Techniques for Reducing Read Latency of Core Bus Wrappers

Roman L. Lysecky, Frank Vahid, Tony D. Givargis
Department of Computer Science and Engineering
University of California, Riverside
{rlysecky, vahid, givargis}@cs.ucr.edu, www.cs.ucr.edu/~dalton

Abstract

Today's system-on-a-chip designs consist of many cores. To enable cores to be easily integrated into different systems, many propose creating cores with their internal logic separated from their bus wrapper. This separation may introduce extra read latency. Pre-fetching register data into register copies in the bus wrapper can reduce or eliminate this extra latency. In this paper, we introduce a technique for automatically designing a pre-fetch unit that satisfies user-imposed register-access constraints. The technique benefits from mapping the pre-fetching problem to the well-known real-time process scheduling problem. We then extend the technique to allow user-specified register interdependencies, using a Petri Net model, resulting in even more efficient pre-fetch schedules.

Keywords

Cores, system-on-a-chip, interfacing, on-chip bus, intellectual property, design reuse, bus wrapper.

1. Introduction

Today's system designers can incorporate increasingly larger numbers of components into a single design [9]. This rapid growth of system-on-a-chip designs has led to the marketability of Intellectual Property (IP) cores, available in various forms ranging from soft cores to hard cores (e.g., [4]). The ability to easily integrate a core into a system can increase a core's usefulness. Early efforts by the Virtual Socket Interface Alliance (VSIA) [12] focused on defining a standard on-chip bus, but this was soon viewed to be infeasible. Instead, they are now defining a standard that prescribes creating cores with internal behavior separated from the bus interface logic through a bus wrapper. Other researchers have also proposed such separation as a method to enable cores to be easily integrated into different systems [2][8][10]. As shown in Figure 1, the wrapper interfaces on one-side with an on-chip bus, and on the other side with a core internal bus. It is this core internal bus that the VSIA is standardizing -- they refer to it as the Virtual Component Interface. Targeting a core to a specific on-chip bus thus requires changes only to its bus wrapper, making integration easier than if changes to the core's internals were required.

Separating the internal behavior from the bus wrapper can lead to extra register read latency when the on-chip bus (OCB) and internal bus (IB) (see Figure 1) are very different. When similar, the wrapper can detect an OCB read request and convert it to an IB request in the same cycle, and likewise pass the read data from the IB to the OCB in one cycle, for a total of 2 cycles for a read. When the buses are very different, though, the wrapper may have to latch the OCB read request before converting it to an IB read on the next cycle, and likewise it may have to latch the data read from the IB before putting it on the OCB, yielding 2 extra cycles, or 4 cycles total for a read. In this paper, we assume that the buses are different and hence 4 cycle reads would be required without pre-fetching.

In [6], we introduced a technique, called pre-fetching, for reducing the extra cycles. Pre-fetching, similar to caching, keeps local copies of a core's internal registers directly in the bus wrapper, and updates those copies during cycles when the OCB is not accessing the core. 2 cycle reads are thus possible for pre-fetched registers, since the wrapper can respond immediately to an OCB read request with data from its own register copies. In this paper, we describe a technique for automatically designing a pre-fetch unit that meets user-provided constraints on maximum allowable latency and data-age for each register. The key design problem is to schedule the pre-fetches over the core's internal bus such that the constraints are met. We observed that the pre-fetching problem could be mapped to the extensively researched real-time process scheduling problem, and thus we apply powerful heuristics and analysis techniques for that problem to solve the pre-fetching problem. Also in this paper, we consider the case where the core designer is able to provide additional information about the registers, in particular, their update interdependencies, which we can use to build an even better pre-fetch schedule. In this case, we use Petri Nets as a model for specifying a core's register update dependencies, and we also provide a heuristic for scheduling pre-fetches based on that model.

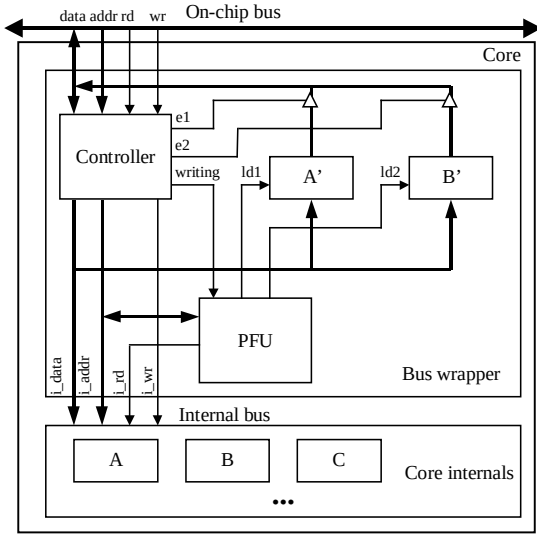
Much work has been done on interfacing with cores, but to our knowledge none of the literature includes the idea of pre-fetching. The bulk of interfacing work has dealt with the automatic synthesis of logic to interface with a bus (e.g., [2][7]), synthesis of the bus itself (e.g., [3]), or defining a standard bus protocol (e.g., [11]). Pre-fetching of course has been applied before to microprocessors and memories, but our focus is on peripheral cores.

2. Problem definition

The basic architecture of a sample core with a pre-fetching bus wrapper is shown in Figure 1. The details of the core internals are omitted except for three registers, A, B, and C. The wrapper has three parts:

1. The *pre-fetch registers* are copies of readable registers in the core internals. In the figure, we assume A and B are readable, so we create pre-fetch registers A' and B'.

Figure 1: Bus wrapper with pre-fetching



2. The *controller* detects OCB write requests corresponding to one of the core's registers, and converts those requests into writes over the internal bus. The controller also detects OCB read requests, and puts data on the OCB from the appropriate pre-fetch register.
3. The *pre-fetch unit (PFU)* is responsible for keeping the pre-fetch registers as up-to-date as possible, by pre-fetching the core's internal registers over the internal bus, when the internal bus is not being used for a write by the controller, i.e., during internal bus *idle* cycles. Only one register can be read from the core internals at a time. Our focus is the design of the PFU.

We assume we are given a list of the core's readable registers, which must be pre-fetched. We also assume that the bus wrapper can accommodate one copy of each such register (future work may consider the situation where only a subset of registers fit in the bus wrapper). Each register in the list is annotated with two important read-access constraints:

- *Register age constraint:* This constraint represents the number of cycles old that data may be when read. In other words, it represents the period during which the pre-fetch register must be updated at least once. An age constraint of 0 means that the data must be the most recent data, which in turn means that the data must come directly from the core and hence pre-fetching is not allowed, since pre-fetched data is necessarily at least one cycle old. A constraint of 0 also means that the access-time constraint must be at least 4 cycles.
- *Register access-time constraint:* This constraint represents the maximum number of cycles that a read access may take. The minimum is 2, in which case the register must be pre-fetched. An access-time constraint greater than 2 denotes that additional cycles may be tolerated.

We wish to design a PFU that reads the core internal registers into the pre-fetch registers using a schedule that satisfies the age and access-time constraints on those registers. Note that certain registers may be pre-fetched more frequently than others if this is required to satisfy differing register access constraints.

Table 1: Pre-fetch scheduling for the core in Figure 1.

Idle cycle	Schedule 1	Schedule 2
0	A	A
1	B	B
2	A	
3	A	
4	B	A
5	A	
6	B	
7	A	B
8	A	A
9	B	

The tradeoff of pre-fetching is performance improvement at the expense of size and power. Our main goal is performance improvement, but we should ensure that size and power do not grow more than an acceptable amount. Future work may include optimizing a cost function of performance, size and power.

For example, Figure 1 shows a core with three registers, A, B, and C. We assume that registers A and B are independent registers that are read-only, and updated randomly by the core internals. Assume that A and B have register age constraints of 4 and 6 cycles, respectively. We might use a naive pre-fetching heuristic that pre-fetches on every idle cycle, reading A 60% and B 40% of the time, leading to Schedule 1 in Table 1. However, we can create a more efficient schedule, as shown in Schedule 2. Although both schedules will meet the constraints, the first schedule will likely consume more power. The naive scheduler also does not consider the effects of register writes, which will be taken into consideration using real-time scheduling techniques.

3. "Real-time" pre-fetching

During our investigation for heuristics to solve the pre-fetching problem, we noticed that the problem could be mapped to the widely-studied problem of real-time process scheduling, for which a rich set of powerful heuristics and analysis techniques already exist. We now describe the mapping and then provide several pre-fetching heuristics (based on real-time scheduling heuristics) and analysis methods.

3.1 Mapping to real-time scheduling

A simple definition of the real-time scheduling problem is as follows. Given a set of N independent periodic processes, and a set of M processors, we must order the execution of the N processes onto the M processors. Each process has a period, P_i, a deadline, D_i, and a computation time, C_i. The period of a process is the rate at which the process requests execution. The deadline is the length of time in which a process must complete execution after it requests to be executed. Finally, the computation time is the length of time a process takes to perform its computation. Therefore, real-time scheduling is the task of ordering the execution of the N processes among the M processors, to ensure that each process, executes once every period P_i and within its deadline D_i, where each process takes C_i time to complete.

A mapping of the pre-fetching problem to the real-time process-scheduling problem is as follows.

Table 2: Pre-fetch core descriptions.

Core	Reg.	Max Age	D	Pri. RM	PF Time	Resp. Time	Util.	Util. Bnd
1	DATA	3	2	1	2	2	66.7	100
2	GCD1	10	2	1	2	2	50.0	78.0
	GCD2	10	2	2	2	4		
	CS	20	2	3	2	6		
3	STAT	5	2	1	2	2	86.0	75.7
	A	25	2	3	2	8		
	B	25	2	4	2	16		
	RES	10	2	2	2	4		

- *Register → Process:* A register that must be scheduled for pre-fetching corresponds to a process that must be scheduled for execution.
- *Internal bus → Processor:* The internal bus can accommodate only one pre-fetch at a time. Likewise, a processor can accommodate only one process execution at a time. Thus, the internal bus corresponds to a processor.
- *Pre-fetch → Process execution:* A pre-fetch occurs over the internal bus, and thus corresponds to a process execution occurring on a processor.
- *Register age constraint → Process Period:* The register age constraint defines the period during which the register must be pre-fetched, which corresponds to the period during which a process must be scheduled.
- *Register access-time constraint → Process Deadline:* The access-time constraint defines the amount of time a read may take relative to the read request, which corresponds to the amount of time a process must complete its execution relative to the time it requested service.
- *Pre-fetch time → Process computation time:* A pre-fetch corresponds to a process execution, so the time for a pre-fetch corresponds to the computation time for a process. In this paper, we assume a pre-fetch requires 2 cycles, although the heuristics and analysis would of course apply if we extended the register model to allow for (the rather rare) situation where different registers would require different amounts of time to read them from the core internals.

Given this mapping, we can now use several known real-time scheduling and analysis techniques to solve the pre-fetching problem.

3.2 Cyclic executive approach

The cyclic executive approach [1] is a straightforward process scheduling method that can be used for a fixed set of periodic processes. The approach constructs a fixed repeating schedule called a major cycle, which consists of several minor cycles of fixed duration. The minor cycle is the rate at which the process with the highest priority will be executed. The minor cycle is therefore equal to the smallest age of the registers to be pre-fetched. This approach is attractive due to its simplicity. However, it does not handle sporadic processes (in our case, sporadic writes), all process periods (register-age constraints) must be a multiple of the minor cycle time, and constructing the executive may be computationally infeasible for a large number of processes (registers).

To serve as examples, we describe three cores with various requirements. Table 2 contains data pertaining to all three of our cores. Table 2 contains information regarding maximum register age constraint (*Max Age*), register access time constraint or deadline (*D*), rate monotonic priority assignment (*Pri. RM*), time required to pre-fetch register (*PF Time*), response time of register (*Resp. Time*), utilization for register set (*Util.*), and utilization bound for register set (*Util Bnd.*). Core1 implements a single channel DAC converter. Although the analog portion of the converter could not be modeled in VHDL, the technique for converting the analog input was implemented. The core has a single register, DATA, that is read-only and updated randomly externally from the system. Core2 calculates the Greatest Common Divisor (GCD) of three inputs while providing checksum information for the inputs and the result. The core contains three registers, GCD1, GCD2, and CS. The result from the GCD calculator is valid when GCD1 is equal to GCD2. Registers GCD1, GCD2, and CS are independent read-only registers that are updated externally from the system. Core3 has five registers, STAT, BIAS, A, B, and RES. STAT is a status register that is read-only, and indicates the status of the core (i.e., busy or not busy). Registers A and B are read-only registers that are updated randomly from outside the system. RES is a read-only register containing the results of some computation on registers A, B, and BIAS, where BIAS is a write-only register that represents some programmable adjustment in the computation.

We can use the cyclic executive approach to create a schedule for each of our three cores. For Core1, both the minor cycle and major cycles are 3. For Core2, the minor cycle is 10 and the major cycle is 20. Finally, for Core3, we can construct a cyclic executive with a minor cycle of 5 and a major cycle of 25.

3.3 Rate monotonic priority assignment

A more general scheduling approach can be used for more complex examples, wherein we determine which process to schedule (register to pre-fetch) next based on a priority scheme. A rate monotonic priority assignment [1] assigns a priority to each register based upon its age. The register with the smallest age will have the highest priority. Likewise, the register with the largest age will have the lowest priority. For our examples we will use a priority of one to indicate the highest priority possible. Rate monotonic priority assignment is known to be optimal in the sense that if a process set can be scheduled with a fixed-priority assignment scheme, then the set can also be scheduled with a rate monotonic assignment scheme.

We again refer to Table 2 for data pertaining to all three of our cores. For Core1, the register age constraint of the register DATA is 3 cycles. Given that DATA is the only register present, it is assigned the highest priority. Core2's registers GCD1, GCD2, and CS have age constraints of 10, 10, and 20 respectively. Therefore, the corresponding priorities from highest to lowest are GCD1, GCD2, and CS. However, because the register age constraint for GCD1 and GCD2 are equal, the priorities for Core2 could also be, from highest to lowest, GCD2, GCD1, and CS. It is important to note that the priorities of registers with the same age constraint can be assigned arbitrary relative priorities as long as the constraints are met.

Equation 1: Utilization-based schedulability test.

$$\sum_{i=1}^{N}\left(\frac{C_i}{A_i}\right) < N\left(2^{1/N}-1\right)$$

For Core3, the age constraints for the registers STAT, A, B, and RES are respectively 5, 25, 25, and 10. Therefore, the priority of the registers from highest to lowest would be STAT, RES, A, and B.

3.4 Utilization-based schedulability test

The utilization-based schedulability test [5] is used to quickly indicate whether a set of processes can be scheduled, or in our case whether the registers can be pre-fetched. All N registers of a register set can be pre-fetched if Equation 1 is true, where C_i is the computation time for register i, A_i is the age constraint of register i, and N is the number of registers to be pre-fetched. The left-hand side of the equation represents the utilization bound for a register set with N registers, and the right-hand side represents the current utilization of the given register set.

If the register set passes this test, all registers can be pre-fetched and no further schedulability analysis is needed. However, if the register set fails the test, a schedule for this register set that meets all constraints might still exist. In other words, the utilization-based schedulability test will indicate that a register set can be pre-fetched, but does not indicate that a register set cannot be pre-fetched.

We can analyze our cores to determine whether we can schedule them. From Table 2, we can see that both Core1 and Core2 pass the utilization-based schedulability test with respective utilizations of 66.7% and 50.0%, where the corresponding utilization bounds were 100% and 78.0%. This indicates that we can create a schedule for both of these cores and we do not need to perform any further analysis. However, Core3 has a utilization of 86.0%, but the utilization bound for 4 registers is 75.7%. Therefore, we have failed the utilization-based schedulability test, though a schedule might still exist.

3.5 Response-time analysis

Response-time analysis [1] is another method for analyzing whether a process set (in our case, register set) can be scheduled. However, in addition to testing the schedulability of a set of registers, it also provides the worst case response time for each register. We calculate the response of a register using Equation 2, where R_i is the response time for register i, C_i is the computation time of register i, and I_i is the maximum interference that register i can experience in any time interval $[t,t+R_i)$. The interference of a register is the amount of time that a process must wait while other higher priority processes execute.

A register set is schedulable if all registers in the set have a response time less than or equal to their age constraint. From Table 2, we can see that the registers of all three cores will meet their register age constraints. Therefore, it is possible to create a pre-fetching schedule for all three cores. It is interesting to note that although the utilization-based schedulability test failed for Core3, response time analysis indicates that all of the registers can be pre-fetched. We refer the reader to [1] for further details on response-time analysis.

Equation 2: Response time analysis.

$$R_i = C_i + I_i$$

3.6 Sporadic register writes

We now consider the impact of writes to core registers. Writes come at unknown intervals, and a write ties up the core's internal bus and thus delays pre-fetches until done. We can therefore view a register write as a high priority sporadic process. We can attribute a maximum rate at which write commands will be sent to the core. We will also introduce a deadline for a write. The deadline of a write is similar to the access-time for a register being pre-fetched. This deadline indicates that when a write occurs, it must be completed within the specified number of cycles.

In order to analyze how a register write will impact this scheduling, we can create a dummy register, WR, in our register set. The age of the WR register will be the period that corresponds to the maximum rate at which a write will occur. WR's access-time will be equal to its deadline. We can now analyze the register set to determine if a pre-fetching schedule exists for it. This analysis will provide us with an analysis of the worst case scenario in which a write will occur once every period.

3.7 Deadline monotonic priority assignment

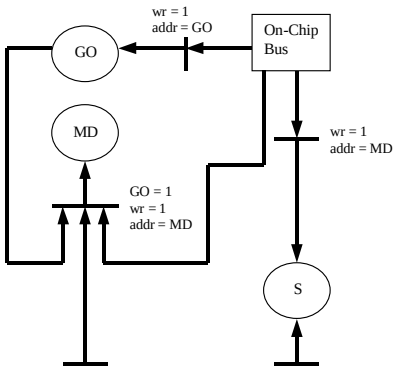
Up to this point, we have been interested mainly in a static schedule of the register set. However, because writes are sporadic, we must provide some dynamic mechanism for handling them. Thus, a dynamic scheduling technique should be used because we cannot accurately predict these writes. Therefore, we can use a more advanced priority assignment scheme, deadline monotonic priority assignment [1]. Deadline monotonic priority assignment assigns a priority to each process (register) based upon its deadline (access-time), where a smaller access-time corresponds to a higher priority. We can still incorporate rate monotonic priority assignment in order to assign priorities to registers with equal access-times. Deadline monotonic priority assignment is known to be optimal in the sense that if a process set can be scheduled by a priority scheme, then it can be scheduled by deadline monotonic priority assignment.

For example, in order to accommodate writes to the BIAS register in Core3, we can add the BIAS register to the pre-fetching algorithm. The deadline for the BIAS register will be such that we can ensure that writes will always have the highest priority when we use the deadline monotonic priority assignment. Using this priority assignment mechanism, the priority of the registers from highest to lowest would be BIAS, STAT, RES, A, and B.

4. Extension for update dependencies

In some cases, a core designer may be able to provide us more information regarding when the core's internal registers

Figure 2: ADJUST Register Dependencies



get updated -- in particular, update dependencies among registers (e.g., if register A is updated externally, then register B will be updated 1 cycle later). Using this information, we can design a schedule that performs fewer pre-fetches to satisfy given constraints, and thus can yield advantages of being able to handle more complex problems, or of using less power.

4.1 General register attributes

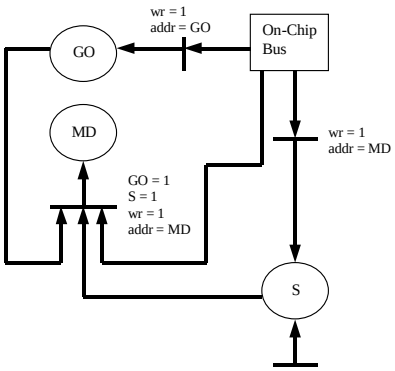
We need a method for capturing the information a designer provides regarding register updates. We divide this information into two kinds: attributes, and update dependencies.

Attributes are used to categorize registers based on how they are used. In [6], we defined 4 attributes: *update-type, access-type, notification-type,* and *structure-type.* The update-type gives information on when the register is updated. It can be static-update (updated regularly), volatile-update (updated at unpredictable times), or induced-update (updated as a result of some other observable event). Access-type describes what kind of accesses the system bus may perform. It is either read-only, write-only, or read/write access. Notification-type indicates how the register informs the system bus of an update. It can be interrupt-notification, register-based flag notification, output flag notification, or no notification. Finally, the structure type indicates how data is arranged. It is one of singly-structured, queue-structured, or block-structured.

Update dependencies provide further details on when the register gets updated as a result of other updates (inducements). There are two kinds of update dependencies:

- Internal dependencies -- dependencies between registers must be accurately described. Dependencies between registers affect both the operation of the core and the time at which registers are updated. Therefore, these dependencies are extremely important in providing an accurate model of a core's behavior.
- External dependencies -- Updates to registers via reads and writes over the OCB also need to be included in our model. This information is important because reads and writes can directly update registers or trigger updates to other registers, e.g., a write to a control register of a CODEC core will trigger an event that will update the output data register. Likewise, updates from external ports to internal core registers must also be present in our model. These events occur at random intervals and cannot be directly monitored by a bus wrapper and are therefore needed to provide a complete model of a core.

Figure 3: Refined ADJUST Register Dependencies



We needed to create a model to capture the above information. After analyzing many possible models to describe both internal and external update dependencies, we concluded that a Petri Net model best fits our requirements.

4.2 Petri Net model construction

As in all Petri Net models, we have places, arcs, and transitions. In our model, a place represents data storage, i.e., a register, or any bus that the bus wrapper can monitor. In this model, a *bus place* will generate tokens that will be outputted over all outgoing arcs and consumed by data storage places whenever an appropriate transition is fired. A transition represents an update dependency between either the bus and a register or between two registers. Transitions may be labeled with conditions that represent some requirement on the data coming into a transition. However, in many cases, a register may be updated from some external source, i.e., the register's update-type is volatile. Therefore, we need a mechanism to describe such updates. We will use a transition without incoming arcs and without an associated condition to represent this behavior. We will refer to such a transition as a random transition. Given random transitions, tokens can also be generated by external sources that cannot be directly monitored by the bus wrapper. Thus, our model provides a complete description of the core's internal register dependencies without providing all details of the core's internal behavior.

We implemented three core examples to analyze our update dependency model and pre-fetching technique. In order to demonstrate the usefulness of our model we will describe one of the cores we implemented, which we will refer to as ADJUST, and elaborate on this example throughout the paper. ADJUST contains three registers GO, MD, and S. First, we annotate each register with the general register attributes described earlier. The GO register has the attributes of static-update, write access, no notification, and singly-structured. The MD register has the attributes of volatile-update, read/write access, no notification, and singly-structured. Finally, the S register has the attributes of volatile-update, read-only access, no notification, and singly-structured. Next, we constructed the Petri Net for ADJUST.

Figure 2 shows the register update dependency model for ADJUST. From this model we can see how each register is updated. GO is updated whenever a write request for GO is initiated on the OCB. S is updated randomly by some external event that is unknown to the pre-fetch unit. MD is updated when GO is equal to 1, a write request for MD is initiated on the OCB, and some external event occurs. Therefore, we now have a complete model of the ADJUST core that can be used to create a pre-fetching algorithm.

Using the current model of ADJUST, we need three pre-fetch registers in the bus wrapper, namely, GO', MD', and S'. GO' would be updated whenever a write to GO was initiated over the OCB. For MD and S, we need some method of refreshing the pre-fetch registers to keep them as up-to-date as possible. We will later discuss the heuristics for updating registers with incoming random transitions. However, we further know that pre-fetching the MD register would not be required until a write to MD was made over the OCB and the GO register was equal to 1. This simple interpretation of the model will reduce the power consumed by the pre-fetch unit by not pre-fetching MD if it is not needed.

4.3 Model refinement for dependencies

Further refinement of our register update dependency model can be made to eliminate some random transitions. Although the model of a particular core may have many random transitions, there may exist some relationships between the registers with random transitions. If two registers are both updated by the same external event, it is possible that a relationship may exist between the registers.

For example, in a typical CODEC core, we would find a data register and a status register. When the data register is updated, the status register is also updated to indicate the operation has completed. Although both registers are updated at random times, we know that if the status register indicates completion, then the data register has been updated.

We can thus eliminate one random transition by replacing the random transition with a transition having an incoming arc from the related register and assigning an appropriate condition to this transition. Thus, we have successfully refined our model to eliminate a random transition. The goal of this refinement is to eliminate as many random transitions as possible, but it is important to note that it is not possible to eliminate all random transitions. Therefore, we still need a method for refreshing the contents of registers with incoming random transitions.

Figure 3 shows a refined register update dependency model for the ADJUST core. In this new model, we have eliminated one random transition by replacing it with a transition that will fire if S is equal to 1. Hence, we need to modify our pre-fetching algorithm to accompany this change. We now know that we only need to pre-fetch MD if S is equal to 1, GO is equal to 1, and a write to MD was initiated over the OCB. This refinement further simplifies our pre-fetching algorithm and will again reduce power consumption.

4.4 Pre-fetch scheduling

Given an update dependency model of a core, we need to construct a schedule to pre-fetch the core's registers into the bus wrapper's pre-fetch registers. Figure 4 describes our update dependency model pre-fetching heuristic using pseudo-code. Our heuristic uses the update dependency model in conjunction with our real-time pre-fetching to create a schedule for pre-fetching the core's registers. The following description will further elaborate on the heuristic.

Figure 4: General register model pre-fetching heuristic used to implement PFU.

```
create a heap sorted by deadline-monotonic priority
if registers have equal priorities {
    sort registers by rate-monotonic priority based on
      current register age
}
add all registers with incoming random transitions to heap

create a list for update arcs

while(1) {

   if write request detected {
      copy data into local pre-fetch register
      add all outgoing arcs to list of update arcs
      set write register's access-time to 0
      add write register to heap
   }

   if read request detected {
      add all outgoing arcs to list of update arcs
   }

   // pre-fetch registers that were most recently updated
   if register at front of heap has access-time = 0 {
      pre-fetch register
      reset register's access-time to initial value
      remove register from heap
      add outgoing arcs to list of update arcs
   }
   // register has incoming random transition and current
   // age has reached zero
   else if register at front of heap has current age = 0 {
      pre-fetch register
      remove register from the heap
      set current age to register age constraint
      add register to heap
      add outgoing arcs to list of update arcs
   }

   while list of update arcs is not empty {
      remove arc from the head of the list
      // analyze transition connected to arc
      if transition fires {
            set corresponding register's access-time to 0
            add register to the heap
      }
   }
}
```

In order to implement our pre-fetching heuristic, we will need two data structures. The first data structure needed is a pre-fetch register heap, or priority queue, used to store the registers that need to be pre-fetched. Secondly, we need a list of update arcs that must be analyzed after a register is pre-fetched or a read or write request is detected on the OCB. Using these data structures, we will next describe how the pre-fetch unit will be designed.

The first step in our pre-fetching heuristic is to add all registers with incoming random transitions to the pre-fetch register heap. These registers will always remain in the heap because they will need to be repeatedly pre-fetched in order to satisfy their register age constraints.

Next, our pre-fetch heuristic needs to respond to read and write requests on the OCB. In the event of a read request, the pre-fetch unit will add any outgoing arcs to the list of arcs

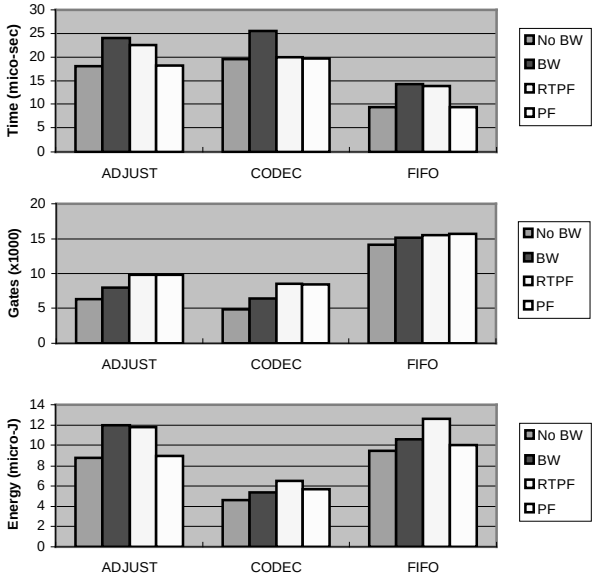
needed to be analyzed. As described in our real-time pre-fetching work, a write is treated as another register with special age and access-time constraints, i.e., the register age constraint is 0 and the access-time constraint is initially set to infinity. Because the core internal bus may be currently in use performing a pre-fetch, we use this mechanism to eliminate any contention. As described below, by setting the access-time constraint on the write register to 0, we will ensure that the write will be the next action performed. Therefore, a write request will be handled by first copying the data into the corresponding pre-fetch register, setting the access-time constraint to 0, and adding the write register to the pre-fetch register heap. In addition, any outgoing arcs will be added to the list of update arcs.

We will use our real-time pre-fetching to pre-fetch registers according to their priorities as assigned by the deadline monotonic priority assignment. When two registers have the same priority assigned by this mechanism, we will use the priority assigned by the rate-monotonic priority assignment to schedule the pre-fetching. According to this heuristic, registers with an access-time constraint of 0 will be pre-fetched first. That means that all write requests and, as we will describe later, all registers that have been updated will be pre-fetched first. Note that writes will still take highest priority because their register age constraint is 0. If no write requests or registers without incoming random transitions need to be pre-fetched, our pre-fetching heuristic will next schedule registers with incoming random transitions according to their rate-monotonic priority assignment. Therefore, our pre-fetch register heap will be sorted first by deadline-monotonic priority assignment and further by rate-monotonic priority assignment.

After each register pre-fetch is made or a read or write request is detected on the OCB, we need to analyze all arcs in the update arc list. If any transition fires, the outgoing arcs of this transition will be added to the list. If a token reaches another place, we set the corresponding register's access-time to 0 and add it to the heap, thus ensuring that this register is pre-fetched as soon as possible.

In order to better understand this pre-fetching heuristic, we will look at the ADJUST core. In ADJUST, we have one random transition which is connected to the S register. We noticed that in our design, on average, we only needed to read the contents of S every 6 cycles. Therefore, we set the register age constraint to 6 cycles, and the register access-time constraint to 2, indicating that the register S must be pre-fetched every 6 cycles. For MD, both the register age and access-time constraints are 2 cycles. GO, however, has neither an age constraint nor an access-time constraint because it is a write-only register. Note that even though GO is a write-only register a copy must be maintained in the bus wrapper, as it is needed in order to analyze the update dependencies. Our pre-fetching algorithm will monitor the OCB. If a write to the GO register is made, the data will be copied into GO', and the write register access-time will be set to 0. On a write to the MD register, the access-time of S will be set to 0. Also if GO is equal to 1 and S is equal to 1, then set the access-time for MD to 0. Finally, we will use the scheduling above to pre-fetch the registers when needed and perform write operations.

Figure 5: Performance (top), size (middle), and energy (bottom) comparisons



5. Experiments

In addition to implementing the ADJUST core as described above, we implemented two additional examples in order to evaluate the impact on performance, size, and energy using our real-time pre-fetching and our Petri Net based register dependency pre-fetching approaches. The *CODEC* core contains three registers DIN, DOUT, and STAT. This core behaves like a simple compressor/decompressor, whereby the input data is modified via some arbitrary translation, after which the STAT register is updated to reflect completion. The *FIFO* core contains two registers DATA and STAT. This core represents a simple FIFO that has data stored in DATA and the current number of items in the FIFO stored in STAT.

We modeled the cores as synthesizable register-transfer VHDL models, requiring 215, 204 and 253 lines of code, respectively -- note that we intentionally did not describe internal behavior of the cores, but rather just the register-access related behavior, so we could see the impacts of pre-fetching most clearly. We used Synopsys Design Compiler for synthesis as well as Synopsys power analysis tools.

Figure 5 summarizes the results for the three cores. For each core, we examined four possible bus wrapper configurations: no bus wrapper (*no BW)*, a bus wrapper without pre-fetching (*BW*), a bus wrapper with real-time pre-fetching (*RTPF*), and a bus wrapper with our update dependency pre-fetching model (*PF*).

The first chart in Figure 5 summarizes performance results. In all three cores, the use of our update dependency pre-fetching method almost entirely eliminated the performance penalty associated with the bus wrapper. There was still a slight overhead caused by starting the pre-fetch unit. Using our real-time pre-fetching heuristic, we can see that although there is a performance improvement when compared to a bus wrapper without pre-fetching, it did not perform as well as our update dependency model. In FIFO, we only see a very small

performance improvement using real-time pre-fetching. This small improvement is due to fact that the DATA register in FIFO cannot be pre-fetched using this approach. If we were to pre-fetch DATA using real-time pre-fetching, we would empty the FIFO and lose data. Furthermore, without any pre-fetching, we can see a significant performance penalty.

The second chart in Figure 5 summarizes size results. As expected, the size of the cores increased when a bus wrapper was added, and further increased when either form of pre-fetching was added to the bus wrapper. The average increase in size caused by adding the update dependency pre-fetching technique to the bus wrapper was only 1.5K gates. In comparison, real-time pre-fetching resulted in an average increase of 1.4K gates. It is interesting to note why the two approaches, although quite different, resulted in approximately the same size increase. Using real-time pre-fetching, we increase the design complexity due to the need to keep track of current register ages. However, using our extended approach, complexity increases due to added logic needed to analyze update dependencies. In both cases, the size increase is relatively small when compared to a typical core size of 10K to 20K gates.

The third chart in Figure 5 summarizes energy consumption for our test vectors. In all three cores, there is an overall increase in energy consumption when a bus wrapper is added to the core. However, the addition of pre-fetching to the bus wrappers does not always strictly increase or decrease energy consumption. In fact, we can see that in AJDUST and FIFO, there is a decrease in energy consumption when our update dependency pre-fetching is added to the bus wrapper, but in CODEC, there is an increase. On the other hand, real-time pre-fetching increases energy consumption in CODEC and FIFO, and decreases energy consumption in ADJUST. As expected, when compared to the core without a bus wrapper, both pre-fetching techniques result in an increase in energy consumption.

More importantly, if we compare the results of our real-time pre-fetching to our update dependency pre-fetching, we notice that the update dependency pre-fetching results in significantly less energy consumption. This is easily explained by the fact that this approach only pre-fetches registers when they have been updated whereas our real-time pre-fetching will pre-fetch registers more often to keep them as up-to-date as possible. Therefore, by eliminating the need to pre-fetch all registers within their register age constraints, we can reduce energy consumption.

6. Conclusions

Pre-fetching can improve the performance of cores that have been designed for portability using the VSIA standard, and this performance enhancement improves the marketability of those cores. In this paper, we have provided the first definition of the pre-fetching problem, involving the description of registers and their access constraints. We have provided a powerful solution to this problem by mapping the problem to the real-time process-scheduling domain, and then applying heuristics and analysis techniques from that domain. We have further provided a general register update dependency model that we used to construct a more efficient pre-fetching schedule, in conjunction with our real-time pre-fetching. We demonstrated the effectiveness of these solutions through several experiments, showing good performance improvements with acceptable size and energy increases. Furthermore, we demonstrated that using our update dependency model, we were able to better pre-fetch registers when compared to our real-time pre-fetching methodology. The two approaches are thus complementary -- the real-time approach can be used when only register constraints are provided, while the model-based approach of this paper can be used when register update information is also provided. Future work includes: (1) restricting the registers that can appear in a bus wrapper to a number less than the number of registers being pre-fetched, resulting in new cache-like issues such as mapping and replacement issues, and (2) developing pre-fetching heuristics that optimize a given cost function of performance, power and size.

7. Acknowledgements

This work was supported by the National Science Foundation (CCR-9811164) and a Design Automation Conference Graduate Scholarship.

8. References

[1] A. Burns and A. Wellings. Real-Time Systems and Programming Languages, Addison-Wesley, 1997.

[2] P. Chou, R.B. Ortega, G. Borriello. Interface Co-Synthesis Techniques for Embedded Systems. International Conference on Computer-Aided Design, pp. 280-287, 1995.

[3] M. Gasteier, M. Glesner. Bus-Based Communication Synthesis on System-Level. ACM Transactions on Design Automation of Electronic Systems, Vol 4, No 1, 1999.

[4] Inventra core library, Mentor Graphics, http://www.mentorg.com/inventra/.

[5] C. Lui and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. JACM, pp. 46-61, 1973.

[6] R. Lysecky, F. Vahid, T. Givargis, and R. Patel. Pre-fetching for Improved Core Interfacing. International Symposium on System Synthesis, 1999.

[7] V. Madisetti, L. Shen. Interface Design for Core-Based Systems. IEEE Design & Test of Computers, pp. 42-51, 1997.

[8] J. Rowson and A. Sangiovanni-Vincentelli, Interface-Based Design, Design Automation Conference, 1997.

[9] Semiconductor Industry Association Roadmap 1997, http://notes.sematech.org/ntrs/PublNTRS.nsf.

[10] F. Vahid and T. Givargis, Incorporating Cores into System Level Specification, International Symposium on System Synthesis, pp. 43--48, 1998.

[11] S. Vercauteren, B. Lin, H. De Man. Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications. Design Automation Conference, pp. 547-551, 1996.

[12] Virtual Socket Interface Association, On-Chip Bus Development Working Group, Specification 1 Version 1.0 (OCB 1 1.0), http://www.vsi.org, 1998.