Prefetching for Improved Bus Wrapper Performance in Cores

ROMAN LYSECKY
University of California, Riverside
and
FRANK VAHID
University of California, Riverside, and University of California, Irvine

Reuse of cores can reduce design time for systems-on-a-chip. Such reuse is dependent on being able to easily interface a core to any bus. To enable such interfacing, many propose separating a core's interface from its internals by using a bus wrapper. However, this separation can lead to a performance penalty when reading a core's internal registers. In this paper, we introduce prefetching, which is analogous to caching, as a technique to reduce or eliminate this performance penalty, involving a tradeoff with power and size. We describe the prefetching technique, classify different types of registers, describe our initial prefetching architectures and heuristics for certain classes of registers, and highlight experiments demonstrating the performance improvements and size/power tradeoffs. We further introduce a technique for automatically designing a prefetch unit that satisfies user-imposed register-access constraints. The technique benefits from mapping the prefetching 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 prefetch schedules.

Categories and Subject Descriptors: B.4.3 [**Input/Output and Data Communications**]: Interconnections (subsystems)—*Interfaces*; B.5.0 [**Register-Transfer-Level Implementation**]: General; C.5.4 [**Computer System Implementation**]: VLSI System

General Terms: Design, Performance

Additional Key Words and Phrases: Bus wrapper, cores, design reuse, intellectual property, interfacing, on-chip bus, PVCI, system-on-a-chip, VSIA

1. INTRODUCTION

Silicon capacity continues to increase faster than the ability for designers to use that silicon, resulting in the well-known productivity gap [Semiconductor Industry Association 1999]. Many people propose extensive reuse of pre-designed

This work was supported by the National Science Foundation under grant Nos. CCR-9811164 and CCR-9876006, and is part of the Dalton project at the University of California, Riverside (http://www.cs.ucr.edu/~dalton).
Authors' addresses: Roman Lysecky and Frank Vahid, Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA 92521; e-mail: rlysecky@cs.ucr.edu; vahid@cs.ucr.edu.

intellectual property *cores* to reduce this gap [Virtual Socket Interface Association 1997a], where typical cores include microprocessors, microcontrollers, digital signal processors, bus interfaces, and numerous peripheral components. In response, several commercial libraries of cores have evolved in recent years, e.g., Mentor Graphics [n.d.]. Soft cores come in the form of synthesizable code, while hard cores come in the form of technology-specific layouts.

A key aspect of a core's marketability, soft or hard, is its ability to be easily integrated into a system-on-a-chip (SOC). This is especially true for peripheral cores, since they must be usable in many systems in order to be profitable for the core designer. Unfortunately, standardizing on one or two on-chip SOC buses, which certainly would ease integration, does not appear to be possible because of the diversity of constraints present in embedded systems, as recognized for example by the Virtual Socket Interface Alliance (VSIA) [Virtual Socket Interface Association 1998]. Thus, to achieve such ease of integration, many have proposed designing cores with their interface behavior implemented in a bus wrapper, separated from the core's internal behavior [Rowson and Sangiovanni-Vincentelli 1997; Vahid and Tauro 1997; Virtual Socket Interface Association 1998]. This separation means that changes necessary to adapt a core to a particular bus would be limited to the bus wrapper.

In Section 3 of this paper, we analyze the impact of using a bus wrapper on the design metrics of performance, power, and size. We analyze the impact on those metrics of using the VSIA interface between a bus wrapper and a peripheral core's internals versus using a customized interface. We show that the modularity achieved with a bus wrapper often comes with a performance penalty. For example, reading a core's internal register from the bus may require extra cycles to first read the register data into the bus wrapper before the data can be output to the bus.

We therefore propose a solution to this performance penalty, called prefetching. Briefly, prefetching is analogous to caching, wherein we store local copies of registers inside a bus wrapper so that a register read results in outputting of this local copy, thus eliminating extra cycles on a read. As with caching, prefetching schemes must strive to maximize the hit ratio. Prefetching requires appropriate bus wrapper architecture design as well as heuristics that maximize the hit/miss ratio. In Section 4, we describe the idea of prefetching, classify common core registers, describe prefetching architectures and simple heuristics for common classes, and provide results demonstrating the impact on performance, power, and size. In Section 5, we describe a technique for automatically designing a prefetch unit (PFU) that meets user-provided constraints on maximum allowable latency and data-age for each register. The key design problem is to schedule the prefetches over the core's internal bus such that the constraints are met. We observe that the prefetching problem could be mapped to the extensively researched problem of real-time process scheduling, and thus we apply powerful heuristics and analysis techniques for that problem to solve the prefetching problem. In Section 6, 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 prefetch 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 prefetches based on that model.

2. RELATED WORK

2.1 Interfacing

Much work has been done on interfacing with cores, but to our knowledge none of the literature includes the idea of prefetching. The bulk of interfacing work has dealt with the automatic synthesis of logic to interface with a bus, synthesis of the bus itself, or defining a standard bus protocol. The VSIA has proposed the Virtual Core Interface (VCI), which defines a bus protocol for connecting a bus wrapper to the core internals. Motorola has also released its IP Interface (IPI) that provides a standard similar to the VCI. These various techniques and standards all strive to reduce system design time when designs become too large for the available software tools to handle efficiently. We now discuss these approaches.

2.1.1 *Automatic Synthesis of Bus Interface Logic.* Chou et al. [1995] proposed a method for automatically generating the software and hardware needed for a specified interface. Their approach starts with defining the bandwidth and timing constraints of the desired interface. You must also describe certain aspects of the processor, such as I/O pins available, interrupt handling mechanisms, and the system bus. All of these aspects must be described in great detail. From this information, they generated the hardware and software needed to interface between the processors and cores in the system. Their technique will enable a designer to change the interface requirements of the system with relative ease.

Madisetti and Shen [1997] proposed a different technique for generating interface logic. Their approach starts in the early phases of SOC design when the design specification is being developed. As part of design specification, they proposed using a specialized language that can define both functional and timing characteristics to describe the interface. Using this approach, they argued that design flaws can be easily found at the early stages of design and corrected. The system designer can then use the interface description to automatically synthesize the necessary interface logic.

2.1.2 *Bus Synthesis.* Instead of generating the interface logic for a bus, Gasteier and Glesner proposed an approach to automatically generate a communication topology consisting of possibly several buses. Their approach starts with a set of processes that communicate via simple send and receive commands. Then, from a simulation phase, data is gathered relating to the sends and receives from each device. This information is used to generate a cost-effective communication topology using buses without arbitration. Although this approach allows for efficiently choosing a bus structure for a set of processes, it does not attempt to increase the retargetability of cores.

2.1.3 *Standard Bus Protocols.* Another method for decreasing design time is to use a standard bus protocol, such as that proposed by Vercauteren et al.

[1996]. They proposed a cosynthesis technique that utilizes a parameterizable architecture with a defined communication protocol. By defining a standard bus protocol, IP cores can be designed to use this protocol and thus facilitate an easy integration into the system architecture. Coupled with other hardware and software strategies, Vercauteren et al.'s proposed technique can significantly reduce design time. However, as previously stated, the success of a standard protocol requires that all core designers adhere to this bus. Due to the multitude of proprietary bus protocols, the VSIA determined that a standard bus protocol would most likely not be adopted.

2.1.4 *VSIA's Virtual Component Interface.* In an effort to promote the use of bus wrappers, the VSIA has proposed a standard bus protocol that interfaces between a bus wrapper and the core internals. The goal of this proposal is to facilitate quick retargeting of a core from one system to another. If a standard internal bus is used, a core can be retargeted to any bus simply by providing an appropriate bus wrapper. The main selling point of this technique is that no changes need to be made to the core itself. The VSIA has currently proposed two version of its VCI, namely the Basic Virtual Component Interface (BVCI), and the Peripheral Virtual Component Interface (PVCI) [Virtual Socket Interface Association 1997b]. VSIA is also planning to provide a more powerful protocol designed for high-performance systems. The upcoming protocol will be the Advanced Virtual Component Interface (AVCI).

The PVCI is the simplest protocol defined by VSIA. It is a simple point-to-point protocol consisting of two unidirectional buses with a simple two-way handshake. The PVCI is designed to be used with peripheral on-chip buses and for point-to-point connections between cores. One of the key aspects of the PVCI is that it is easy to implement due to its simple communication scheme. The BVCI is a superset of the PVCI and is targeted at on-chip buses and point-to-point connections between cores. The BVCI adds features such as DMA transfers, interrupt handling, etc. The BVCI was designed with the intent to provide a bus protocol that can handle most on-chip bus interfacing needs.

2.1.5 *Motorola's IP Interface.* Since the release of the VSIA's Virtual Component Interface, Motorola has also announced its own interfacing standard. Like the VCI, Motorola's IPI [Motorola 1999] provides a standard bus protocol for interfacing between cores. However, Motorola uses the concept of Bus Interface Gasket instead of bus wrapper. Similar in nature to the VSIA's proposal, Motorola also defines varying levels to its protocol. However, the IPI does so by defining different communication *lines*. For example, the SkyBlue-Line defines the interface between a peripheral bus interface gasket and a core. In fact, the SkyBlue-Line defined by the IPI is very much similar to the PVCI. In order to handle more complex interfacing needs, the IPI defines additional lines, e.g., the DarkBlue-Line defines a DMA interface between a core and a gasket.

2.1.6 *Rapid Prototyping.* Many groups have focused on rapid prototyping approaches. There are different approaches that have been taken in this aspect. Philips has developed the Velocity platform, which is a large predesigned SOC [Philips Semiconductors 1999]. The system includes a processor, memory, cache, and common peripheral components. Users of the system need only deactivate the components that are not being used and add any custom logic to a large Field Programmable Gate Array (FPGA).

Another approach uses a high-level language to speed to design flow. CoWare produces a product called *N2C*, which stands for "napkin-to-chip" [CoWare 1999]. This product is designed such that systems designers can start with a detailed description of the system using C/C++. This specification is then used throughout the design flow to verify the functionality of the design at each step.

Other systems are targeted at specific applications such as Internet connectivity. Metaflow has designed the Implosion SOC Platform, which combines an SOC with on-chip Internet capabilities [Metaflow 1999]. At the heart of the Implosion platform is an ARM processor. The system also comes with a myriad of common peripheral components, e.g., Ethernet, DMA controller, interrupt controller, etc. The goal of this system is to enable designers to rapidly create designs that can access information via the Internet.

Fleischmann et al. [1999], Clement et al. [1999], and Kuhn et al. [1997] have each proposed systems in which a high-level language (C, C++, or Java) is used to model and test a system in various stages of design. These techniques rely on the speed and efficiency of the high-level languages to decrease verification through a combination of codesign techniques, cosimulation, and cosynthesis.

2.2 Heterogeneous Systems

Some researchers have argued that a design paradigm shift caused by the availability of transistors on a single die will lead computer architectures in a new direction [Kozyrakis and Patterson 1998]. The flexibility provided by SOC designs has led way to heterogeneous systems composed of many different types of computing elements. V Meerbergen et al. [1998] and Rabaey et al. [1997] have presented how these systems can be used efficiently. Their respective techniques provide design methodologies aimed at increasing the productivity of designers using these systems.

2.3 Prefetching

Prefetching has been used in the area of microprocessor and memory system design. In fact, prefetching has been studied in great detail in these areas. However, to our knowledge no work on prefetching has dealt with peripheral cores. Prefetching is used in memory systems to reduce the miss rate of caches Patterson and Hennessy [1990]. When data is read or written, the address being accessed is checked to see if the desired data block is present in the cache. If the data is not found in the cache, the appropriate block is fetched from memory. The block is then placed into the cache by replacing a block in the cache according to the replacement policy. Prefetching works by reading a block of memory either into the cache itself or into a prefetch buffer by predicting which block will be needed shortly based upon the current address being accessed.

Another type of prefetching in this area is compiler-controlled prefetching. In this prefetching technique, a compiler inserts prefetch instructions that will

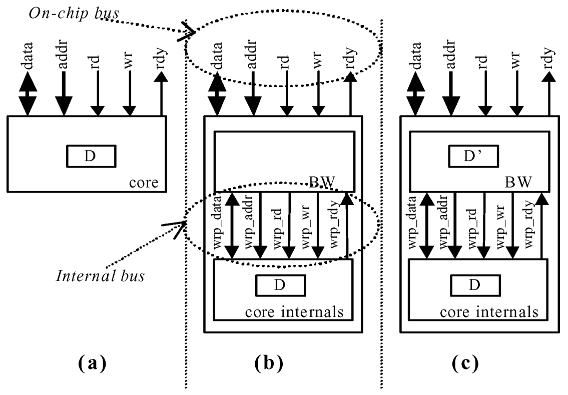

Fig. 1. Core interface options: (a) no BW, (b) BW without prefetching, (c) BW with prefetching.

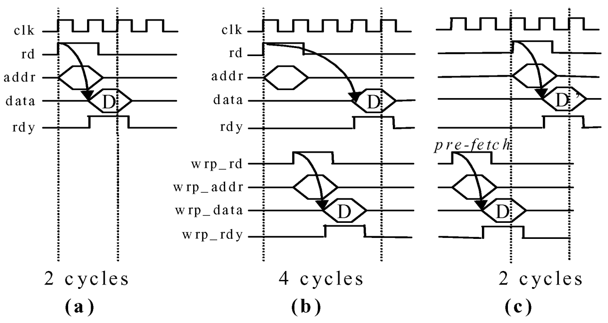

Fig. 2. Interface option timing: (a) no BW, (b) BW without prefetching, (c) BW with prefetching.

load data into the cache while not blocking the processor from executing other instructions. The goal of compiler-controlled prefetching is to overlap the execution of the processor with the fetching of data, thus reducing the miss rate.

3. BUS WRAPPERS

3.1 Overview

Separating a core's interface behavior and internal behavior can lead to performance penalties. For example, consider the core architectures shown in Figures 1(a), 1(b) and 1(c), showing a core with no bus wrapper, a core with a bus wrapper (BW) but without prefetching, and a core with a BW with prefetching, respectively. The latter two architectures are similar to that being proposed by the VSIA. The BW interfaces with the system bus, whose protocol may be arbitrarily complex, include a variety of features like arbitration. The BW also interfaces with the core internals, over a core internal bus; this bus is typically extremely simple, implementing a straightforward data transfer. It is this internal bus that the VSI On-Chip Bus group is standardizing. Without a BW, a read of a core's internal register from the on-chip bus may take as little as two cycles, as shown in Figure 2(a). With a BW, the read of a core's internal register may require four cycles, two from the internal module to the BW, and

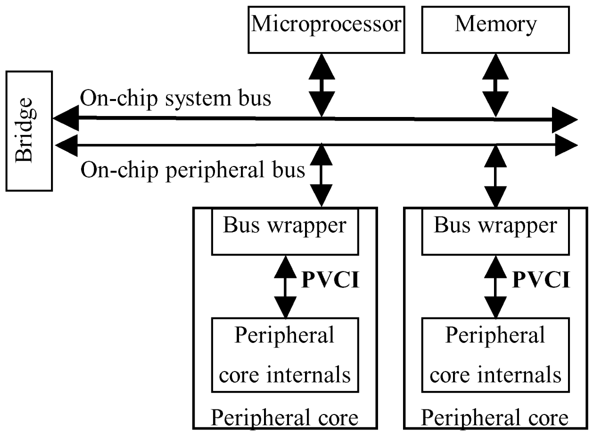

Fig. 3. PVCI's location in a system-on-a-chip.

two from the BW to the bus. Thus, a read may require extra cycles compared with a core whose interface and internal behavior are combined.

However, a core with its interface behavior separated into a bus wrapper is believed to be much easier to retarget to different buses than a core whose interface behavior is integrated with its internal behavior. By standardizing the interface between the core's internals and the bus wrapper, retargeting of a core may become easier.

3.2 PVCI

After deciding that a single on-chip bus standard was unlikely, the VSIA developed the VCI [Virtual Socket Interface Association 1997b]. The VCI is a proposed standard interface between a core's internals and a core's bus wrapper, as illustrated in Figure 3. Retargeting a core using VCI will involve roughly the same changes to the bus wrapper, since the VCI ensures that the changes are limited to the wrapper and not the internals, and since a bus provider can even provide bus wrapper templates between the bus and the VCI. The VCI is a far simpler protocol than a typical bus protocol, since it is a point-to-point transfer protocol. In contrast, a bus protocol may involve more advanced features, such as arbitration, data multiplexing, pipelining, and so on. Thus, standardizing the VCI is far simpler than standardizing a bus protocol.

The PVCI is a simplified version of the VCI, specifically intended for peripherals. PVCI cores would reside on a lower-speed peripheral bus as shown in Figure 3, and thus would not need some of the high-speed features of the VCI, e.g., packet chaining. The general structure of the PVCI is shown in Figure 4. It consists of two unidirectional buses. One bus leads from the wrapper to the internals. The wrapper sets the *read* line to indicate a read or a write, and sets the *address* lines with a valid address. For a write, it also sets the *wdata* lines. It asserts the *val* line to actually initiate the read or write. The wrapper must hold all these lines constant until the internals assert the *ack* line. For a write, this means that the internals have captured the write data. For a read, this means that the internals have put the read data on the *rdata* bus. The transaction

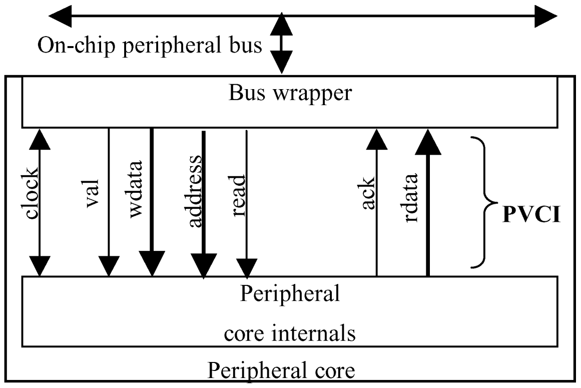

Fig. 4. PVCI's general structure.

is completed on the next rising clock edge. A fast internals module can keep *ack* asserted continuously to provide for fast transfers, similar in spirit to the synchronous wait protocol [Vahid and Givargis 1999].

3.3 Experiments with Bus Wrappers

We sought to evaluate the impact of a wrapper and of PVCI using a simple peripheral bus. We used a bus with a two-phase handshake protocol to ensure that the communication was as fast as possible for a given peripheral. As previously demonstrated, using a wrapper results in a two-cycle overhead per read as compared with an integrated core.

Figure 2(a) illustrates the timing of a read cycle of this peripheral bus for an integrated core. The peripheral bus master (in our case, the bridge) places an address on *addr* and then strobes *rd*. The peripheral responds by placing data on *data* and strobing *rdy* as early as once cycle after receiving the *rd* strobe. Thus, the total read cycle could be as little as two clock cycles.

Figure 2(b) illustrates the read cycle of the bus for a core using a bus wrapper. After the bus master places the address and strobes *rd*, the wrapper responds by translating this read request into a read request over the internal bus. This translation involves translating the address to one appropriate for the core and then placing that address on *wrp_addr*, and then asserting *wrp_read*. The core's internals respond by placing data on *wrp_data* and then asserting *wrp_rdy*. The wrapper receives the data, puts it on the peripheral bus, and strobes *rdy*.

A write cycle need not incur any performance overhead in the wrapper versions. When the bus master sets the addresses and strobes the appropriate ready line, the wrapper can respond immediately by capturing the data and strobing the ready line, just like an integrated core will do. The wrapper can then proceed to write the captured data to the core internals, while the bus master proceeds with other activities.

The example we evaluated was a simple version of a digital camera system, illustrated in Figure 5. The camera system consists of a (simplified) MIPS microprocessor, BIOS, and memory, all on a system bus, with a bridge from the system bus to a peripheral bus (ISA) having a CCD (charge-coupled device)

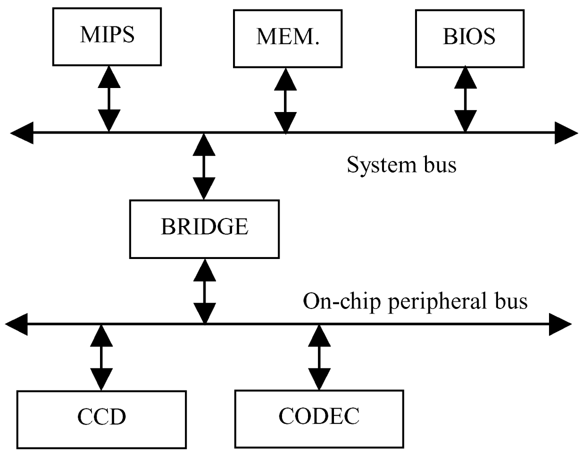

Fig. 5. Digital camera example system.

preprocessor and a simple CODEC (compressor/decompressor). The two-level bus structure is in accord with the hierarchical bus concept described in [Virtual Socket Interface Association 1997a]. The camera is written in register-transfer level synthesizable VHDL, and synthesizes to about 100,000 cells. We used the Synopsys Design Compiler as well as the Synopsys power analysis tools to evaluate different design metrics. Power and performance were measured for the processing of one frame.

We made changes to the CCD preprocessor and CODEC cores since they represent the peripherals on the peripheral bus. These cores are used heavily while processing a frame. We created three versions of the camera system:

(1) *Integrated*: The CCD preprocessor and CODEC cores were written with the interface behavior inlined into the internal behavior of the core. Thus, synthesis generates one entity for each core.
(2) *Non-PVCI wrapper*: The CCD preprocessor and CODEC cores were written with the interface behavior separated into a wrapper. Thus, synthesis generates two connected entities for the core. The interface between these two wrapper and internal entities consisted of a single bidirectional bus, a strobe control line and a read/write control line, and however many address lines were necessary to distinguish among internal registers.
(3) *PVCI wrapper*: Same as the previous version, except that the interface between the wrapper and internal entities was PVCI.

The non-PVCI wrapper version was created for another purpose, well before the PVCI standard was developed and with no knowledge that the version would be used in these experiments. Thus, its structure was developed to be as simple as possible.

Table I summarizes size, performance, and power results. Size is reported in equivalent NAND gates, time in nanoseconds, and power in milliwatts. The size overhead when using a bus-wrapper (non-PVCI) compared to the integrated version was roughly 1500 gates per core. This overhead comes from extra control

Table I. Comparison of Interface Versions Using a Custom Bus

Version	Ex.	Size of Wrapper	Size of Internals	Power for 1 frame	Total time (1 frame)	I/O Time (1 frame)
Integrated	CCD	0	34320	7.90	75175	7760
	CODEC	0	1926			
Non-PVCI Wrapper	CCD	1661	34556	8.11	79055	15520
	CODEC	1674	1904			
PVCI Wrapper	CCD	1439	33978	7.98	79055	15520
	CODEC	1434	1588			

and registers. In the integrated version, the core's internals includes control to interface to the peripheral bus. In the wrapper version, this control is replaced by control for interfacing to the wrapper, so the size of the core's internals stays the same. However, the wrapper now must implement control for interfacing to the internals, and for interfacing to the peripheral bus, representing overhead. The wrapper must also include registers whose contents are copied to/from the internals, representing additional overhead. The reason that the non-PVCI wrapper version shows more size overhead than the PVCI wrapper version is because the non-PVCI version used a single bus for transfers both two and from the core internals, whereas PVCI specifies two separate buses, resulting in less logic but more wires. Fifteen hundred gates of size overhead seems quite reasonable, given the continued increase of chips' gate capacities, and given that peripheral cores typically posses 20,000 gates or more [Mentor Graphics n.d.].

The system power overhead was only about 1%. The extra power comes from having to transfer items twice per access. On a write, an item must be transferred first from the bus to the wrapper, then from the wrapper to the internals. On a read, an item must be transferred first from the internals to the wrapper, then from the wrapper to the bus. However, the power consumed by the memory, system bus, and processor dominate, so the extra power due to the wrappers is very small—*even though* the CCD and CODEC are heavily used when processing a frame.

In Table I, we can see that there is a 100% increase in peripheral I/O access time when bus wrappers are employed. This overhead is due to the use of a wrapper, which would have occurred whether using PVCI or another wrapper. In our experiments, the CCD was accessed 256 times per image frame, while the CODEC was accessed a total of 128 times per frame. Because the MIPS processor executed approximately 5000 instructions per frame, the overall overhead of the bus wrappers amounts to approximately 5%.

One difference between the non-PVCI and PVCI interface that does not appear in the results is the number of wires internal to the core. The non-PVCI version uses a multiplexed bus, and has fewer signals (some PVCI signals were not shown), and thus would have fewer internal wires.

Noting that our CCD and CODEC cores are relatively small and have simple interfaces, it took us 6 designer hours, excluding synthesis and simulation time, to retarget a design from one wrapper to another, e.g., to convert the CCD's non-PVCI wrapper to a PVCI implementation. Synthesis time for the CCD and CODEC was approximately 1 hour. Simulation time for capturing one image frame was slightly over 10 hours and power analysis was an additional 5 hours. These times were obtained by synthesizing the models down to gates using Synopsys Design Compiler with medium mapping effort, using the *lsi_10k* library supplied by Synopsys, with no area or delay constraints specified. We used a dual 200-MHz Ultra Sparc II machine to perform both our synthesis and simulation. Synthesis and simulation times were relatively the same between the integrated bus implementations and those using a bus wrapper. We note that peripheral devices capable of DMA or burst mode I/O with interrupts will require more time to integrate into a system.

Although the use of bus wrappers improves the usefulness of a core by making it easier to retarget to varying systems, this reusability comes at a cost. Bus wrappers introduce both performance and power overhead, as we have demonstrated. In tightly constrained systems where peripheral access time is critical, this overhead is often infeasible. Ideally, the use of bus wrappers could allow for quick retargeting of a core while not degrading performance. In the next section, we present a technique called *prefetching* that effectively eliminates the performance overhead of bus wrappers.

4. BASIC PREFETCHING

4.1 Overview

Our focus is to minimize this performance penalty in order to maximize the usefulness of the core. We seek to do so in a manner transparent to both the developers of the core internal behavior as well as developers of the on-chip bus. Because of the continued exponential growth in chip capacity, we seek to gain performance by making the tradeoff of increased size, since size constraints continue to ease. However, we note that our approach increases the switching activity of the core, and thus we must also evaluate the increased power consumption and seek to minimize this increase.

We focus on peripheral cores, whose registers will be read by a microprocessor over an on-chip bus (perhaps via a bus bridge) with the idea being to minimize the read latency experienced by the microprocessor.

The basic technique that we propose is called *prefetching*. *Prefetching* is the technique of copying a core's internal register data into a prefetch register in a core's BW, so that when a read request from the bus occurs, the core can immediately output prefetched data without spending extra cycles to first get the data from the core's internal module. We use the terms *hit* and *miss* in a manner identical for caches; a hit means that the desired data is in a prefetch register, while a miss means that the data must first be fetched into a prefetch register before being output to the on-chip bus. For example, Figure 2(c) shows that prefetching a core's internal register D into a BW register D′ results in a system read again requiring only two cycles, rather than four.

4.2 Classification of Core Registers

We immediately recognized the need to classify common types of registers found in peripheral cores, since different types would require different prefetching approaches.

After examining cores, primarily from the Inventra library [Mentor Graphics n.d.], focusing on bus peripherals, serial communication, encryption, and compression/decompression, we defined a register classification scheme based on four attributes: update type, access type, notification type, and structure type:

(1) The *update type* of a register describes how the register's contents are modified. Possible types include:
   (a) A *static-update* register is updated by the system only, where the system is the device (or devices) that communicate with the core over the on-chip bus. An example of a static register is a configuration register. After the system updates the register, the register's content does not change until the system updates it again.
   (b) A *volatile-update* register is updated by a source other than the system (e.g., internally by the core or externally by the core's environment) at either a random or fixed rate. An example is an analog-to-digital converter, which samples external data, converts the data to digital, and stores the result in a register, at a fixed rate.
   (c) An *induced-update register* is updated as a direct result of another register within the core being updated. Thus, we associate this register with the inducing register. Typically, an induced register is one that provides status information.

(2) The *access type* of a register describes whether the system reads and/or writes the register, with possible types including: (a) *read-only access*, (b) *write-only access*, and (c) *read/write access*.

(3) The *notification type* describes how the system is made aware that a register has been updated, with possible types including:
   (a) An *interrupt notification* in which the core generates an interrupt when the register is updated.
   (b) A *register-based flag notification* in which the core sets a flag bit (where that bit may be part of another register).
   (c) An *output flag notification* in which the core has a specific output signal that is asserted when the register is updated.
   (d) *No notification* in which the system is not informed of updates and simply uses the most recent register data.

(4) The *structure type* of the register describes the actual storage capability of the register, with possible types including:
   (a) A *singly structured* register is accessed through some address and is internally implemented as one register.
   (b) A *queue-structured* register is a register that is accessed through some address but is internally implemented as a block of memory. A common example is a buffer register in a UART.
   (c) A *block-structured* register is a block of registers that can be accessed through consecutive addresses, such as a register file or a memory.

4.3 Commonly Occurring Register Types

For our first attempt at developing prefetching techniques for cores, we focused on the following three commonly occurring combinations of registers in cores:

(1) *Core1—configuration registers*: Many cores have configurable settings controlled by a set of configuration registers. A typical configuration register has the features of static update, read/write access, no notification, and singly structured. We refer to this example as *Core1*.

(2) *Core 2—task registers*: Many cores carry out a specific task from start to completion and have a combination of a data input register, a data output register, and a status register that indicates completion of the core's task. For example, a CODEC (compress/decompress) core typically has such a set of registers. We looked at how to prefetch the data output and status registers. The data output register has the following features: volatile-update at a random rate, read-only access, register-based flag notification with the flag stored in the status register, and singly structured. The status register has the following features: induced update by an update to the data output register, read-only access, no notification, and singly structured. Although the data input register will not be prefetched, its features are: volatile-update at a random rate, write-only access, no notification, and singly structured. We refer to this example as *Core2*.

(3) *Core3—input-buffer registers*: Many cores have a combination of a queue data buffer that receives data and a status register that indicates the number of bytes in the buffer. A common example of such a core is a UART. Features of the data buffer include: volatile-update at a random rate, read-only access, register-based flag notification stored in the status register, and queue-structured. The status register features include: induced-update by an update to the data register, read-only access, no notification, and singly structured. We refer to this example as *Core3*.

4.4 Prefetching Architectures and Heuristics

4.4.1 *Architecture.* In order to implement the prefetching for each of the above listed combinations of registers, we developed architectures for bus wrappers for each. Figure 6 illustrates the architecture for each of the three combinations respectively. Each BW architecture has three regions:

(1) *Controller*: The controller's main task is to interface with the on-chip bus. It thus handles reads and writes from and to the core's registers. For a write, the controller writes the data over the core internal bus to the core internal register. For a read, the controller outputs the appropriate prefetch register data onto the bus; for a hit, this outputting is done immediately, while for a miss, it is done only after forcing the prefetch unit to first read the data from the core internals.

(2) *Prefetch registers*: These registers are directly connected to the on-chip bus for fast output. Any output to the bus must pass through one of these registers.

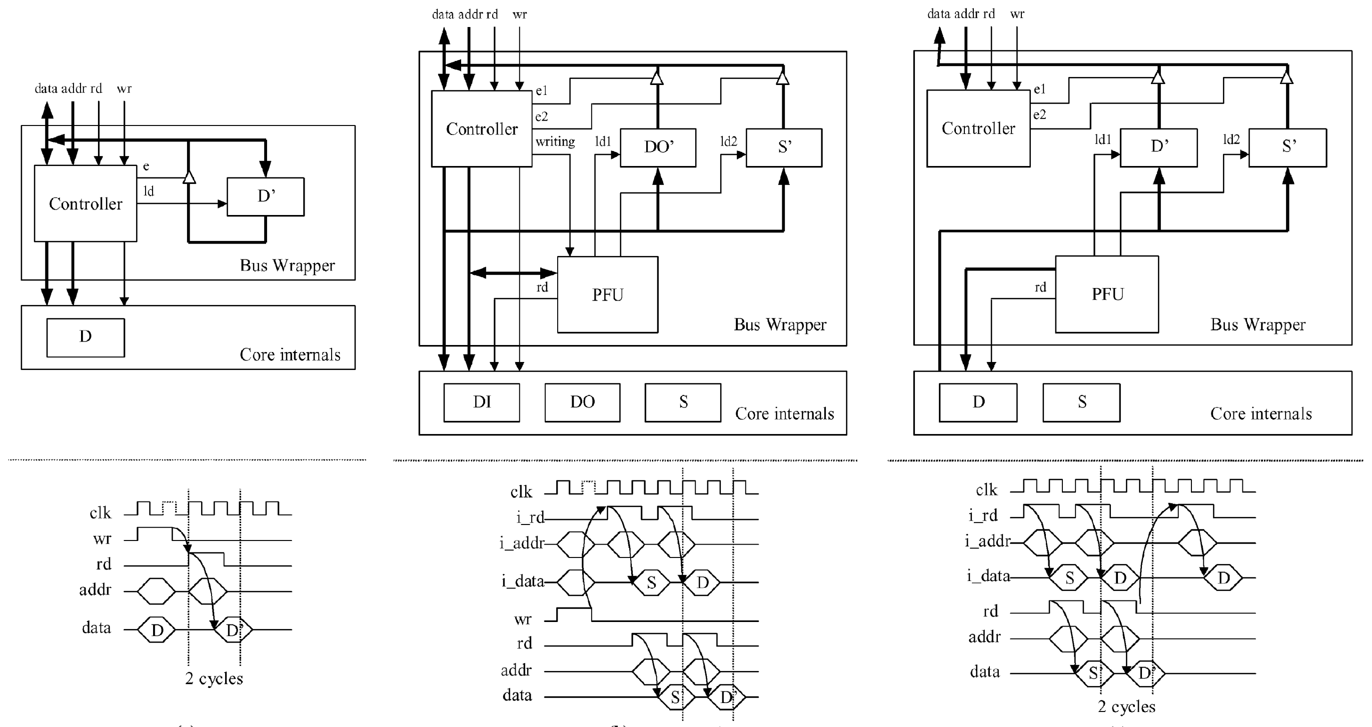

Fig. 6. Bus wrapper architecture and timing diagrams for (a) Core1, (b) Core2, and (c) Core3.

P1: IBC
CM042A-03 ACM-TRANSACTION January 18, 2002 10:37

(3) *Prefetch unit*: The PFU implements the prefetch heuristics, and is responsible for reading data from the core internals to the prefetch registers. Its goal is to maximize hits.

The architecture for the Core1 situation is shown in Figure 6(a), showing one register D and its corresponding prefetch register D′. Since D is only updated by the on-chip bus, no prefetch unit is needed; instead, we can write to D′ whenever we write to D. Such a lack of a PFU is an exception to the normal situation. Figure 6(b) shows the architecture for the Core2 situation. The data output register DO and status register S both have prefetch registers in the BW, but the data input register DI does not since it is never read by the on-chip bus. The PFU carries out its prefetch heuristic (see next section), unless the controller asserts the "writing" line, in which case the PFU suspends prefetching so that the controller may write to DI over the core internal bus. Figure 6(c) shows the architecture for the Core3 example, which has no write-access registers and hence does not include the bus between the controller and the core internal bus.

4.4.2 *Heuristics.* We applied the following prefetch heuristics within each core's bus wrapper:

Core1: Upon a system write to the data register D, simultaneously write the data into the prefetched data register D′. This assumes that a write to the data register will occur prior to a read from the register.

Core2: After the system writes to the data input register DI, we read the core's internal status register S into the prefetched status register S′. If the status indicates completion, we read the core's internal data output register DO into the prefetched data-output register DO′. We repeat this process.

Core3: We continuously read the core's internal status register S into the prefetched status register S′ until the status indicates the buffer is no longer empty. We then read the core's data register D into the prefetched data register D′. While waiting for the system to read the data, we continuously read the core's internal status register into the prefetched status register, thereby providing the most current status information. When the data is read by the system, depending on whether the buffer is empty, we either read the next data item from the core or repeat the process.

Figure 6 shows timing diagrams for the three cores with a BW and prefetching. In all three cores, the read latency for each core with a BW and prefetching was equal to the latency of that core without a BW, thus eliminating the performance penalty.

Note that a BW's architecture and heuristic are dependent on the core internals. This is acceptable since the core developer builds the BW. The BW controller's bus interface is not, however, dependent on the core internals, as desired.

4.5 Experiments

We implemented cores representing the three earlier common examples, in order to evaluate performance, power, and size tradeoffs achievable through

Table II. Impact of Prefetching on Several Cores

Metric	Core1	Core2	Core3
Size w/o BW (gates)	1080	2638	10571
Size w/BW w/o PF (gates)	2669	4234	11506
Size w/BW w/PF (gates)	3066	6172	13146
Performance w/o BW (ns)	6895	5515	2865
Performance w/BW w/o PF (ns)	9835	8515	4305
Performance w/BW w/PF (ns)	6895	5545	2875
Power w/o BW (microwatts)	1365	480	2016
Power w/BW w/o PF (microwatts)	1399	616	1521
Power w/BW w/PF (microwatts)	1422	560	2229
Energy w/o BW (nJ)	9.41	2.65	5.77
Energy w/BW w/o PF (nJ)	13.76	5.25	6.55
Energy w/BW w/PF (nJ)	9.81	3.11	6.41

prefetching. Results are summarized in Table II. All three cores were written as soft cores in register-transfer-level behavioral VHDL. The three cores required 136, 220, and 226 lines of VHDL, respectively. We synthesized the cores using Synopsys Design Compiler. Performance, average power, and energy metrics were measured using Synopsys analysis tools, using a suite of core test vectors for each core. It is important to note that these cores have simple internal behavior and were used for experimentation purposes only. Although these examples are small, because the PFU unit is independent of the core internals our approach can be applied to larger examples as well.

In all three cores, when prefetching was added to the BW's, any performance penalty was effectively eliminated. In Core2 and Core3, there was a trivial one-time 30-ns and 10-ns overhead associated with the initial time required to start and restart the prefetching process for the particular prefetch heuristics.

The addition of a BW to cores adds size overhead to the design, but size constraints continue to relax as chip capacities continue their exponential growth. In the three cores described above, there was an average increase in the size of each core by 1352 gates. The large percentage increase in size for Core1 and Core2 was due to the fact that these cores were unusually small to begin with since they had only simple internal behavior, having only 1000 or 2000 gates; more typical cores would have closer to 10,000 or 20,000 gates, so the percentage increase caused by the few thousand extra gates would be much smaller.

In order for prefetching to be a viable solution to our problem, power and energy consumption must also be acceptable. Power is a function of the amount of switching in the core, while energy is a function of both the switching and the total execution time. BWs without prefetching caused both an increase in power (due to additional internal transfers to the BW) and an increase in overall energy consumption (due to longer execution time) in all three cores. Compared to BWs without prefetching, BWs with prefetching may increase or decrease power depending on the prefetch heuristic and particular application. For example, in Core1 and Core3, there was an increase in power due to the constant activity of the prefetch unit, but in Core2, there was a decrease in power due to the periods of time during which the prefetch unit was idle. However, in all three cores, the use of prefetching in the BW decreased energy consumption over the

Table III. Impact of Prefetching on Digital Camera Performance

	Reads	Cycles w/o prefetching	Cycles w/ prefetching
CCD—Status	3	12	6
CCD—Data	256	1024	512
CODEC—Status	256	1024	512
CODEC—Data	257	1028	514
Total for 2 cores	772	3088	1544
Digital Camera Peripheral I/O Access		6,224	4,680
Digital Camera Processor Execution		42,392	42,392
Digital Camera		48,616	47,072

Table IV. Impact of Prefetching on Digital Camera Power/Energy

	No BW	BW w/o prefetching	BW w/prefetching
Power, mW	95.4	98.1	98.1
Energy, μJ	44.9	47.7	46.2

BW without prefetching because of reduced execution time. In addition, the increase in energy consumption relative to the core without a bus wrapper was fairly small.

To further evaluate the usefulness of prefetching, we analyzed a digital camera as shown in Figure 5. We initially had implemented the CCD and CODEC cores using BWs without prefetching. We therefore modified them to use prefetching, and compared the two versions of the digital camera system. Table III provides the number of cycles for reading status and data registers for the two cores to capture one picture frame. The number of cycles required for these cores with prefetching was half of the number of cycles required without prefetching. The improvement in performance for reads from the CCD and CODEC was 50%. The overall improvement in performance for the digital camera was over 1500 cycles just by adding prefetching to these two cores, out of a total of about 47,000 cycles to capture a picture frame. The prefetching performance increase of the digital camera was directly related to the ratio of I/O access to processor computation. Because the digital camera spends 78% of execution time performing computation and only 12% performing I/O access, prefetching did not have a large impact on overall performance. However, the increase in performance for peripheral I/O access was 25%. Therefore, for a design that is more I/O intensive, one would expect a greater percentage performance increase. Furthermore, if the processor was pipelined, the number of cycles required for program execution would decrease, and the percentage of time required for I/O access would increase. Thus, one would again expect a greater percentage performance increase from prefetching. Adding prefetching to other cores would of course result in even further reductions. The power and energy penalties are shown in Table IV. We see that, in this example, prefetching is able to eliminate any performance overhead associated with keeping interface and internals separated in a core.

Prefetching enables elimination of the performance penalty while fully supporting the idea of a VSI standard for the internal bus between the BW and core internals. It can also be varied to tradeoff performance with size and power; ideally, a future tool would synthesize a BW satisfying the power, performance, and size constraints given by the user of a core.

5. “REAL-TIME" PREFETCHING

5.1 Overview

One of the drawbacks to the prefetching technique described above is that the prefetch unit was manually designed and created. We desired to also investigate an automatic solution to designing a prefetch unit. The bus wrapper in our automated approach has an identical architecture to our previous bus wrapper with prefetching. However, we now redefine the task of the prefetch unit (PFU). The prefetch unit is responsible for keeping the prefetch registers as up-to-date as possible, by prefetching 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.

We assume we are given a list of the core’s readable registers, which must be prefetched. We also assume that the bus wrapper can accommodate one copy of each such register. 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 prefetch 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 prefetching is not allowed, since prefetched data is necessarily at least one cycle old. A constraint of 0 also means that the access-time constraint must be at least four cycles.

—*Register access-time constraint*: This constraint represents the maximum number of cycles that a read access may take. The minimum is two, in which case the register must be prefetched. 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 prefetch registers using a schedule that satisfies the age and access-time constraints on those registers. Note that certain registers may be prefetched more frequently than others if this is required to satisfy differing register access constraints.

The tradeoff of prefetching 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 7 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 four and six cycles, respectively. We might use a naive prefetching heuristic that prefetches on every idle cycle, reading A 60% and

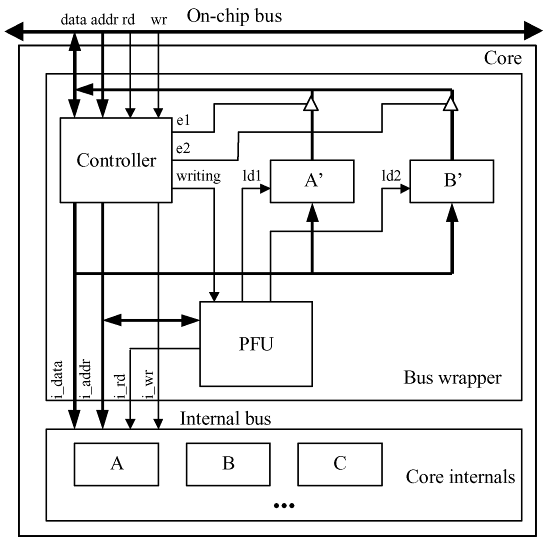

Fig. 7. Bus wrapper with prefetching.

Table V. Prefetch Scheduling for the Core in Figure 7

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	

B 40% of the time, leading to Schedule 1 in Table V. 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.

During our investigation for heuristics to solve the prefetching 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 prefetching heuristics (based on real-time scheduling heuristics) and analysis methods.

5.2 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 prefetching problem to the real-time process-scheduling problem is as follows.

—*Register → process*: A register that must be scheduled for prefetching corresponds to a process that must be scheduled for execution.

—*Internal bus → processor*: The internal bus can accommodate only one prefetch at a time. Likewise, a processor can accommodate only one process execution at a time. Thus, the internal bus corresponds to a processor.

—*Prefetch → process execution*: A prefetch 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 prefetched, 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.

—*Prefetch time → Process computation time*: A prefetch corresponds to a process execution, so the time for a prefetch corresponds to the computation time for a process. In this paper, we assume a prefetch requires two 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 prefetching problem.

5.3 Heuristics

5.3.1 *Cyclic Executive Approach.* The cyclic executive approach [Burns and Wellings 1997] 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 prefetched. This approach is attractive due to its simplicity. However, it does not handle sporadic processes (in our case, sporadic writes),

Table VI. Prefetch Core Descriptions

Core	Register	Max Age	D	Priority RM	PF Time	Response Time	Util.	Util. Bound
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		

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 VI contains data pertaining to all three of our cores. Table VI contains information regarding maximum register age constraint (*Max Age*), register access time constraint or deadline (*D*), rate monotonic priority assignment (*Priority RM*), time required to prefetch register (*PF Time*), response time of register (*Response Time*), utilization for register set (*Util.*), and utilization bound for register set (*Util. Bound*). 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 three. 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 five and a major cycle of 25.

5.3.2 *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 prefetch) next based on a priority scheme. A rate monotonic priority assignment [Burns and Wellings 1997] 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 1 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 VI for data pertaining to all three of our cores. For Core1, the register age constraint of the register DATA is three 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. 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.

5.3.3 *Utilization-Based Schedulability Test.* The utilization-based schedulability test [8] is used to quickly indicate whether a set of processes can be scheduled, or in our case whether the registers can be prefetched. All N registers of a register set can be prefetched 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 prefetched. 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:

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

If the register set passes this test, all registers can be prefetched 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 prefetched, but does not indicate that a register set cannot be prefetched.

We can analyze our cores to determine whether we can schedule them. From Table VI, 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 four registers is 75.7%. Therefore, we have failed the utilization-based schedulability test, though a schedule might still exist.

5.3.4 *Response-Time Analysis.* Response-time analysis [Burns and Wellings 1997] 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.

$$R_i = C_i + I_i \tag{2}$$

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 VI, we can see that the registers of all three cores will meet their register age constraints. Therefore, it is possible to create a prefetching 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 prefetched. We refer the reader to Burns and Wellings [1997] for further details on response-time analysis.

5.3.5 *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 prefetches 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 prefetched. 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 prefetching 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.

5.3.6 *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 [Burns and Wellings 1997]. 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 prefetching algorithm. The deadline for the BIAS register will be such that we can ensure that writes will always have

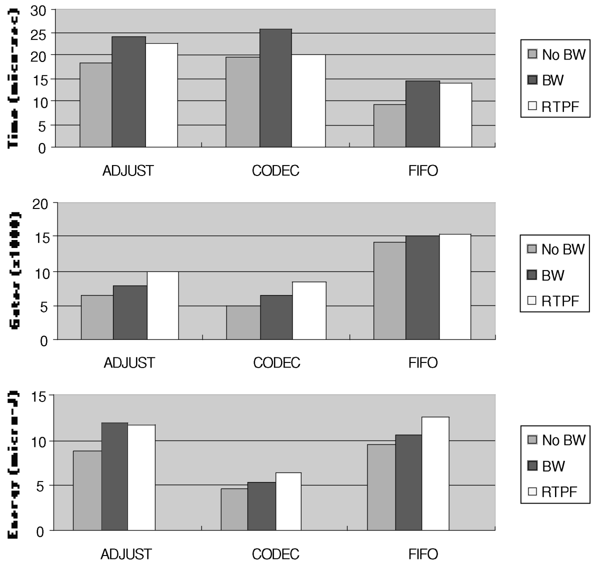

Fig. 8. Performance in ns (top), size in gates (middle), and energy in nJ (bottom).

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.

5.4 Experiments with "Real-Time" Prefetching

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. 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 prefetching most clearly. We used Synopsys Design Compiler for synthesis as well as Synopsys power analysis tools.

Figure 8 summarizes the results for the three cores. For each core, we examined three possible bus wrapper configurations: no bus wrapper (*No BW*), a bus wrapper without prefetching (*BW*), and a bus wrapper with real-time prefetching (*RTPF*).

The first chart in Figure 8 summarizes performance results. Using our real-time prefetching heuristic, we can see a good performance improvement when compared to a bus wrapper without prefetching. However in FIFO, we only see a very small performance improvement using real-time prefetching. This small improvement is due to fact that the DATA register in FIFO cannot be prefetched using this approach. If we were to prefetch DATA using real-time prefetching, we would empty the FIFO and lose data. Furthermore, without any prefetching, we can see a significant performance penalty.

The second chart in Figure 8 summarizes size results. As expected, the size of the cores increased when a bus wrapper was added, and further increased when prefetching was added to the bus wrapper. The average increase in size caused by adding real-time prefetching to the bus wrapper was only 1.4K gates. This increase in design complexity was due to the need to keep track of current register ages. Furthermore, this size increase was relatively small when compared to a typical core size of 10K to 20K gates.

The third chart in Figure 8 summarizes energy consumption for our test vectors. In all three cores, there was an overall increase in energy consumption when a bus wrapper was added to the core. However, the addition of prefetching to the bus wrappers did not always strictly increase or decrease energy consumption. In fact, real-time prefetching increased energy consumption in CODEC and FIFO, and decreased energy consumption in ADJUST. As expected, when compared to the core without a bus wrapper, prefetching resulted in an increase in energy consumption.

6. UPDATE-DEPENDENCY BASED PREFETCHING USING PETRI NETS

6.1 Overview

In some cases, a core designer may be able to provide us more information regarding when the core's internal registers get updated—in particular, update dependencies among registers, e.g., if register A is updated externally, then register B will be updated one cycle later. Using this information, we can design a schedule that performs fewer prefetches to satisfy given constraints, and thus can yield advantages of being able to handle more complex problems, or of using less power.

6.2 General Register Attributes

We need a method for capturing the information a designer provides regarding register updates. In Section 3.2, we provided a taxonomy of register attributes can be used to categorize how a register is used. We extend this by introducing *update dependecies*. Update dependencies provide further details on when a 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.

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 would best fit our requirements.

6.3 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 outputed 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 prefetching 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 9 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 prefetch unit. MD is updated when GO is equal to 1, a write request for MD is initiated on the OCB, and some external

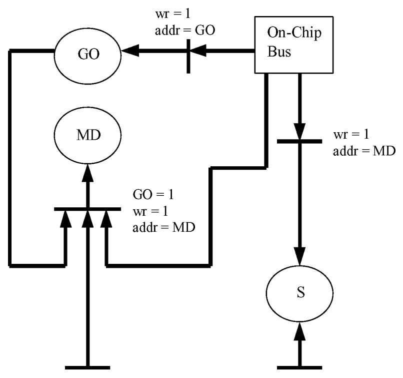

Fig. 9. ADJUST register dependencies.

event occurs. Therefore, we now have a complete model of the ADJUST core that can be used to create a prefetching algorithm.

Using the current model of ADJUST, we need three prefetch 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 prefetch 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 prefetching 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 prefetch unit by not prefetching MD if it is not needed.

6.4 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 10 shows a refined register update dependency model for the ADJUST core. In this new model, we have eliminated one random transition by replacing

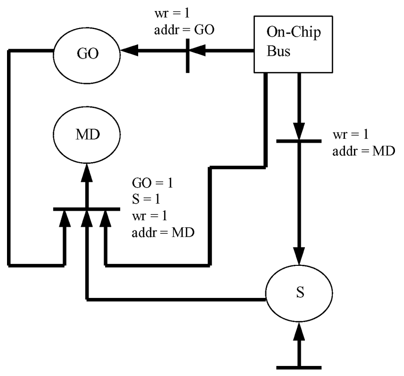

Fig. 10. Refined ADJUST register dependencies.

it with a transition that will fire if S is equal to 1. Hence, we need to modify our prefetching algorithm to accompany this change. We now know that we only need to prefetch 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 prefetching algorithm and will again reduce power consumption.

6.5 Prefetch Scheduling

Given an update dependency model of a core, we need to construct a schedule to prefetch the core's registers into the bus wrapper's prefetch registers. Figure 11 describes our update dependency model prefetching heuristic using pseudo-code. Our heuristic uses the update dependency model in conjunction with our real-time prefetching to create a schedule for prefetching the core's registers. The following description will further elaborate on the heuristic.

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

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

Next, our prefetch heuristic needs to respond to read and write requests on the OCB. In the event of a read request, the prefetch unit will add any outgoing arcs to the list of arcs needed to be analyzed. As described in our real-time prefetching 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 prefetch, 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

```
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
      }
   }
}
```

Fig. 11. General register model prefetching heuristic used to implement PFU.

into the corresponding prefetch register, setting the access-time constraint to 0, and adding the write register to the prefetch register heap. In addition, any outgoing arcs will be added to the list of update arcs.

We will use our real-time prefetching to prefetch 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 prefetching. According to this heuristic, registers with an access-time constraint of 0 will be prefetched first. That means that all write requests and, as we will describe later, all registers that have been updated will be prefetched 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 prefetched, our prefetching heuristic will next schedule registers with incoming random transitions according to their rate-monotonic priority assignment. Therefore, our prefetch register heap will be sorted first by deadline-monotonic priority assignment and further by rate-monotonic priority assignment.

After each register prefetch 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 prefetched as soon as possible.

In order to better understand this prefetching 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 six cycles. Therefore, we set the register age constraint to six cycles, and the register access-time constraint to two, indicating that the register S must be prefetched every six cycles. For MD, both the register age and access-time constraints are two 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 prefetching 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 prefetch the registers when needed and perform write operations.

6.6 Experiments with Update-Dependency Prefetching

We implemented the update dependency prefetching on the same three cores as above, namely ADJUST, CODEC, and FIFO. Figure 12 summarizes the results for the three cores. We now have four possible bus wrapper configurations for each core: no bus wrapper (*no BW*), a bus wrapper without prefetching (*BW*), a bus wrapper with real-time prefetching (*RTPF*), and a bus wrapper with our update dependency prefetching model (*UDPF*).

The first chart in Figure 12 summarizes performance results. In all three cores, the use of our update dependency prefetching method almost entirely eliminated the performance penalty associated with the bus wrapper. There was still a slight overhead caused by starting the prefetch unit. Using our real-time prefetching heuristic, we can see that although there is a performance improvement when compared to a bus wrapper without prefetching, it did not perform as well as our update dependency model.

The second chart in Figure 12 summarizes size results. The average increase in size caused by adding the update dependency prefetching technique to the bus wrapper was only 1.5K gates. In comparison, real-time prefetching resulted in an average increase of 1.4K gates. It is interesting to note why the two

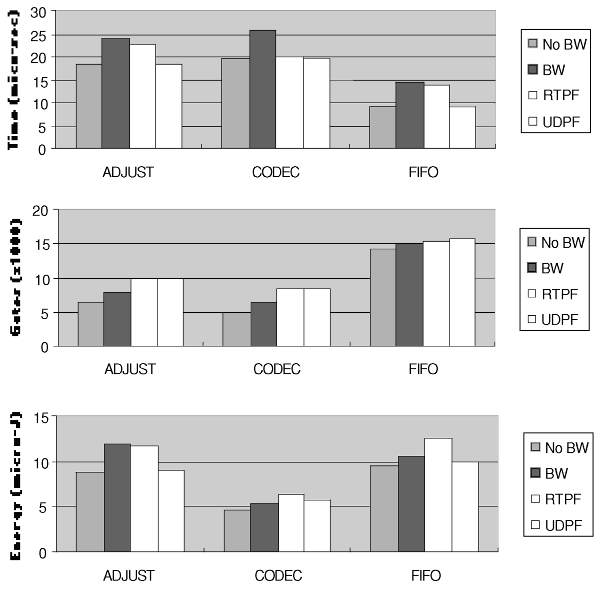

Fig. 12. Performance in ns (top), size in gates (middle), and energy in nJ (bottom).

approaches, although quite different, resulted in approximately the same size increase. As stated earlier, using real-time prefetching, 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.

The third chart in Figure 12 summarizes energy consumption for our test vectors. However, the addition of prefetching to the bus wrappers does not always strictly increase or decrease energy consumption. In fact, we can see that in ADJUST and FIFO, there is a decrease in energy consumption when our update dependency prefetching is added to the bus wrapper, but in CODEC, there is an increase. On the other hand, real-time prefetching increases energy consumption in CODEC and FIFO, and decreases energy consumption in ADJUST.

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

7. CONCLUSIONS

While keeping a core's interface and internal behavior separated is key to a core's marketability, we demonstrated that the use of such bus wrappers, both non-PVCI and PVCI, results in size, power, and performance overhead. Thus, the retargetability advantages of such a standard seem to come with some penalties.

We introduced prefetching as a technique to overcome the performance overhead. We demonstrated that in some common cases of register combinations, prefetching eliminates the performance degradation at the expense of acceptable increases in size and power. By overcoming the performance degradation associated with bus wrappers, prefetching thus improves the usefulness of cores.

We have further 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 also provided a general register update dependency model that we used to construct a more efficient prefetching schedule, in conjunction with our real-time prefetching. 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 prefetch registers when compared to our real-time prefetching 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.

8. FUTURE WORK

Although prefetching works well, there are many possibilities for improvements. In our current approach we assume that all registers of the core will be prefetched. However, for cores with large numbers of registers, this approach is not feasible. Thus, we are considering restricting the number of registers that can appear in a bus wrapper. This creates new cache-like issues such as mapping, replacement, and coherency issues that are not present in our current design. In addition, we can further evaluate the effects of prefetching on larger core examples. Another direction involves developing prefetching heuristics that optimize a given cost function of performance, power, and size.

REFERENCES

BURNS, A. AND WELLINGS, A. 1997. *Real-time systems and programming languages.* Addison-Wesley, Reading, MA.

CHOU, P., ORTEGA, R. B., AND BORRIELLO, G. 1995. Interface co-synthesis techniques for embedded systems. In *International Conference on Computer-Aided Design* (1995), 280–287.

CLEMENT, B., HERSEMEULE, R., LANTREIBECQ, E., RAMANADIN, B., COULOMB, P., AND POGODALLA, F. 1999. Fast prototyping: a system design flow applied to a complex system-on-chip multiprocessor design. In *Design Automation Conference* (1999).

COWARE. 1999. *CoWare N2C design system overview.* http://www.coware.com/products.html.

FLEISCHMANN, J., BUCHENRIEDER, K., AND KRESS, R. 1999. Java driven codesign and prototyping of networked embedded systems. In *Design Automation Conference* (1999).

KOZYRAKIS, C. AND PATTERSON, D. A. 1998. A new direction for computer architecture research. *Computer 31*, 11.

KUHN, T., ROSENSTIEL, W., AND KEBSCHULL, U. 1997. Description and simulation of hardware/software systems with Java. In *Design Automation Conference* (1997).

LUI, C. AND LAYLAND, J. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. Assn. Comput. Mach.* 46–61.

MADISETTI, V. AND SHEN, L. 1997. Interface design for core-based systems. *IEEE Des. Test Comput.* 42–51.

MENTOR GRAPHICS. n.d. Inventra core library. http://www.mentorg.com/inventra/.

METAFLOW. 1999. *Implosion ARM-based Internet appliance system on a chip platform.* http://www.metaflow.com/html/implosion.htm.

MOTOROLA. 1999. IP *interface standard document.* http://www.mot-sps.com/technology/srs/.

PATTERSON, D. A. AND HENNESSY, J. L. 1990. *Computer architecture, a quantitative approach.* Morgan Kaufmann, San Francisco, CA.

PHILIPS SEMICONDUCTORS. 1999. *Introduction to rapid silicon prototyping: hardware-software co-design for embedded systems-on-a-chip ICs.* http://www.semiconductors.com/acrobat/other/technology/velocity/paper2221.pdf.

RABAEY, J., ABNOUS, A., ICHIKAWA, Y., SENO, K., AND WAN, M. 1997. Heterogeneous reconfigurable systems. In *Proceedings of the IEEE Workshop on Design and Implementation of Signal Processing Systems* (1997), 24–34.

ROWSON, J. AND SANGIOVANNI-VINCENTELLI, A. 1997. Interface-based design. In *Design Automation Conference* (1997).

SEMICONDUCTOR INDUSTRY ASSOCIATION. 1999. *International technology roadmap for semiconductors: 1999 edition.* International SEMATECH, Austin, TX.

VAHID, F. AND GIVARGIS, T. 1999. The case for a configure-and-execute paradigm. In *International Workshop on Hardware/Software Codesign* (1999).

VAHID, F. AND TAURO, L. 1997. An object-oriented communication library for hardware-software co-design. In *International Workshop on Hardware/Software Codesign* (1997), 81–86.

VAN MEERBERGEN, J., TIMMER, A., LEIJTEN, J., HARMSZE, F., AND STRIK, M. 1998. Experiences with system level design for consumer ICs. In *Proceedings of the IEEE Computer Society Workshop on VLSI System Level Design* (1998).

VERCAUTEREN, S., LIN, B., AND DE MAN, H. 1996. Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications. In *Design Automation Conference* (1996), 547–551.

VIRTUAL SOCKET INTERFACE ASSOCIATION. 1997a. *Architecture document.* http://www.vsi.org.

VIRTUAL SOCKET INTERFACE ASSOCIATION. 1997b. *Virtual component interface standard version 1.0.* http://www.vsi.org.

VIRTUAL SOCKET INTERFACE ASSOCIATION. 1998. *On-chip bus development working group, Specification 1 Version 1.0* (OCB 1 1.0). http://www.vsi.org.

Received July 2000; accepted May 2001