performance degradation is measured by comparing the overall program execution time of the MCD processor with an identically configured, fully synchronous (*i.e.*, single, global clock domain) processor. The fully synchronous processor is clocked at 1.0GHz. Each domain of the MCD processor is clocked by an independent 1.0GHz clock (*i.e.*, independent jitter for each domain clock, no phase relationship between any domain clocks).

7 Results

The average performance degradation of the Alpha 21264-like MCD processor over all 30 benchmarks is less than 2%. The individual results show (Figure 9) that there is variation in the performance degradation caused by the MCD microarchitecture ranging from a maximum of 3.7% for `swim` to a performance *improvement* of 0.7% for `power`. Although this performance improvement is quite modest, it is surprising since we expect the MCD microarchitecture to impose a performance penalty. To understand what is happening, we have to look

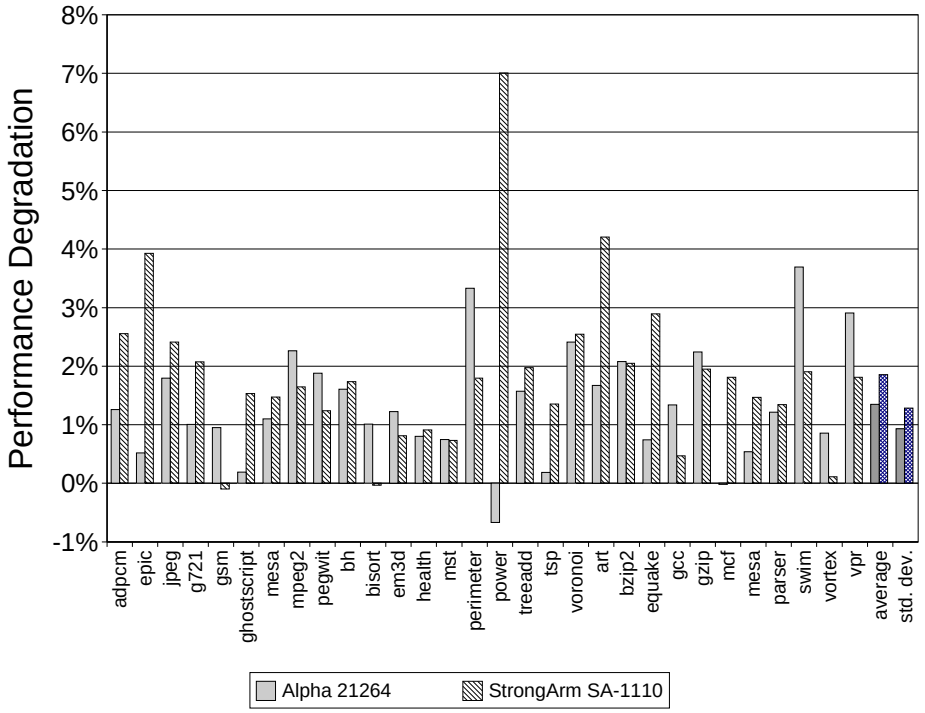


Figure 9. Application performance degradation of MCD processor over fully synchronous processor.

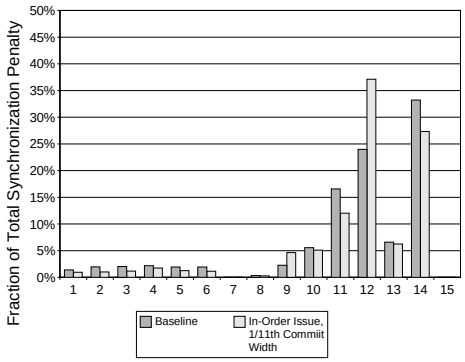
closely at what happens to instructions within an MCD processor compared with a fully synchronous processor. Within the MCD processor there is the potential for individual instructions to take additional cycles to execute due to the inter-domain communication penalties inherent in the MCD microarchitecture. In most cases this is exactly what happens: individual instructions take additional cycles to execute which results in performance degradation for the entire application execution. There are side effects to adding cycles to the execution of individual instructions, *i.e.*, the instruction commit rate and cache access patterns are changed slightly. Although these changes typically do not have a significant impact on the application execution, there are cases where the impact is appreciable. This is precisely the case for `power`. By examining detailed per-interval processor statistics, we could determine that the change in instruction timing causes an *increase* in the branch mispredict penalty of 2.2% but a *decrease* in the average memory access time of approximately 1.2%. Since greater than one out of every four instructions in the simulated instruction window is a memory reference, and less than one in eight is a branch, the decreased memory access time translates into an overall performance improvement.

Intuitively, it makes sense that there would be higher performance degradation and greater variation for the in-order StrongARM MCD processor than for the out-of-order Alpha 21264. In fact, the maximum performance degradation for the SA–1110-like MCD processor is 7.0% for `power` and a 0.1% performance *improvement* for `gsm`. The larger range of performance degradation occurs because the internal operations of the StrongARM are inherently more serial than those of the Alpha 21264-like processor. At first glance it would seem that the performance degradation of the StrongARM-like processor should be significantly higher than that of the Alpha 21264-like processor given the vastly different architectures. The percent change in performance is somewhat misleading in this regard. The baseline StrongARM processor has a Cycles Per Instruction (CPI) of approximately 3.5, whereas the Alpha 21264 baseline CPI is approximately 1.0. Therefore, although the performance degradation percent values are not significantly different, the actual change in CPI is larger for the in-order processor. When comparing the changes in CPI directly, the increase for the StrongARM processor is nearly $5\times$ that of the Alpha processor configuration. As such, a significant portion of the MCD performance degradation can be eliminated by the addition of out-of-order superscalar execution features as demonstrated in Table 4, using the in-order StrongARM-like processor as a starting point; where *Performance Degradation Eliminated* is the percentage of the baseline performance degradation that was eliminated as a result of adding out-of-order superscalar execution features to the in-order processor architecture.

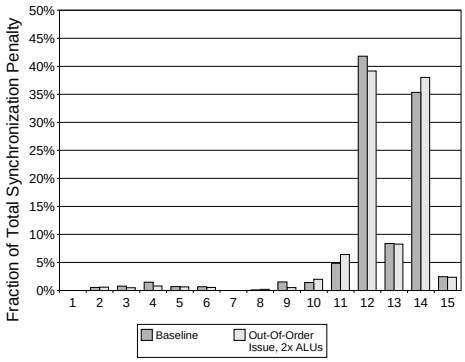
Table 4. StrongARM SA–1110 performance degradation eliminated by out-of-order superscalar architectural features.

StrongARM SA–1100	
Out-Of-Order Issue	16%
Out-Of-Order Issue and 4× Decode	53%
Out-Of-Order Issue and 2× ALUs	62%

Inter-domain synchronization penalties naturally occur over the most highly utilized communication channels. However, the synchronization penalties do not necessarily result in performance degradation. This is true because the most highly utilized channels (shown in Figure 10 for the Alpha 21264-like and StrongARM-like families of MCD processors) are the most dominant channels, *i.e.*, the integer instruction commit (#12)



(a) Alpha 21264.



(b) StrongARM SA–1110.

Figure 10. MCD Inter-domain synchronization overhead (synchronization channels are as shown in Figure 1).

and load/store instruction commit (#14), and the inter-domain synchronization penalties in this case are largely hidden by the Re-order Buffer (ROB). This means that an MCD processor is likely to put greater pressure on the ROB resources and could potentially benefit from a larger ROB. Close examination of Figure 1 would show that channel pairs #9 & #12 and #11 & #14 should incur the same fraction of synchronization penalties since these pairs represent instruction processing through the integer and load/store domains, respectively. The reason for the somewhat counter-intuitive results of Figure 10 is that channels #9 and #11 (instruction dispatch) are significantly wider than channels #12 and #14 (instruction completion). Whereas instruction dispatch width is governed by the type of instructions present in the dynamic instruction stream, instruction completion is governed by the actual instruction level parallelism available, which is likely to be considerably less. This means that for the same number of instructions passing through channel pairs #9 & #12 and #11 & #14, the instruction completion paths will contain information more frequently, but with each transaction being smaller than for the instruction dispatch paths. This results in more possible synchronization penalties being accessed over the instruction completion paths.

Table 5 shows this characteristic for two Alpha 21264-like processor and two Strong ARM SA–1110-like processor configurations. In these figures, *Synchronization Time* is the percentage of the total execution time that events within

Table 5. Architecture effects on MCD inter-domain synchronization penalty.

Alpha 21264		
	Baseline	In-Order Issue and Commit Width=1
Performance Degradation	1.3%	2.4%
Synchronization Time	24.2%	21.5%
Hidden Synchronization Cost	94.3%	87.3%
StrongARM SA–1100		
	Baseline	Out-Of-Order Issue and 2× ALUs
Performance Degradation	1.9%	0.7%
Synchronization Time	12.2%	10.9%
Hidden Synchronization Cost	83.4%	94.1%

Table 6. MCD processor Cycles Per Instruction (CPI).

Alpha 21264	
Baseline	1.04
In-Order Issue	2.17
In-Order Issue and $\frac{1}{10^{th}}$ Extra Registers	2.20
In-Order Issue and $\frac{1}{11^{th}}$ Commit Width	2.18
StrongARM SA–1100	
Baseline	3.51
Out-Of-Order Issue	3.23
Out-Of-Order Issue and 4× Decode	3.23
Out-Of-Order Issue and 2× ALUs	3.21

the processor were delayed due to inter-domain synchronization overhead. *Hidden Synchronization Cost* is the percentage of synchronization time which did not result in actual performance degradation. For the aggressive out-of-order superscalar 21264-like processor, the figure shows that as the out-of-order and superscalar features of the processor are eliminated, more of the inter-domain synchronization penalty results in performance degradation. The opposite effect is seen when an in-order processor such as the StrongARM SA–1110 is augmented with out-of-order superscalar architectural features. Table 6 shows that although various out-of-order superscalar features were removed from the 21264-like processor, and various out-of-order superscalar features were added to the SA–1110-like processor, the effects of these changes on CPI were consistent. This underscores the idea that the in-order versus out-of-order instruction issue and serial versus superscalar execution are the most dominant factors in determining the effects of inter-domain synchronization overhead.

8 Related Work

There has been significant research into the characteristics, reliability, and performance of low-level synchronization circuits. In [23], two asynchronous data communication mechanisms are presented, which utilize self-timed circuits to achieve reliable and low-latency data transfers between independently clocked circuits. In [16], a pausible clock is used to ensure reliable communication and an arbiter circuit is used to reduce the frequency of occurrence of delaying the clock. Other approaches to synchronization have been proposed, such as [22], which detect when the possibility of a metastability problem may occur (*i.e.*, data transitioning too close to a clock edge) and delay either the data or the clock until there is no possibility of metastability.

In addition to the evaluation of low-level synchronization circuits, there has been research into high-level evaluation of GALS systems. In [15], the potential power savings of a GALS system is quantified. In addition, an algorithmic technique for partitioning a synchronous system into a GALS system is presented with the goal of achieving low power. This work shows promising power savings results. Unfortunately, corresponding performance evaluation was not presented. In [10], a GALS processor with 5 independent domains is evaluated with respect to power consumption and performance degradation. This work is most closely related to our analysis. A conclusion of this work is that with their GALS processor a performance degradation of 10% on average was achieved. The authors also conclude that studies of latency hiding techniques are needed to more fully understand the architectural reasons for performance degradation. We concur with this conclusion and have performed this study on our GALS processor. We attribute the disparity in performance degradation between these two studies to the domain partitioning differences (*i.e.*, five versus four domains). We have determined that a clock domain partitioning that separates the fetch circuits (instruction cache and branch predictor) from the dispatch circuits can significantly degraded overall performance.

9 Conclusions

We have shown that the out-of-order superscalar execution features of modern, aggressive processors are the same features that minimize the impact of the MCD synchronization overhead on processor performance. With an aggressive, out-of-order superscalar processor such as the Alpha 21264, as much as 94% of the inter-domain synchronization penalties are effectively hidden and do not result in performance degradation. When out-of-order superscalar execution features are added to a simple in-order serial processor like the StrongARM SA–1110, as much as 62% of the MCD performance degradation can be eliminated. Our prior work studied the potential energy saving advantage of our GALS MCD processor. In that work we demonstrated approximately 20% reduction in energy consumption. These results combine to demonstrate the overall benefit of our GALS MCD processor design.

The resulting modest performance impact due to synchronization can potentially be overcome by faster per-domain clocks permitted by the removal of a global clock skew requirement, and the ability to independently tune each domain clock. These are areas proposed for future work.

References

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

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

[3] E. Brunvand, S. Nowick, and K. Yun. Practical Advances in Asynchronous Design and in Asynchronous/Synchronous Interfaces. In *36th Design Automation Conference*, June 1999.

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

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

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

[7] T. Chelcea and S. M. Nowick. Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols. In *Design Automation Conference*, pages 21–26, June 2001.

[8] 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.

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

[10] A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *Proceedings of the* 29^{th} *International Symposium on Computer Architecture*, May 2002.

[11] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

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

[13] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 Microprocessor Architecture. In *Proceedings of the International Conference on Computer Design*, pages 90–95, Oct. 1998.

[14] 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 *The Proceedings of the* 30^{th} *International Symposium on Computer Architecture*, June 2003.

[15] T. Meincke, A. Hemani, S. Kumar, P. Ellervee, J. Berg, D. Liqvist, H. Tenhunen, and A. Postula. Evaluating benefits of Globally Asynchronous Locally Synchronous VLSI Architecture. In *Proceedings of the* 16^{th} *NORCHIP Conference*, pages 50–57, Nov. 1998.

[16] S. Moore, G. Taylor, B. Mullins, and P. Robinson. Channel Communication Between Independent Clock Domains. In *First ACiD-WG Workshop of the European Commission's Fifth Framework Programme*, Feb. 2001.

[17] M. Nyström and A. J. Martin. Crossing the Synchronous-Asynchronous Divide. In *Workshop on Complexity-Effective Design*, May 2002.

[18] F. I. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang. Q-Modules: Internally Clocked Delay-Insensitive Modules. *IEEE Transactions on Computers*, 37(9):1005–1018, Sept. 1988.

[19] 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 *The Proceedings of the* 35^{th} *Annual Symposium on Microarchitecture*, Nov. 2002.

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

[21] M. Singh and S. M. Nowick. High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2000.

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

[23] F. Xia, A. Yakovlev, D. Shang, A. Bystov, A. Koelmans, and D. J. Kinniment. Asynchronous Communication Mechanisms Using Self-Timed Circuits. In *Proceedings of the 6th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2000.