Pre-fetching for Improved Core Interfacing

Roman Lysecky, Frank Vahid, Tony Givargis, Rilesh Patel
Department of Computer Science and Engineering
University of California, Riverside
http://www.cs.ucr.edu/~dalton

Abstract

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. However, this separation can lead to a performance penalty when reading a core's internal registers. We introduce pre-fetching, 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 pre-fetching technique, classify different types of registers, describe our initial pre-fetching architectures and heuristics for certain classes of registers, and highlight experiments demonstrating the performance improvements and size/power tradeoffs.

Keywords

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

1. Introduction

Silicon capacity continues to increase faster than the ability for designers to use that silicon, resulting in the well-known productivity gap [1]. Many people propose extensive reuse of pre-designed intellectual property *cores* to reduce this gap [2], where typical cores include microprocessors, microcontrollers, digital signal processors, encoders/decoders, bus interfaces, and numerous other common peripheral components. In response, several commercial libraries of cores have evolved in recent years (e.g., [3]). 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 across a wide variety of buses. Unfortunately, standardizing on one or two on-chip buses 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) [4]. Thus, to achieve such ease of integration, many have proposed designing cores with their interface behavior kept separate from their internal behavior [4][5][6]. This is especially true for peripheral cores, for which portability is a key issue. Such separation isolates any bus-specific changes to a small region of the core, which we will call an *interface module (IM)*.

However, such modularity 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 interface module before the data can be output to the bus.

We propose a solution to this performance penalty, called pre-fetching. Briefly, pre-fetching is analogous to caching, wherein we store local copies of registers inside an interface module so that a register read results in outputting of this local copy, thus eliminating extra cycles on a read. As with caching, pre-fetching schemes must strive to maximize the hit ratio. Pre-fetching requires appropriate interface module architecture design, the focus of this paper, as well as heuristics that maximize the hit/miss ratio.

In this paper, we describe the idea of pre-fetching, classify common core registers, describe pre-fetching architectures and simple heuristics for common classes, and provide results demonstrating the impact on performance, power and size. Finally, we provide conclusions and discuss future work.

2. Pre-fetching overview

2.1 Problem description

Separating a core's interface behavior and internal behavior can lead to performance penalties. For example, consider the core architectures shown in Figure 1(a), (b) and (c), respectively showing a core with no IM, a core with an IM but without pre-fetching, and a core with an IM with pre-fetching. The latter two architectures are similar to that being proposed by the VSIA. The IM interfaces with the system bus, whose protocol may be arbitrarily complex, including a variety of features like arbitration. The IM also interfaces with the core internals, over a core internal bus; this bus is typically extremely simple, implementing a straightforward data transfer (and it is this internal bus that the VSI On-Chip Bus group is standardizing). Without an IM, a read of a core's internal register from the system bus may take as little as 2 cycles, as shown in Figure 2(a). With an IM, the read of a core's internal register may require 4 cycles, 2 from the internal module to the IM, and 2 from the IM to the bus. Thus, a read may require extra cycles compared with a core whose interface and internal behavior was combined.

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 the system 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.

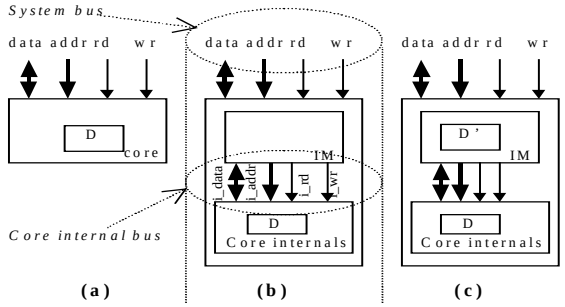

Figure 1: Core interface options: (a) No IM, (b) IM w/o pre-fetching, (c) IM w/ pre-fetching.

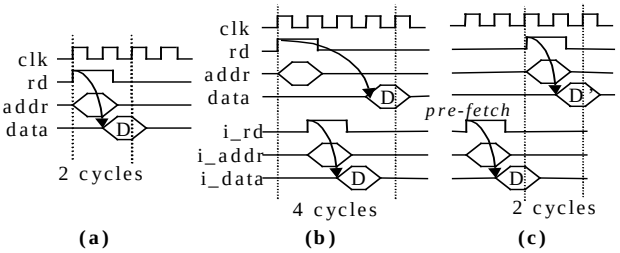

Figure 2: Interface option timing: (a) No IM, (b) IM w/o pre-fetching, (c) IM w/ pre-fetching.

We focus on peripheral cores, whose registers will be read by a microprocessor over a system 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 pre-fetching. *Pre-fetching* is the technique of copying a core's internal register data into a pre-fetch register in a core's IM, so that when a read request from the bus occurs, the core can immediately output pre-fetched 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 pre-fetch register, while a miss means that the data must first be fetched into a pre-fetch register before being output to the system bus.

For example, Figure 2(c) shows that pre-fetching a core's internal register D into an IM register D' results in a system read again requiring only 2 cycles, rather than 4.

2.2 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 pre-fetching approaches.

After examining cores (primarily from the Inventra library) 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 system 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.

2.3 Commonly-occurring register types

For our first attempt at developing pre-fetching 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.

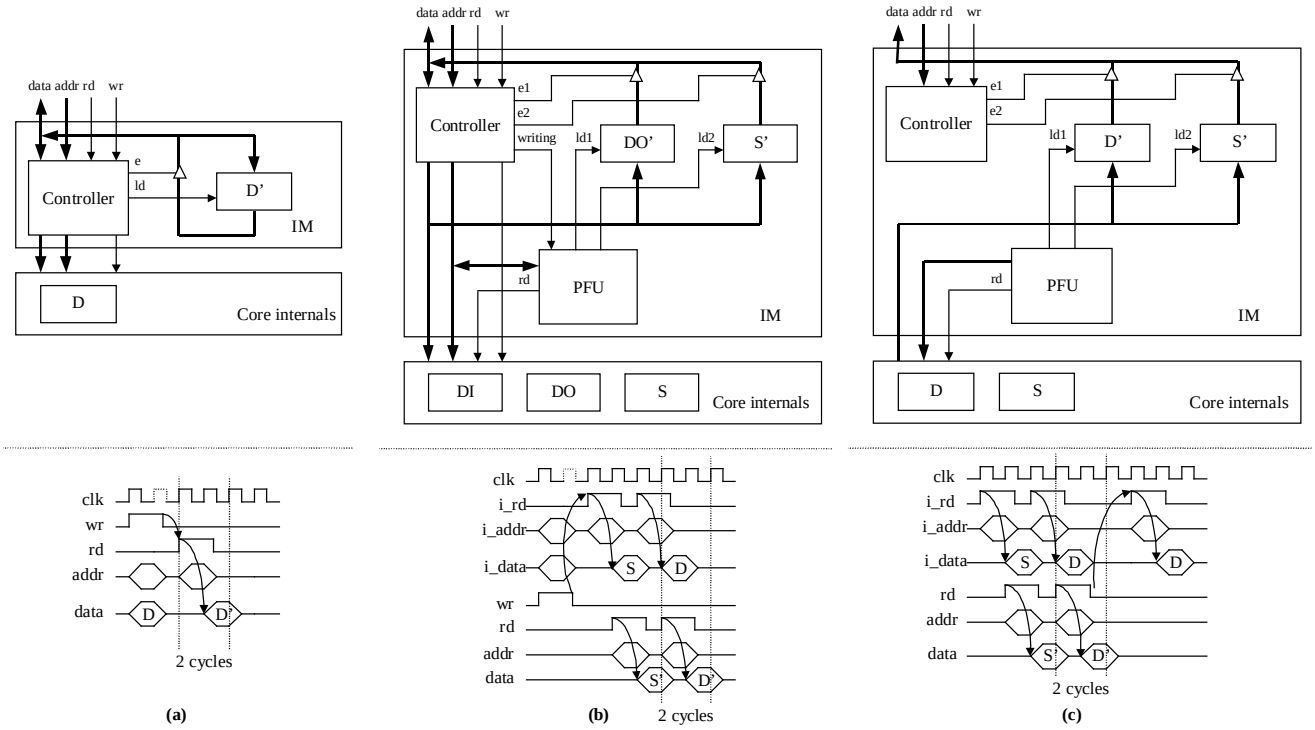

Figure 3: Interface module architecture and timing diagrams for (a) Core1, (b) Core2, and (c) Core3.

(2) Core2 -- 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 pre-fetch 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 pre-fetched, 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.

2.4 Related work

While much work has been done on interfacing, to our knowledge none of the literature includes the idea of pre-fetching. Most interfacing work has focused on automatically synthesizing logic to interface to a bus (e.g., [9][10]), synthesizing the bus itself (e.g., [11]), or defining a standard bus protocol (e.g., [12]).

3. Pre-fetching architectures and heuristics

3.1 Architectures

In order to implement the pre-fetching for each of the above listed combinations of registers, we developed architectures for interface modules for each. Figure 3 illustrates the architecture for each of the three combinations respectively. Each IM architecture has three regions:

1. *Controller*: The controller's main task is to interface with the system 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 pre-fetch register data onto the bus; for a hit, this outputting is done immediately, while for a miss, it is done only after forcing the pre-fetch unit to first read the data from the core internals.
2. *Pre-fetch registers*: These registers are directly connected to the system bus for fast output. Any output to the bus must pass through one of these registers.
3. *Pre-fetch unit*: The PFU implements the pre-fetch heuristics, and is responsible for reading data from the core internals to the pre-fetch registers. Its goal is to maximize hits.

The architecture for the Core1 situation is shown in Figure 3(a), showing one register D and its corresponding pre-fetch register D'. Since D is only updated by the system bus, no pre-

fetch 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 3(b) shows the architecture for the Core2 situation. The data output register DO and status register S both have pre-fetch registers in the IM, but the data input register DI does not since it is never read by the system bus. The PFU carries out its pre-fetch heuristic (see next section), unless the controller asserts the "writing" line, in which case the PFU suspends pre-fetching so that the controller may write to DI over the core internal bus. Figure 3(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.

3.2 Heuristics

We applied the following pre-fetch heuristics within each core's interface module.

Core1: Upon a system write to the data register D, simultaneously write the data into the pre-fetched 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 pre-fetched status register S'. If the status indicates completion, we read the core's internal data output register DO into the pre-fetched data-output register DO'. We repeat this process.

Core3: We continuously read the core's internal status register S into the pre-fetched status register S' until the status indicates the buffer is no longer empty. We then read the core's data register D into the pre-fetched data register D'. While waiting for the system to read the data, we continuously read the core's internal status register into the pre-fetched 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 3 shows timing diagrams for the three cores with an IM and pre-fetching. In all three cores, the read latency for each core with an IM and pre-fetching was equal to the latency of that core without an IM, thus eliminating the performance penalty.

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

4. Experiments

We implemented cores representing the three earlier common examples, in order to evaluate performance, power and size tradeoffs achievable through pre-fetching. Results are summarized in Table 1. 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.

Table 1: Impact of pre-fetching on several cores.

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

In all three cores, when pre-fetching was added to the IM'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 pre-fetching process for the particular pre-fetch heuristics.

The addition of an IM 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 is due to the fact that these cores were unusually small to begin with since they had only simple internal behavior, having only one or two thousand gates; more typical cores would have closer to ten or twenty thousand gates, so the percentage increase caused by the few thousand extra gates would be much smaller.

In order for pre-fetching 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. IM's without pre-fetching cause both an increase in power (due to additional internal transfers to the IM) and an increase in overall energy consumption (due to longer execution time) in all three cores. Compared to IM's without pre-fetching, IM's with pre-fetching may increase or decrease power depending on the pre-fetch heuristic and particular application. For example, in Core1 and Core3, there was an increase in power (due to the constant activity of the pre-fetch unit), but in Core2, there was a decrease in power (due to the periods of time during which the pre-fetch unit was idle). However, in all three cores, the use of pre-fetching in the IM decreased energy consumption over the IM without pre-fetching (because of reduced execution time). In addition, the increase in energy consumption relative to the core without an interface module was fairly small.

To further evaluate the usefulness of pre-fetching, we analyzed a digital camera system [8]. The digital camera consists of a CCD preprocessor core for capturing images, a CODEC core to compress and decompress the picture frames,

Table 2: Impact of pre-fetching on Digital Camera performance.

	Reads	Cycles w/o pre-fetching	Cycles w/ pre-fetching
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		48,616	47,072
Digital Camera Peripheral I/O Access		6,224	4,680
Digital Camera Processor Execution		42,392	42,392

Table 3: Impact of pre-fetching on Digital Camera power/energy.

	No IM	IM w/o pre-fetching	IM w/ pre-fetching
Power, mW	95.4	98.1	98.1
Energy,µJ	44.9	47.7	46.2

and several other cores, along with a microprocessor, BIOS and memory. We initially had implemented the CCD and CODEC cores using IM's without pre-fetching. We therefore modified them to use pre-fetching, and compared the two versions of the digital camera system. Table 2 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 pre-fetching is half of the number of cycles required without pre-fetching. The improvement in performance for reads from the CCD and CODEC was 50%. The overall improvement in performance for the digital camera was over 1,500 cycles just by adding pre-fetching to these two cores, out of a total of about 47,000 cycles to capture a picture frame. The pre-fetching performance increase of the digital camera is 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, pre-fetching 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 pre-fetching. Adding pre-fetching to other cores would of course result in even further reductions. The power and energy penalties are shown in Table 3. We see that, in this example, pre-fetching is able to eliminate any performance overhead associated with keeping interface and internals separated in a core.

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

5. Conclusions

We introduced pre-fetching as a technique to overcome the main drawback of degraded performance when keeping a core's interface and internal behavior separated. As such separation is key to a core's marketability, pre-fetching thus improves the usefulness of cores. In this paper, we demonstrated that in some common cases of register combinations, pre-fetching eliminates the performance degradation, at the expense of acceptable increases in size and power. Extensive future work will focus on developing pre-fetching architectures and heuristics for more complex register combinations found in many cores, requiring techniques for specifying register inter-dependencies and priorities, for allocating pre-fetch registers within certain size constraints, for scheduling pre-fetching over the core internal bus to maximize the hit/miss ratio, while considering power and performance constraints.

6. Acknowledgements

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

7. References

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

[2] Virtual Socket Interface Association, Architecture Document, http://www.vsi.org, 1997.

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

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

[5] F. Vahid and L. Tauro, An Object-Oriented Communication Library for Hardware-Software Co-Design, International Workshop on Hardware/Software Codesign, pp. 81--86, 1997.

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

[7] B. Payne. Rapid Silicon Prototyping: Paradigm for Custom System-on-a-Chip Design, http://www.vlsi.com/velocity, 1998.

[8] F. Vahid, T. Givargis, The Case for a Configure-and-Execute Paradigm. International Workshop on Hardware/Software Codesign, 1999.

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

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

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

[12] 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.