TOKYO INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

An Efficient AMI-Based Cache System on FPGA Computer Systems

Yuki Matsuda

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Engineering

Advisor: Associate Professor Kenji Kise

Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

January 2016

Abstract

An FPGA (Field Programmable Gate Array) is an integrated circuit on which developers can implement their custom circuits. Since the design of soft processors, which are processors implemented on FPGAs, can be modified more flexibly than conventional processors, soft processors are widely used for embedded applications. They often contain a cache system to achieve faster access speed than the external DRAM. Because the resources on an FPGA are limited, high performance cache with fewer resources for FPGA soft processors has been required.

Conventional approaches to improve cache performance are following: larger cache capacity and higher associativity. These approaches work effectively in many cases, however, cache conflicts occur frequently in some applications because the best cache configuration depends on application characteristics. In order to mitigate these conflicts, I propose a cache system on an FPGA soft processor which selects the suitable number of cache lines depending on an application. The proposed system uses Arbitrary Modulus Indexing (AMI) to implement non-power-of-2 cache lines and reduce cache conflicts.

To evaluate the performance of the proposed cache system, I propose and develop an IBM PC compatible SoC on an FPGA where hardware developers can evaluate their custom architectures. The SoC has an x86 soft core processor which can run general purpose operating systems. The proposed system runs on FPGAs of two major vendors, i.e. Xilinx and Altera, and the SoC is released as open-source. Therefore this SoC can be widely used for learning computer systems.

I evaluate the proposed cache system on the proposed SoC. In the evaluation, the proposed SoC is implemented on Terasic's Altera DE2-115 FPGA board, and boots Tiny Core 5.3, which is a distribution of Linux. The SPEC CPU2000 INT benchmark is used to evaluate

the system performance. I show that the proposed cache system with AMI-based functions reduces miss rate significantly with low hardware overhead.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An FPGA (Field Programmable Gate Array) is an integrated circuit on which developers can implement their custom circuits. A processor is often implemented on FPGAs in order to run applications, control the whole system, and so on. This processor can change their function with rewriting HDL source code like software, hence called a soft processor.

The design of soft processors can be modified more flexibly than conventional processors. Because of this versatility, soft processors are widely used for embedded applications. Soft processors generally use an external DRAM as the main memory. Since soft processors can run much faster than the external DRAM, they often contain a cache system to provide faster access speed. A cache system is implemented on the embedded memory in an FPGA. While the embedded memory support very fast access speed, its capacity is much less than the capacity of external DRAM. Therefore high performance cache with fewer resources for FPGA soft processors has been required.

Conventional approaches to improve cache performance are following: larger cache capacity and higher associativity. Although the larger cache capacity is the most stable approach to improve cache performance, the amount of required embedded memory significantly increases. The higher associativity, which improves the performance in many cases, increases the latency, and has the disadvantage of larger logic circuits. While these approaches may work effectively in many cases, cache conflicts occur frequently in some applications because the best cache configuration depends on application characteristics. Consequently, optimization with an application-specific cache is required.

In order to reduce conflicts, non-power-of-2 indexing functions have been studied [1] [2]. In general, a division circuit is costly to be implemented, and hence it has not been used for conventional cache systems. However, Diamond et al. proposed Arbitrary Modulus Indexing (AMI), which is a method to generate non-power-of-2 indexing functions with low hardware cost, and achieved the improvement of GPU performance [1].

I propose a cache system on an FPGA soft processor which selects the suitable number of cache lines depending on an application. The proposed system uses AMI to implement non-power-of-2 cache lines. The system can reduce cache conflicts, and thus improve the system performance.

To evaluate the performance of the proposed cache system, I propose and develop Frix, which is an IBM PC compatible SoC on an FPGA where hardware developers can evaluate their custom architectures. Frix has an x86 soft processor, and can boot general-purpose operating systems (NOT embedded OSs). Frix can be implemented on two major vendors' FPGA board: Altera DE2-115, Xilinx Nexys4 and Xilinx Nexys 4 DDR. The most parts of the system are written in Verilog HDL, and vendor-dependent IP cores are not used in the main function. Therefore this SoC can be widely used for learning computer systems or evaluating the system performance.

I evaluate the AMI-based cache systems on Frix. In the evaluation, Frix is implemented on Terasic's Altera DE2-115 FPGA board, and boots Tiny Core 5.3, which is a distribution of Linux. The SPEC CPU2000 INT benchmark is used to evaluate the system performance.

The rest of this papers is organized as follows, Chapter 2 describes the research background, Chapter 3 introduces my proposal: AMI-based cache system, Chapter 4 also introduces Frix, Chapter 5 represents the usability of Frix for computer research and the efficiency of AMI-based cache system, and finally Chapter 6 summarizes this paper.

Chapter 2

Background: Cache Architecture

This chapter describes research background. Overall cache architecture is detailed in Section 2.1. Section 2.2 specifies cache conflicts, and represents conventional approaches to reduce cache conflicts. Section 2.3 represents related work for reducing cache conflicts.

2.1 Overall Architecture

A cache is a memory system placed on a higher level of memory hierarchy in a processor. It has a set of memory blocks called cache lines, and data in the main memory are copied into cache lines. Since the behavior of caches is concealed, programmers can use the fast memory unconsciously.

Figure 2.1 shows an architecture of a cache with M cache lines. When a memory access occurs, its address is divided by M. The quotient and the remainder of division are called tag and index, respectively. The index, which gets the value from 0 to $M - 1$, defines a cache line in which the data will be stored. In order to restore the original address with index, the tag is stored in the cache line. Although a cache can be implemented with an easy division, however, the division circuit by M is still a costly implementation.

By contrast, when $M = 2^n$ and n is a positive number, a cache can be implemented with less hardware cost. Figure 2.2 shows an architecture of a cache with 2^n cache lines. In this implementation, the index and the tag are calculated with simple bit selection. Since the division by power-of-2 value is suitable for hardware, a conventional cache employs 2^n cache

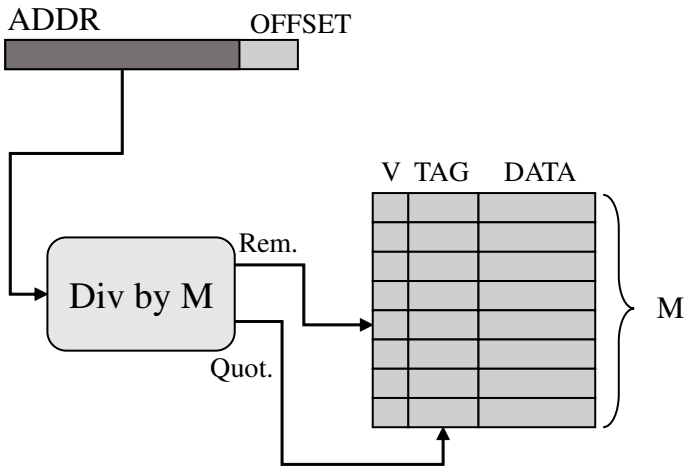


Figure 2.1: Cache architecture

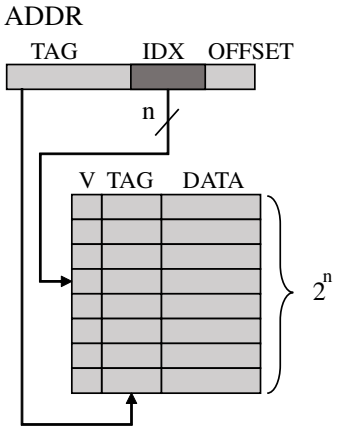


Figure 2.2: Conventional cache architecture

lines.

2.2 Cache Conflicts

When requested data is not found in a cache, the cache needs to fetch the data from the main memory. This is called a cache miss. In order to improve system performance, cache misses should be reduced.

Cache misses are classified into three categories: compulsory, capacity, and conflict. A compulsory miss occurs when a processor first requests data in the main memory. Since the first requested data is not stored in a cache at all, the compulsory miss occurs no matter how

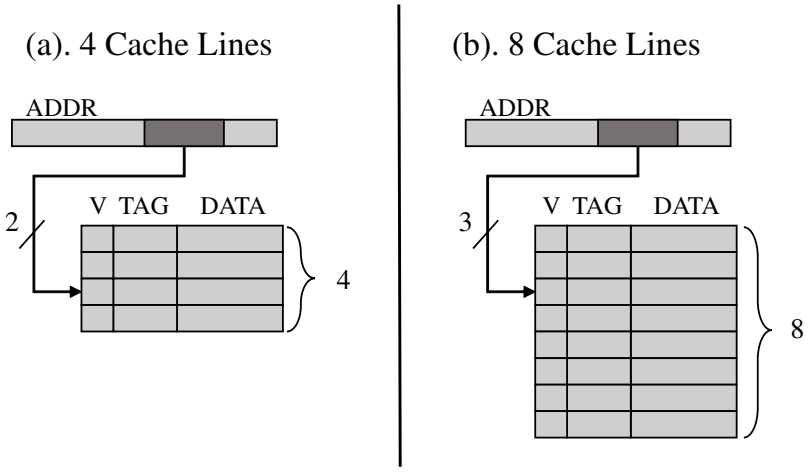


Figure 2.3: Larger capacity

well a cache is designed. A capacity miss occurs when cache capacity is less than the memory capacity needed to execute an application. In order to reduce the capacity miss, cache capacity needs to increase. A conflict miss occurs when two different data are stored in the same cache line. When a cache has 2^n cache lines, the same n-bit indices from two different addresses cause a conflict miss. This conflict of indices is called a cache conflict. The purpose of this research is to reduce cache conflicts for better system performance.

Larger cache capacity is one of the conventional approach to reduce cache conflicts. Figure 2.3 shows an example of the larger cache capacity. In this example, the number of cache lines increases from 4 to 8. The conflict caused by the same 2-bit indices may not occur with the 3-bit indices. Therefore, the larger cache capacity can reduce some conflicts.

Higher associativity is another conventional approach to reduce cache conflicts. In the simplest cache introduced in the Section 2.1, the address of a memory access determines a cache line where the data will be stored. The higher associativity increases candidates of the cache lines corresponding to the data, and contributes to reduce conflicts.

A cache with higher associativity is particularly called a set associative cache. A set associative cache divides cache lines into some different groups. A group of cache lines is called a way. When a memory access occurs, the address is divided by the number of cache lines on each way, and the index determines the corresponding cache lines. A group of these corresponding cache lines are called a set. In a set associative cache, each way has a corresponding

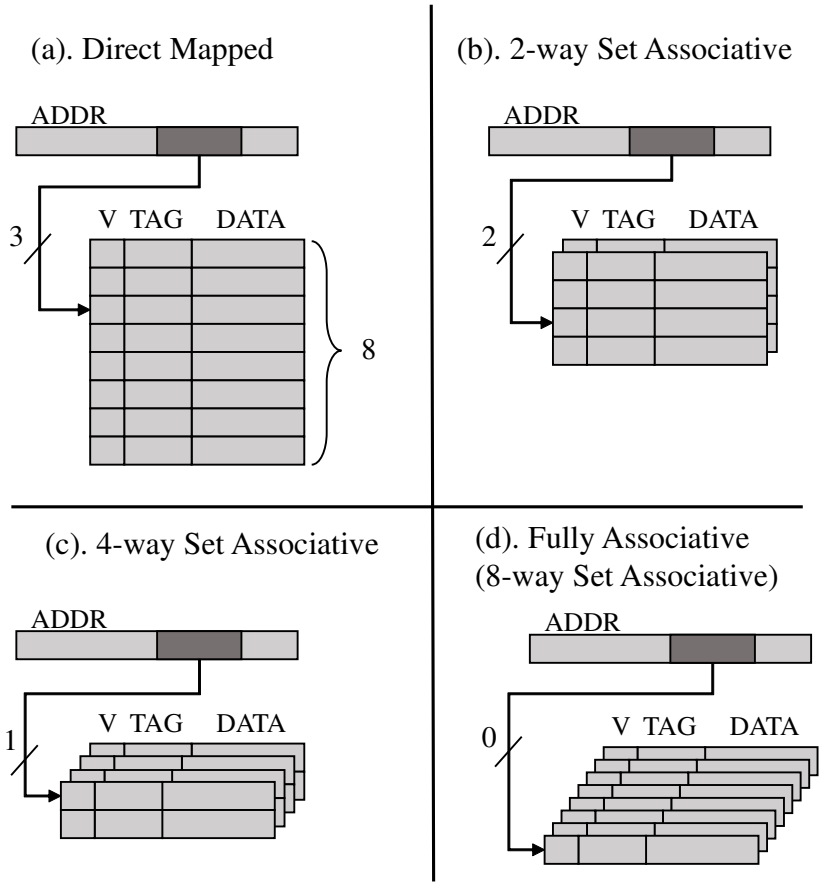


Figure 2.4: Higher associativity

cache line to the memory access. Hence, since the number of the candidates is the same as the number of ways, the higher associativity reduces conflicts.

Figure 2.4 shows four different associativities when the number of cache lines is fixed to 8. Associativities of (a), (b), (c) and (d) are 1, 2, 4 and 8, respectively. When associativity is 1, an address directly determines the corresponding cache line, and thus this cache is especially called a direct mapped cache. In a direct mapped cache, the cost of implementation is very low, but this cache tends to occur many conflicts. On the other hand, when the number of cache lines is the same as the associativity like (d), the cache is called a fully associative cache. In a fully associative cache, data can be stored in any cache lines, and hence there are almost no conflicts. However, the implementation cost is too hard. In a modern cache system in Intel Core i7, the associativity gets values from 4 to 16 [3]. The tradeoff between the performance and the implementation cost determines the associativity.

Though both of these approaches can reduce cache conflicts in many cases, however, cache conflicts occur frequently in some applications because the best cache configuration depends on application characteristics. Therefore optimization with an application-specific cache is required.

2.3 Related Work: Methods with Hash Functions

In order to reduce cache conflicts, methods to generate indices with hash functions have been studied. In general, cache index I of address A is calculated as $I = A \ mod \ D$. D means the number of cache lines. In the calculation with hash functions, the index is calculated as $I = H(A)$. H means a hash function which maps A to I, $(0 \leq I \leq D - 1)$. Many methods have been proposed such as a bit selection [4] and calculations with exclusive OR [5] or rotation [6]. A heuristic algorithm to determine hash functions is also proposed [7]. This algorithm uses memory traces of the application to determine a good hash function. Skewed-associative cache [5] and ZCache [8] are methods for set associative caches and these methods also use hash functions.

Though these approaches are effective, cache conflicts still occur frequently because a cache is composed of power-of-2 cache lines. Moreover, the suitable hash function also depends on application characteristics. In this research, I propose a cache system which selects the suitable number of cache lines to avoid more conflicts.

Chapter 3

Proposal: AMI-Based Cache System

In this chapter, I propose an AMI-based cache system. Section 3.1 describes my research background, called Arbitrary Modulus Indexing (AMI) [1]. AMI is a method to implement a cache with non-power-of-2 cache lines with low hardware cost. Section 3.2 describes the design of AMI-based cache systems.

3.1 Arbitrary Modulus Indexing

Arbitrary Modulus Indexing (AMI) [1] is a method to implement indexing with arbitrary moduli. The basic idea of AMI is the multiplication by the reciprocal, instead of calculating the division. This calculation is easily integrated into circuits. Furthermore, circuit implementation with AMI has the advantage of low hardware cost.

I demonstrate generating an indexing function with AMI for a cache system with 255 cache lines. When the address $A = abcd_{256}$ is given, in order to compute $A/255$, AMI multiplies A by the constant $1/255$. The constant $1/255$ can be expressed as follows:

$$\begin{aligned} 1/255 &= 1/11111111_2 \\ &= 0.0000000100000001..._2 \\ &= 0.11..._{256} \\ &= 0.\overline{1}_{256} \end{aligned}$$

```
A = 0xabcdefff
---------------------------
    ab + cd = 178   …(1)
1 + 78 + ef = 168   …(2)
1 + 68 + ff = 168   …(3)
1 + 68      =  69   …(4)
---------------------------
0xabcdefff mod 255 = 0x69
```

Figure 3.1: Calculation of 0xabcdefff mod 255

Then, AMI multiplies A by $1/255$. This can also be expressed as follows:

$$\begin{aligned} A \times (1/255) &= abcd_{256} \times 0.\overline{1}_{256} \\ &= abc.d + ab.cd + a.bcd + ... \\ &= aaa.a... + bb.b... + c.c... + 0.d... \\ &= aaa.\overline{a} + bb.\overline{b} + c.\overline{c} + 0.\overline{d} \\ &= DIV.\overline{R} \end{aligned}$$

The cache tag (TAG) is obtained by floor($DIV.\overline{R}$) = DIV, and the cache index (IDX) is also obtained by frac($DIV.\overline{R}$) $\times$ 255 = $0.\overline{R}/(1/255)$ = $0.\overline{R}/(0.\overline{1})$ = R. Hence, with the circuit calculating $DIV.\overline{R}$, the indexing function for the cache system with 255 cache lines can be implemented.

The circuit calculating $DIV.\overline{R}$ is implemented as several adders but needs carries to be connected properly. Since the $0.\overline{R}$ is equal to $0.\overline{a}+0.\overline{b}+0.\overline{c}+0.\overline{d}$, the adders calculate $a+b+c+d$. In this case, the a and b are 8-bit values, and accordingly the result of $a + b$ is 8-bit sum and 1-bit carry. The special calculation to obtain the $0.\overline{R}$ is only to use the output carry of the previous addition as the next input carry.

Figure 3.1 shows the calculation of $0.\overline{R}$ when the address A is given as A = 0xabcdefff. When A is denoted by $A = abcd_{256}$, the a, b, c and d equals to 0xab, 0xcd, 0xef and 0xff, respectively. Firstly, the sum of a = 0xab and b = 0xcd is calculated, and accordingly the result 0x178 is given, as shown in Figure 3.1.(1). Secondly, the 8-bit sum of previous result

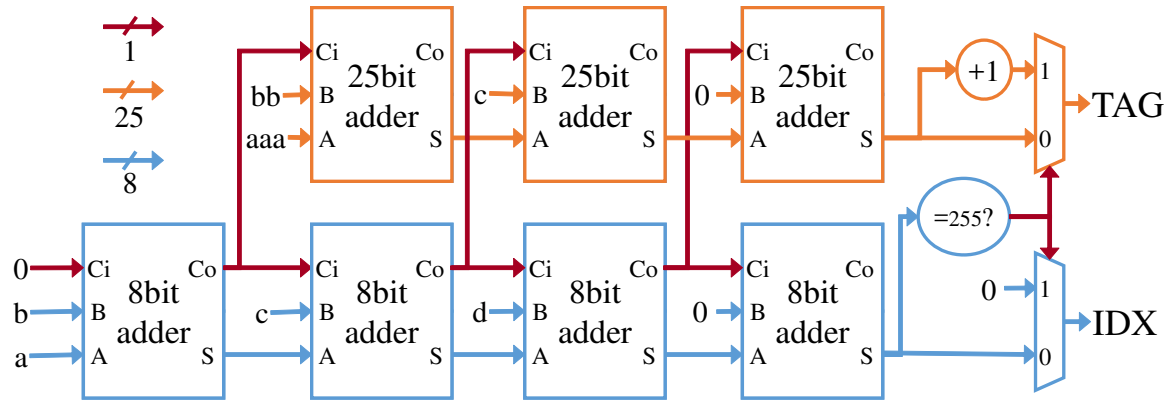


Figure 3.2: A generator of tag and index for a cache with 255 cache lines

0x78 is added to c. In this case, since the output carry of the previous addition is 1, the value 1 is used as the input carry (Figure 3.1.(2)). Thirdly, the previous result is also added to d (Figure 3.1.(3)). Finally, the output carry of the previous addition is added to the previous result 0x68 (Figure 3.1.(4)). This calculation does not generate a carry because the maximum result in (3) is 0x1fe. To calculate $TAG = aaa + bb + c$, the correct result is given with adding the output carries of (1), (2) and (3) to $aaa + bb + c$.

Figure 3.2 shows a generator of tag and index for a cache with 255 cache lines. In this figure, there are four 8-bit adders and three 25-bit adders. The output carries of the previous additions are connected to the input carries of the next additions, and thereby the TAG and IDX are obtained. Since this circuit is implemented as a cascade of several adders without complex division nor multiplier circuit, the hardware cost of AMI-based indexing is low. There is a circuit which sets IDX to zero and increments TAG when $IDX = 255$, because the simple implementation of AMI generates the wrong value $\{TAG, IDX\} = \{X - 1, 255\}$ even when the correct value is $\{TAG, IDX\} = \{X, 0\}$.

While the tag and index can be calculated with the above expression, caches need to restore the original address for writing back the data to the main memory. The original address A is obtained by $A = TAG \times 255 + IDX$. Since the multiplication by non-power-of-2 value is a costly implementation, it is also a difficult problem to get the original address A. However, in this case, the address can be calculated with a simple subtraction. The original address A is

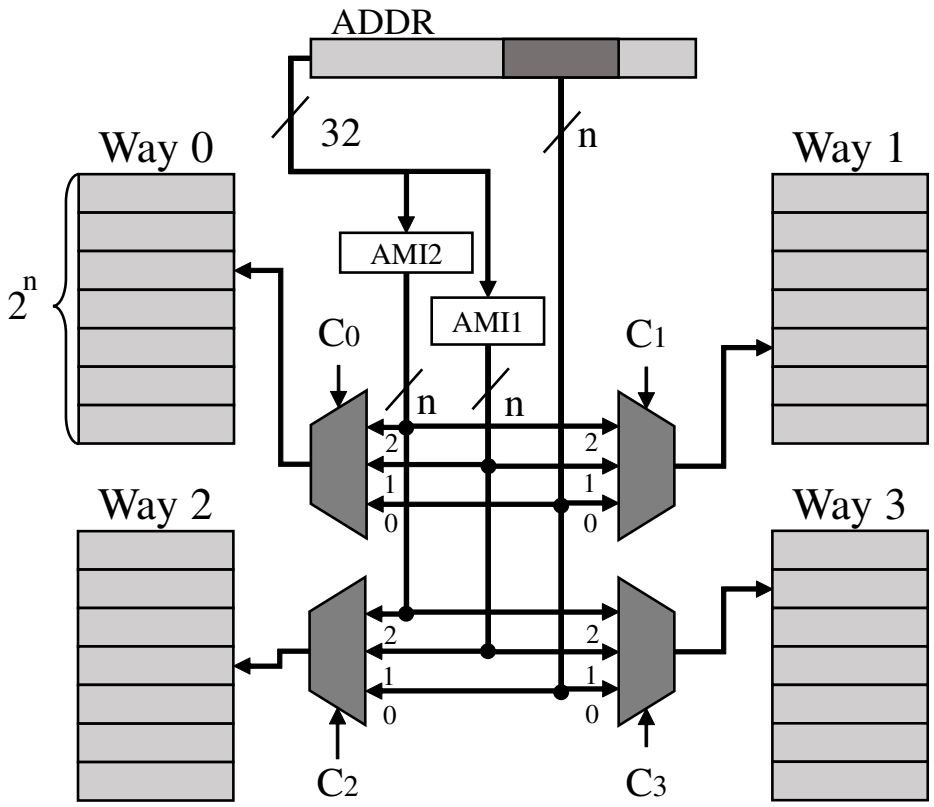


Figure 3.3: Proposed AMI-based cache system for 4-way set associative cache

obtained by the following expression:

$$A = (TAG \ll 8) - TAG + IDX$$

$TAG \ll 8$ means 8-bit left shift of TAG, and $TAG \ll 8$ is the same as $TAG \times 256$. Since hardware cost of a subtraction circuit is much smaller than that of the multiplication by non-power-of-2 value, the overhead of this circuit is also small.

3.2 AMI-Based Cache System

3.2.1 Overall Design

I propose an AMI-based cache system which selects the suitable number of cache lines depending on an application. Figure 3.3 shows the proposed system for a 4-way set associative

Table 3.1: Example: Execution Time

(C_0, C_1, C_2, C_3)	(0, 0, 0, 0)	(1, 1, 1, 1)	(2, 2, 2, 2)
app0	3600 (sec)	3800 (sec)	3200 (sec)
app1	7200 (sec)	6200 (sec)	8200 (sec)

cache. This cache has 2^n cache lines per way. In addition to conventional indexing function, two AMI-based indexing functions for $2^n - 1$ and $2^n - 2$ cache lines are implemented. Multiplexers select an index from these 3 indices. The control wires: C_0, C_1, C_2 and C_3 can be changed for the running application. In this figure, these multiplexers select 2^n indexing when $C_i = 0$, AMI-based $2^n - 1$ indexing when $C_i = 1$ and AMI-based $2^n - 2$ indexing when $C_i = 2$. In this way, suitable cache configuration is selected by C_0, C_1, C_2 and C_3.

When the index by AMI-based indexing function is utilized, the tag field needs extra 1-bit in the cache line. This is because the maximum value of tags in $2^n - 1$ or $2^n - 2$ indexing is larger than that in 2^n indexing, and the tag field needs more digits for expressing and storing the tag. When D, denoting a number of cache lines, satisfies $2^{n-1} \leq D < 2^n$, the tag field in D indexing requires more 1 bit than that in 2^n indexing. Hence, adding 1-bit can allow the cache to use any D indexing satisfying $2^{n-1} \leq D < 2^n$. In order to utilize $2^n - 1$ or $2^n - 2$ indexing anytime, the 1-bit is added to the tag field in all cache lines.

In this paper, two AMI-based indexing functions, $2^n - 1$ and $2^n - 2$, are utilized. One reason is the number of unused cache lines. In the proposed system, 2^n cache lines are implemented. Consequently, $2^n - 1$ and $2^n - 2$ cache lines are the best implementations which minimize the unused cache lines.

Another reason is that these functions are easy to implement. When the number of cache lines is $2^n - 1$, indexing functions are easily implemented. Furthermore, when implementing an indexing function for $2^m(2^n - 1)$ cache lines, the indexing function can be implemented with an easy combination of $2^n - 1$ indexing and simple shift circuits. On the other hand, for the other number of cache lines, the implementation of indexing functions is more difficult and more costly because AMI-based indexing functions become larger and more complex when binary notation of $1/D$ is not $0.\overline{1}_x$. Therefore caches with $2^n - 1$ and $2^n - 2$ cache lines are the better implementation.

The proposed cache system selects the suitable indexing functions depending on a running application. In order to select the best indexing functions, the execution times in each configuration are firstly measured, as shown in Table 3.1. Then, the result show that the best configuration for app0 is $(C_0, C_1, C_2, C_3) = (2, 2, 2, 2)$ and the one for app1 is $(1, 1, 1, 1)$. After selecting the suitable indexing functions, the configuration is fixed while running the application.

The control wires C_0, C_1, C_2 and C_3 are the input to the cache from a soft processor or out-FPGA switches. In order to ensure the coherence of data, before using different indexing functions, the data in the cache needs to write back to the main memory. This writing back function does not need a special hardware when a processor supports that function such as "INVD" instruction on x86 architecture.

3.2.2 Skewed Indexing function

Skewed associative cache [5] is a method to improve cache performance. The skewed associative cache uses hash functions to determine the cache index and therefore different indices are chosen in each way. Since a normal cache uses low address bits as the cache index, when memory access which has the same low address bits successively occurs, it results in many cache conflicts. By using skewed cache with proper hash functions, it has less incidence of cache conflict even if the memory accesses have the same successive low address bits because each address has a different cache index due to the hash function.

In the proposed system, C_0, C_1, C_2 and C_3 are not always the same. The example of $(C_0, C_1, C_2, C_3) = (0, 0, 1, 2)$ is shown in Figure 3.4. Each value means the number of unused cache lines. There is an unused cache line in the way 2, and there are two unused cache lines in the way 3. Hence, the numbers of cache lines on each way are different in this configuration. This cache can be considered as the expansion of skewed-associative cache with AMI, and thus this technique can reduce more cache conflicts.

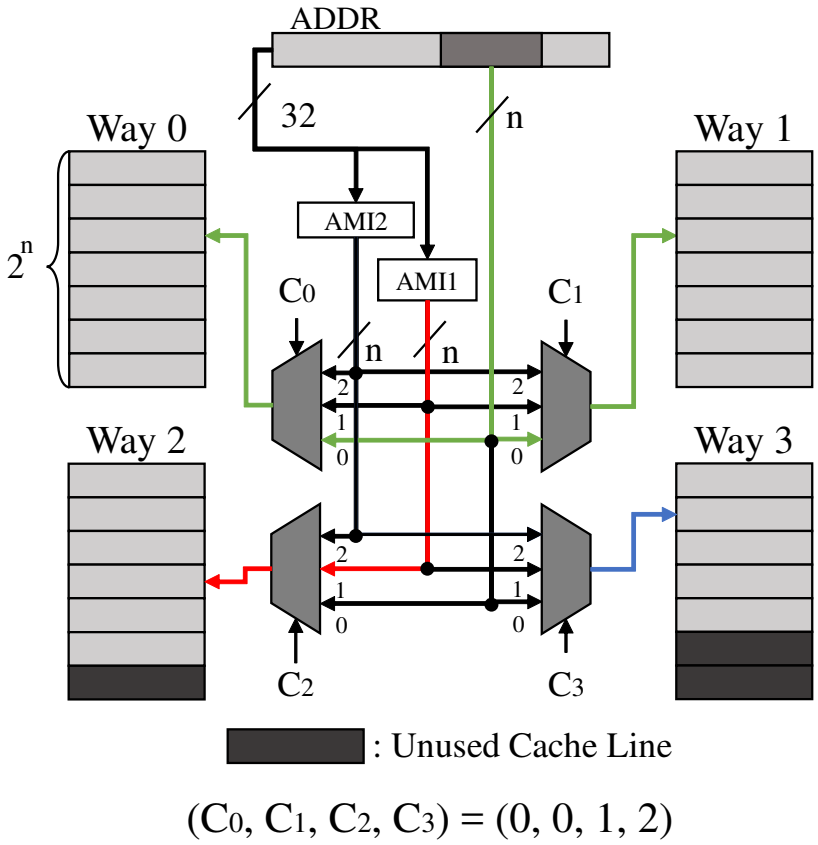


Figure 3.4: Different indexing functions are selected when $(C_0, C_1, C_2, C_3) = (0, 0, 1, 2)$

3.2.3 Replacement Policy

When a cache miss occurs, the corresponding cache line fetchs the desired data from the main memory. Before storing the desired data, the cache line evicts the existing data. In a set associative cache, there are several candidates of cache lines corresponding to a data. Hence, the cache needs to select the cache line which evicts the existing data.

Replacement policy defines which data on a set are replaced with the requested data under a cache miss. Since the better choice of the evicted data can reduce cache misses, the role of replacement policy is important. LRU (Least Recently Used) replacement policy is the most common replacement policy on microprocessors. However, the LRU cannot be used in the proposed cache because of its data management. This section describes the problem of LRU on the proposed system and the alternative replacement policy, named NRU (Not Recently Used) [9].

LRU replacement policy does not work in skewed associative caches. The LRU manages the accessing orders of cache lines on a set. In a conventional cache, a cache line belongs to a fixed set, and accordingly the number of sets is the same as the number of cache lines on each way. Thus, assuming N as the number of cache lines on the cache and W as associativity, the number of total sets on a cache is N/W. In a skewed associative cache, however, the number of total sets on the cache is $(N/W)^W$ because a cache line can belong to multiple sets. Consequently, the LRU replacement policy in a skewed associative cache needs too much resources.

NRU (Not Recently Used) is a replacement policy, employed in some high performance processors [9] [10]. In the NRU replacement policy, a NRU-bit on a cache line represents whether the cache line is recently accessed or not. Since the NRU manages the information not on a set but on a cache line, it does not need more information even in a skewed associative cache.

When the NRU-bit is low, the bit shows that the cache line is recently accessed. NRU replacement policy updates the NRU-bit to low on each access, and updates the all NRU-bits in a set to high when the all NRU-bits in the set are low. The algorithm of NRU is as follows:

Algorithm of NRU

```
Cache Hit:
    (1) set nru-bit of the line to '0'
Cache Miss:
    (1) search for '1' bit on the set from Way 0
    (2) if '1' bit is found then goto (5)
    (3) set all nru-bit of the set to '1'
    (4) goto (1)
    (5) set nru-bit of the line to '0'
```

Figure 3.5 shows a behavior of LRU and NRU. In LRU, the accessing order is managed on a set. The order "0123" means the cache line on way0 is the least recently used cache line. Hence, when an access "a_4" occurs, the cache line on way0 is updated and the accessing order is changed to "1230". By contrast, NRU manages a NRU-bit on a cache line. The NRU-bit on

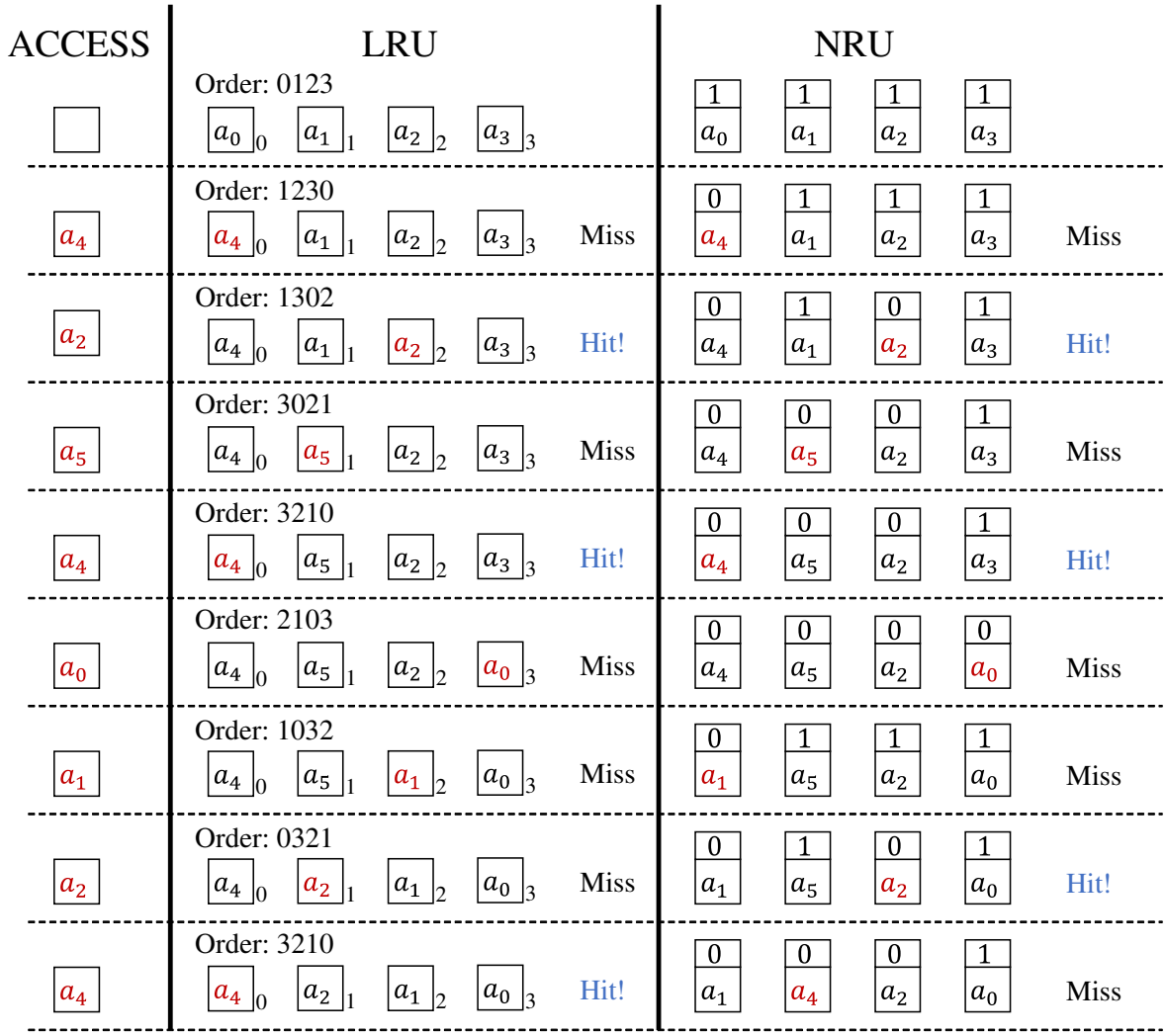


Figure 3.5: Behavior of LRU (Least Recently Used) and NRU (Not Recently Used)

way0 is set to low under the access "a_4", and the cache line on way0 is also updated.

Chapter 4

Evaluation Environment: Frix

In order to evaluate the performance of the cache proposed in Chapter 3, I propose a feasible and reconfigurable IBM PC Compatible SoC (Frix). Since this SoC can run general purpose OSs and therefore general purpose benchmarks, users can implement and evaluate their architecture on Frix. In order to design a commonly-used system, I develop Frix to satisfy the following three conditions: the SoC can be implemented on both Altera's FPGA and Xilinx's FPGA, the corresponding FPGA board is a commercially-available FPGA board, and the HDL source code of Frix is released as open-source.

4.1 Frix: Feasible and Reconfigurable IBM PC Compatible SoC

I propose a SoC to fulfill the following two requirements.

One is to be a suitable environment where developers can efficiently evaluate their architectural ideas to improve performance of computer systems. The SoC environment allows developers to easily change its hardware configuration and evaluate its performance on a general purpose OS.

The other is to be a suitable one where learners can easily understand how a computer system works. To realize this, complicated hardware components are replaced with simple ones.

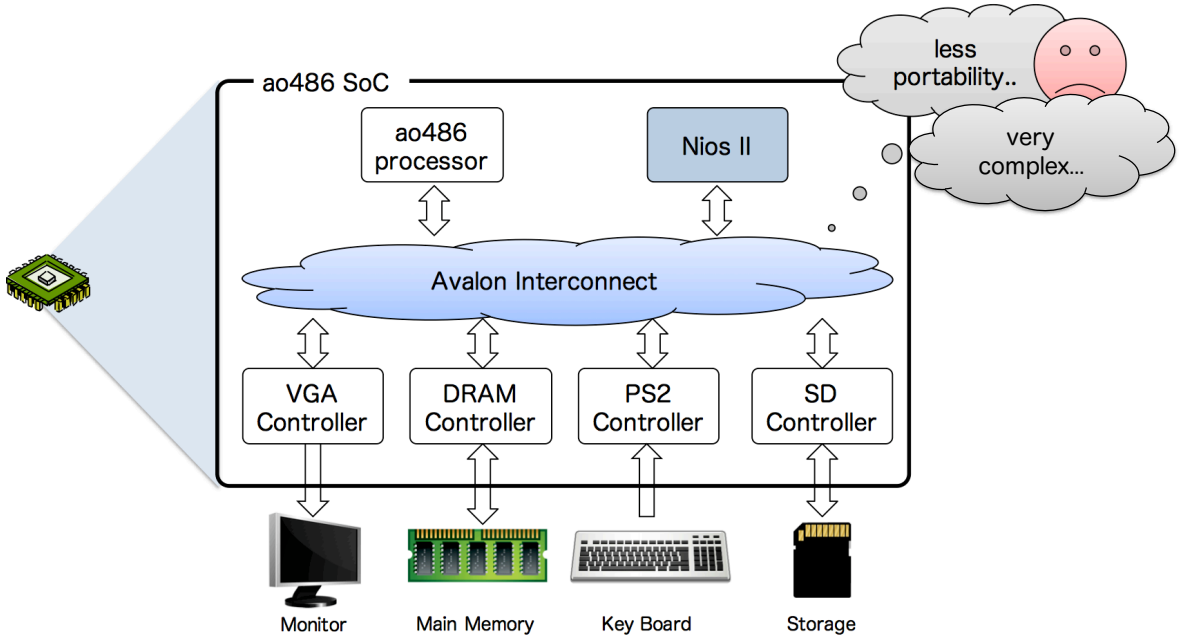


Figure 4.1: The design overview of ao486 SoC

From next section, the design and implementation of Frix is explained, and the accurate run of Frix is demonstrated in detail.

4.1.1 Design of Frix

To realize a suitable environment for research and education, Frix is based on ao486 SoC [11]. Ao486 SoC has a 32-bit x86 compatible soft core processor based on the standards of Intel 80486 SX, and several device controllers — PS/2 controller, HDD controller, VGA controller, etc. This SoC runs on a Terasic's Altera DE2-115 FPGA board and according to the author's document, it can boot the Linux kernel version 3.13 and Windows 95. Since ao486 SoC can boot a general purpose OS, it can be used to precisely evaluate target hardware by running widely used benchmark such as SPEC.

Figure 4.1 shows the design overview of ao486 SoC. It has an x86 processor named ao486 processor, a Nios Ⅱ soft processor, and several device controllers. The Nios Ⅱ processor handles BIOS loader. This function will be described in Section 4.1.2. All these components are connected to Altera's IP core named Avalon interconnect (a bus system), and communicate each other via this interconnect. Since ao486 SoC uses Altera's IP cores as a basic function

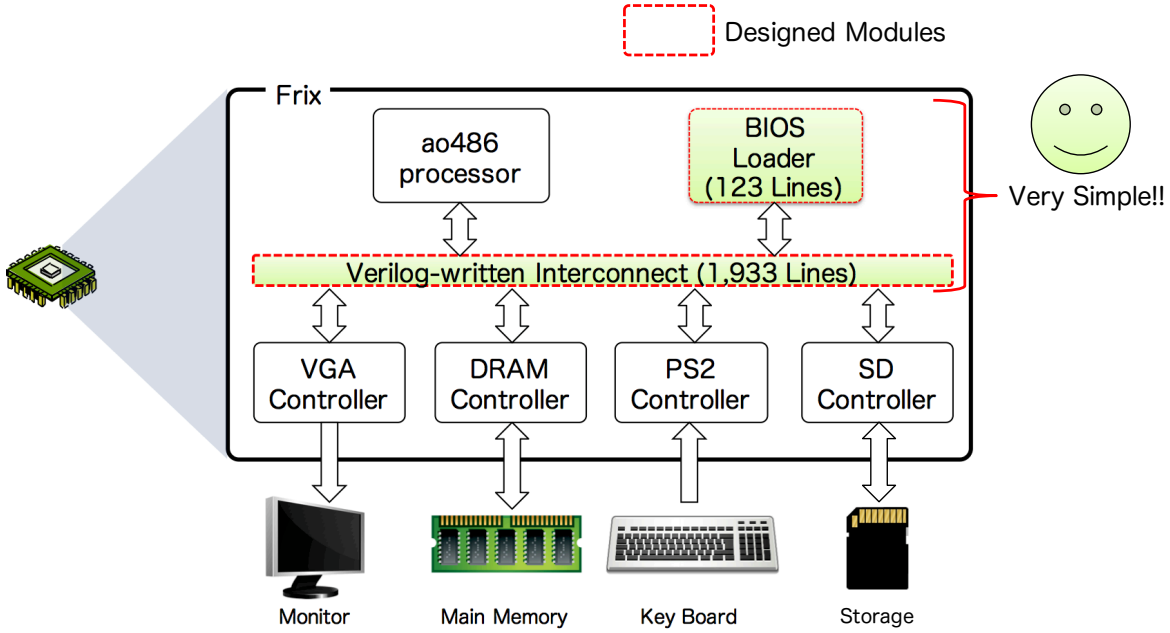


Figure 4.2: The design overview of Frix

of a computer such as a bus system, it can work only on an Altera's FPGA. Furthermore, because the IP cores' insides are abstracted, it is difficult for learners to understand how the SoC operates. To address these problems, some modules of ao486 SoC are modified and some new modules are added to the proposed SoC. Thus, the proposed SoC is simpler and more portable so that it can work on other FPGAs.

4.1.2 Implementation of Frix

Figure 4.2 shows the design overview of Frix. The newly designed hardwares are depicted as red rectangles. Altera dependent modules (Nios II and Avalon Interconnect) are replaced with simple substitute modules in Verilog HDL. Thus, the design of Frix is very simple.

In general, when an x86 PC is booted, processor's instruction pointer is reset to FFFF0h where the jump instruction to the head of a BIOS program is stored. BIOS programs are usually stored in a non-volatile ROM on a motherboard and therefore the jump instruction to a BIOS region is actually an access to the ROM. Instead of a ROM, ao486 SoC uses an SD card to store a BIOS program. Once the power supply switch is turned on, Nios II loads the BIOS data from the SD card and stores in DRAM region whose address range has been preserved

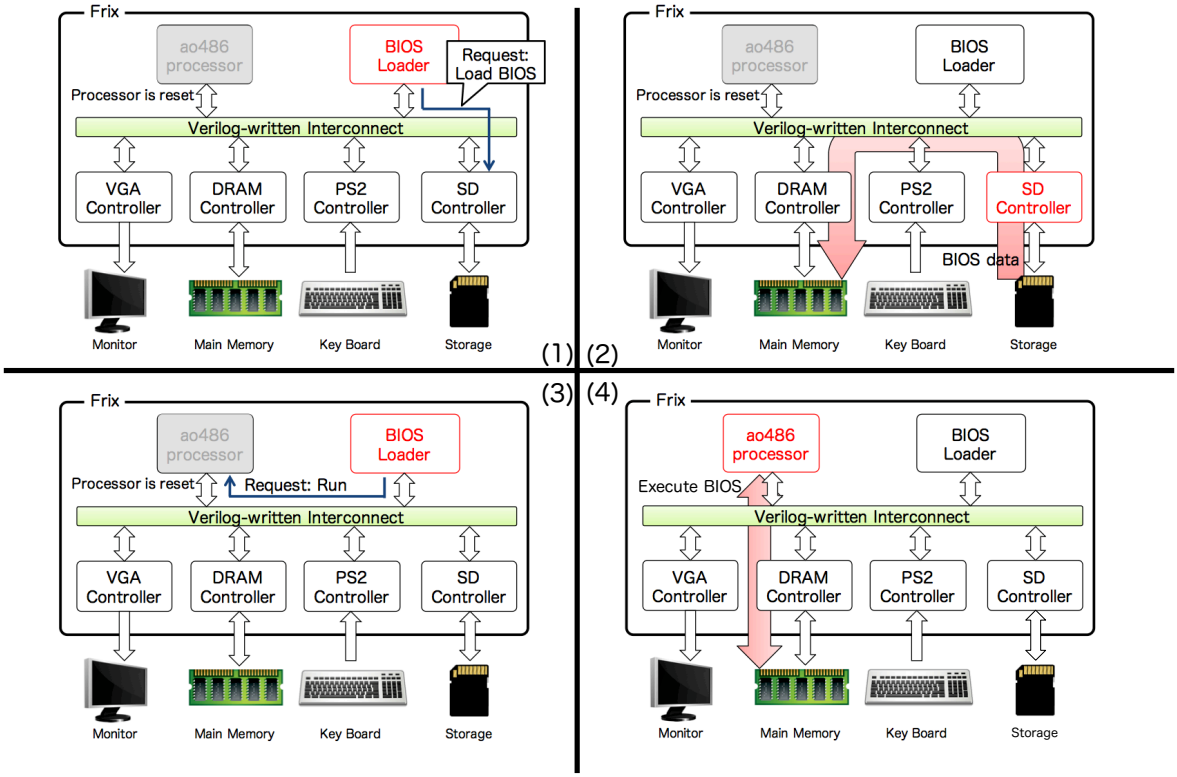


Figure 4.3: The system boot sequence with BIOS loader module. (1) The loader sends several signals. (2) The BIOS data is moved to DRAM. (3) The ao486 processor wakes up. (4) The processor executes BIOS and programs.

for the BIOS program.

To achieve high portability, a BIOS loader module in Verilog HDL is added so that Nios I I is removed. Figure 4.3 shows the SoC boot process with the BIOS loader module. Firstly, the BIOS loader module sends several signals to the SD controller in order to load a BIOS program (Figure 4.3 (1)). Secondly, the SD controller loads the BIOS data from an SD card to a DRAM (Figure 4.3 (2)). Thirdly, after the BIOS load is done, the BIOS loader module awakes ao486 processor (Figure 4.3 (3)). Finally, ao486 processor begins to execute the BIOS program (Figure 4.3 (4)).

This BIOS loader module is written only in 123 lines Verilog HDL, while Nios Ⅱ is written in 1395 lines Verilog HDL. In other words, the number of lines of the BIOS loader module is about one-tenth of that of Nios Ⅱ.

Ao486 SoC uses Altera's Avalon Interconnect as the bus system. All of the embedded hard-

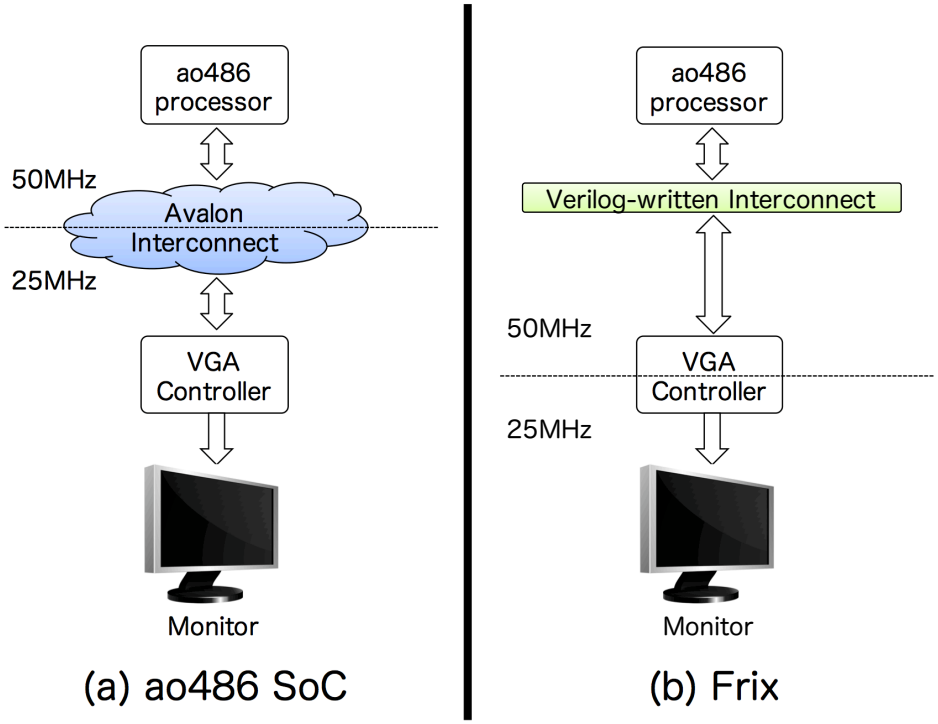


Figure 4.4: The clock domains of original and the new design. (a) In original design, the Avalon Interconnect has complex function converting 50MHz to 25MHz. (b) In the new design, Verilog written interconnect is simple running at 50MHz.

ware components communicate each other via this IP. In this SoC, this Avalon Interconnect mainly plays three roles.

The first one is to deal with the different data widths between the processor and the peripheral modules. From the processor to the peripherals, the Avalon Interconnect breaks down 4-byte (1-word) data from the processor into four 1-byte data, and then sends them to the peripherals. From the peripherals to the processor, this IP packs 1-byte data from peripheral modules into one 4-byte data, and then sends it to the processor.

The second one is to handle a burst request from the processor. The burst request is that the processor loads/stores multiple data from/in a memory. Memory addresses of these data are consecutive, whose range is from 1-word to 4-word. For example, if the processor loads consecutive 3-word data, the processor sends a request, which includes an initial address and the number of words to be loaded, to the interconnect. After receiving the request, the interconnect turns the request into three load instructions, and then loads data from the memory

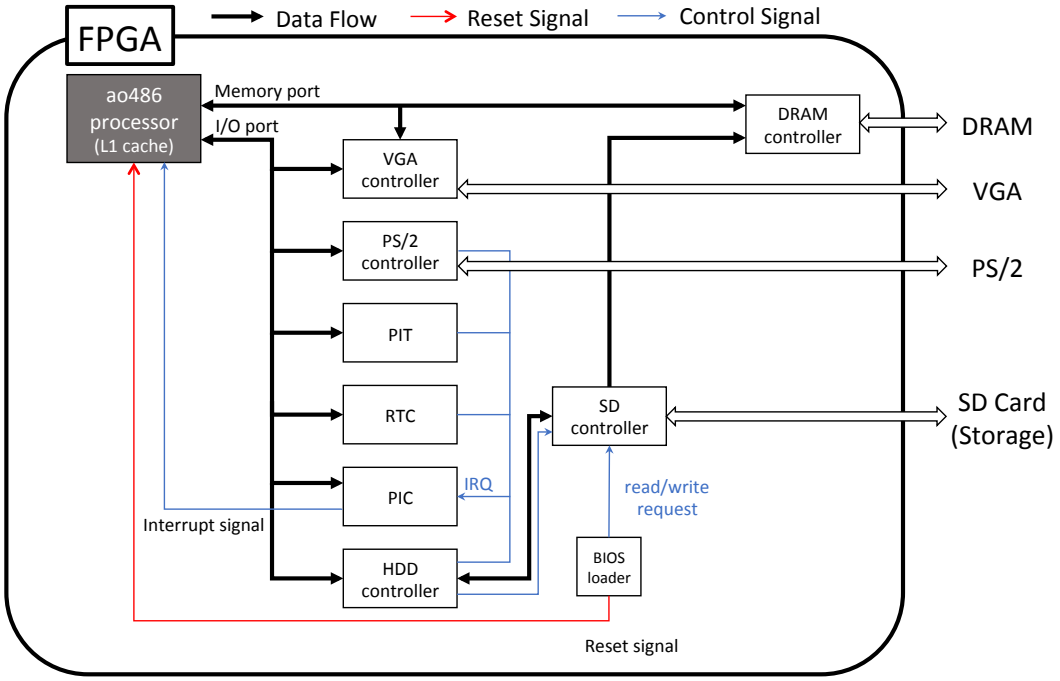


Figure 4.5: The block diagrams of Frix.

1-word by 1-word.

The last one is clock control for a monitor connected to the SoC. For the monitor, the VGA controller's frequency is set as 25MHz while the SoC's clock frequency is 50MHz. To communicate between the processor and the VGA controller, the interconnect handles two clock domains.

The Avalon Interconnect's feature is abstracted and systems with this interconnect can be synthesized by only Qsys [12]. This causes two disadvantages. One is that developers have to use Altera's FPGAs. The other is that it is difficult for learners to understand how the SoC works. To address these problems, a substitute bus system module, which realizes the three functions, is implemented so that the Avalon Interconnect is removed.

For first and second functions, the substitute logics are implemented with Verilog HDL. For the clock control, a new VGA controller with an asynchronous RAM is also implemented with Verilog HDL. Figure 4.4 shows the clock domain difference between the original and new design. Between the monitor and the output of the VGA controller, the operating frequency is set as 25MHz. For the other side, that is set as 50MHz. This new bus system is written

in 1,933 lines of Verilog HDL. Since the Avalon Interconnect is written in 97,562 lines of Verilog HDL, the number of lines of the new bus system is one-fiftieth of that of the Avalon Interconnect.

By replacing the vendor dependent IP cores such as Nios II or Avalon Interconnect, Frix can be used on other vendor's FPGAs rather than only Altera's FPGAs. However, since each FPGA board uses different peripherals (especially DRAM), developers have to consider their specifications.

Figure 4.5 shows the block diagrams of Frix, which is compatible with IBM PC. In addition to ao486 processor, Frix includes peripheral controllers in Verilog HDL such as VGA Controller, Intel 8042 Compatible PS/2 Controller, Intel 8254 Compatible PIT (Programmable Interval Timer), RTC (Real Time Clock), Intel 8259 Compatible PIC (Programmable Interrupt Controller), and HDD (HardDisk Drive) Controller. Frix uses an SD card as a virtual HDD. SD controller is the module that controls the writing to and reading from a SD card. HDD controller signals SD controller when the processor's load/store requests from/in hard disk are executed. By doing this, an operating system recognizes the SD card as HDD. PIC module handles interrupt requests (IRQ) from PS/2, PIT, RTC and HDD module, and sends an interrupt signal to ao486 processor. As mentioned above, the reset signal is active until finishing to send a BIOS program from the SD card to DRAM.

4.1.3 Verification of Frix

In addition to DE2-115 FPGA board, Frix can work on the Digilent Nexys4 DDR FPGA board which has a Xilinx's Artix-7 FPGA, by modifying DRAM controller. It is confirmed that Frix works properly and can boot general purpose OSs: FreeDOS 1.1 and Tiny Core 5.3 (Linux kernel 3.8). As described in the previous section, the OS image file and the BIOS data are stored in the specified locations of the SD card in advance.

On FreeDOS 1.1, various applications including games such as DOOM can run. Besides, by installing compilers, programs can be executed on FreeDOS. Figure 4.6 shows a snapshot of DOOM, which is a computer game developed by Id Software, running on the Digilent Nexys4 DDR FPGA board. Figure 4.7 shows the compilation and execution result of a Hello

Figure 4.6: The snapshot of DOOM running on Nexys4 DDR FPGA board

World program with Open Watcom C Compiler. The above program is written with an editor on FreeDOS. On Frix, users can enjoy software programming. Figure 4.8 shows other applications running on FreeDOS. Frix can run complex applications.

It is also verified that Frix works properly and can boot Tiny Core 5.3. Figure 4.9 shows the execution result of the "top" command on Tiny Core 5.3 on a Terasic's Altera DE2-115 FPGA board. Some daemon processes can be seen, as shown in the figure. Figure 4.10 shows the example of "echo" commands after booting Tiny Core 5.3. These commands set color codes of the terminal. Besides, the SoC can execute the SPEC CPU2000 INT benchmark suite on Tiny Core 5.3. The SPEC CPU2000 INT execution binary are inserted into the image file of Tiny Core. The source code of SPEC CPU2000 INT is compiled with gcc 4.8.2 (an option "-m32 -march=i486"). Since the proposed SoC architecture is x86 and many general purpose computers have an x86 processor, no special cross compiler is required in most cases. The execution result files of the benchmark are compared with the correct result files, in order to confirm that Frix accurately executes the benchmark.

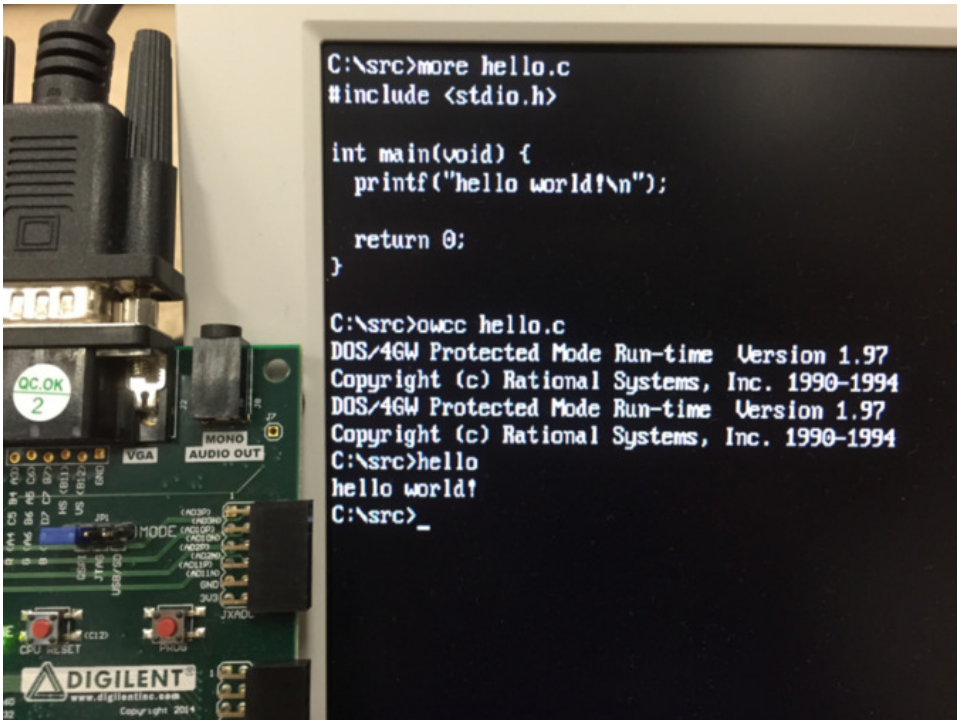


Figure 4.7: The snapshot of "Hello World!" execution result

Figure 4.8: The snapshot of applications running on FreeDOS

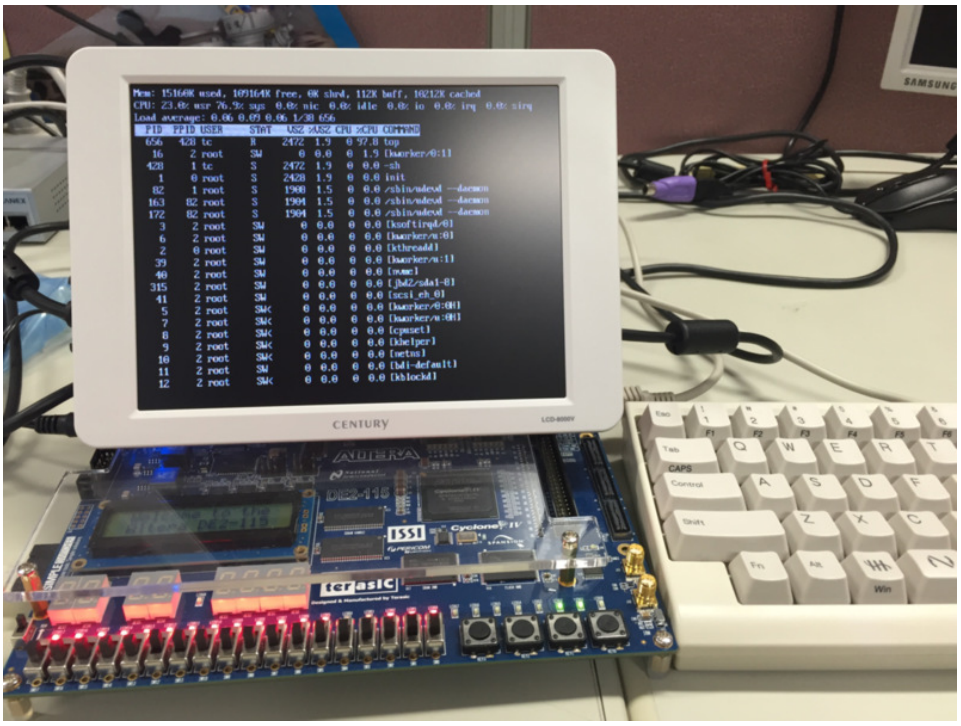


Figure 4.9: The snapshot of Linux (Tiny Core) running “top” command on DE2-115 FPGA board

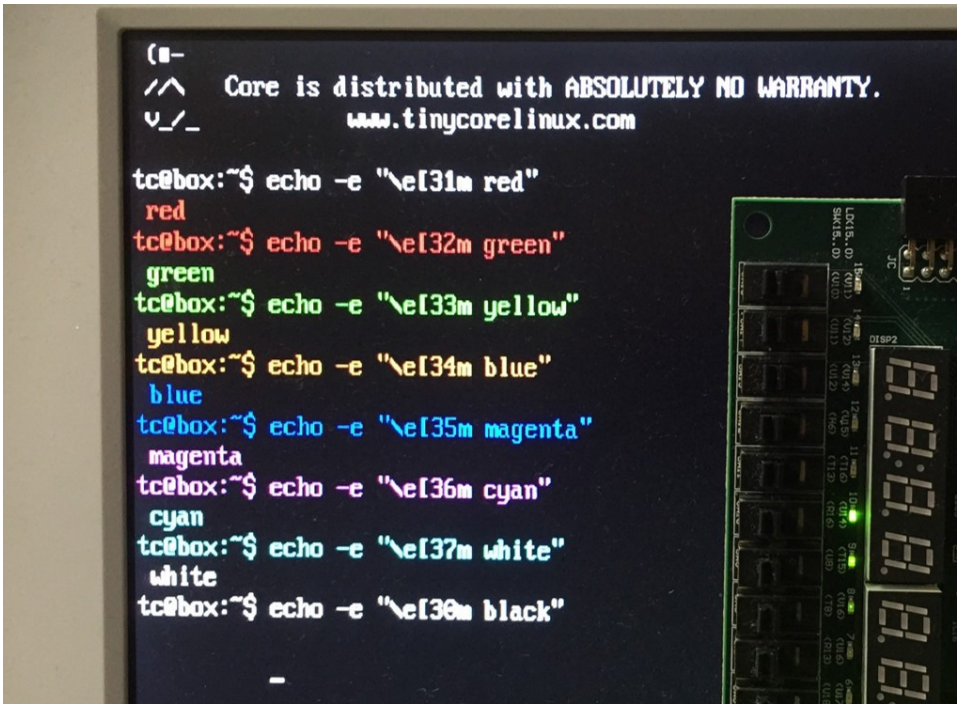


Figure 4.10: The snapshot of “echo” commands after booting Tiny Core 5.3

Chapter 5

Evaluation and Discussion

In this chapter, I evaluate the usability of Frix for computer research, and the efficiency of AMI-based cache system.

5.1 Usability of Frix for Computer Research

As a case study to demonstrate the usability of Frix for computer research, a data cache with non-power-of-2 cache lines is implemented in ao486 processor. In this section, I show that Frix can evaluate the cache performance with SPEC CPU2000 INT benchmark and hardware resources.

5.1.1 Implementation

The configuration of a data cache in ao486 processor is as follows:

- Capacity: 16KB
- Associativity: 4-way set associative
- Line size: 16B
- Write policy: writeback
- Replacement policy: NRU

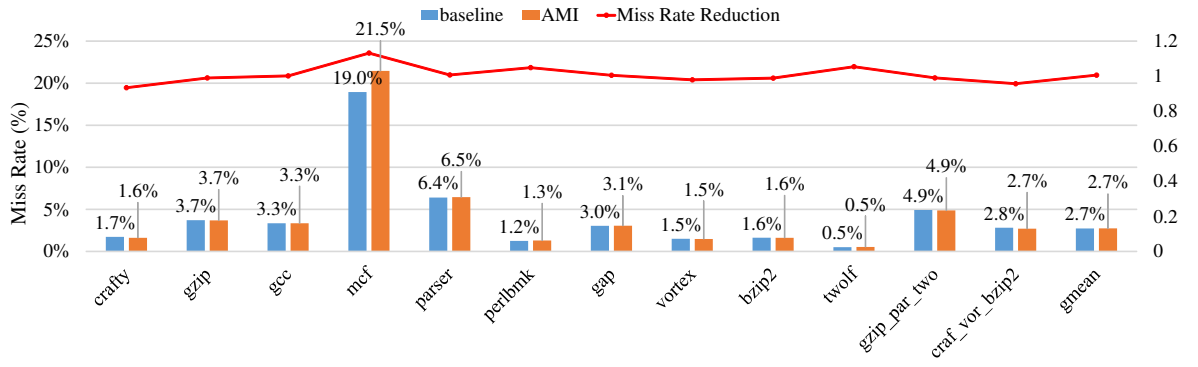


Figure 5.1: Execution time (sec) and speedup

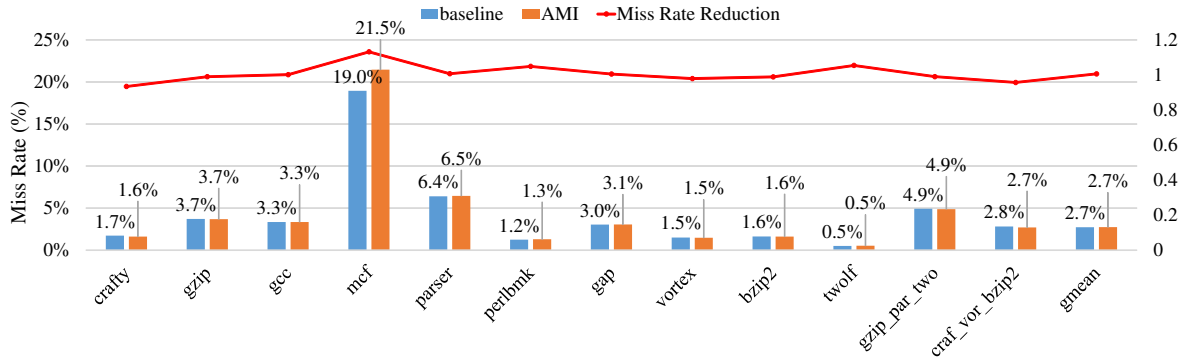


Figure 5.2: Miss rate and reduction

- # of cache lines per way: 256

The replacement policy of the original data cache is pseudo LRU, but changed to the NRU in order to evaluate the effect of AMI. This data cache is modified to the AMI-based cache, and the number of cache lines per way is fixed to 255. Thus, both of the skewed indexing functions and the selection of indexing functions are not implemented. In this evaluation, the AMI-based cache is compared with the original data cache.

5.1.2 Evaluation

For evaluation, the design is synthesized with Quartus Prime 15.1 and implemented on a Terasic's Altera DE2-115 FPGA board. The system frequency is set to 40MHz.

In order to measure time and miss rate, a subset of SPEC CPU2000 INT benchmark suite is executed on Tiny Core 5.3. This subset includes ten programs: gzip, gcc, mcf, crafty, parser,

Table 5.1: Hardware Resource

	Logic Cells		Memory (Kbits)	
	dcache	total	dcache	total
baseline	4520	54559	154.624	2585.856
AMI	5067	55206	155.648	2586.880
AMI/baseline	1.206	1.011	1.007	1.000

perlbmk, gap, vortex, bzip2 and twolf. Each application is compiled with gcc 4.8.2 compiler using the -O2 flag, and executed three times. The size of benchmark is "test". The results, such as execution cycles, number of cache misses, number of cache accesses and so on, are recorded with hardware logics and sent to a PC with UART.

The parallel executions of three SPEC applications are also used to measure the system performance. The following commands execute gzip, parser and twolf in parallel with the task function of the OS.

```
$ ./start_count
$ ./run_gzip &
$ ./run_parser &
$ ./run_twolf &
$ wait
$ ./end_count
```

The commands "start_count" and "end_count" are the start and end of the measurement, respectively.

Figure 5.1 shows the execution time and speedup against baseline. In this figure, baseline means a configuration with 256 cache lines, and AMI means an AMI-based configuration with 255 cache lines. In this evaluation, AMI hardly affects the execution time.

Figure 5.2 shows the miss rate and the miss rate reduction. Miss rates increase with AMI in some applications, but this effect is also small.

Next, Table 5.1 shows the hardware resource. AMI raises occupied logic cells by 1.206 times in the scale of dcache, and by 1.011 times in the scale of the overall system. The

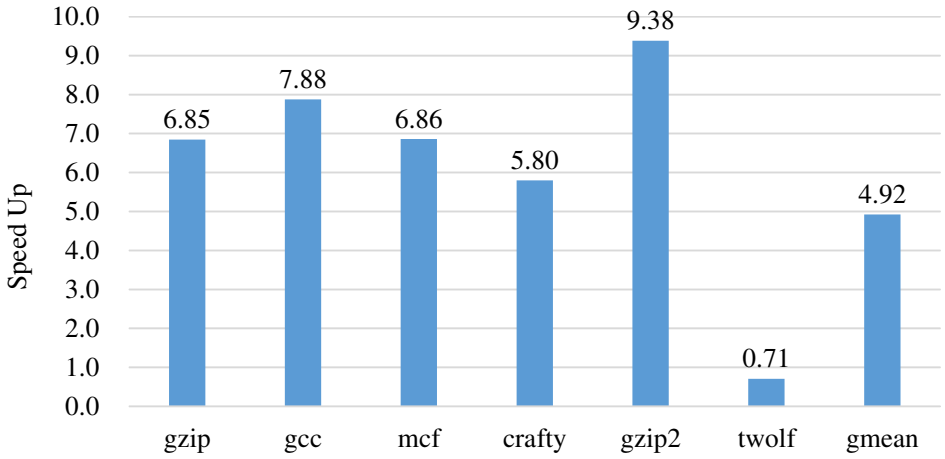


Figure 5.3: The speedup of execution time on Frix compared with a software simulator gem5

occupied embedded memory to implement dcache increases by only 1.007 times. Thus, these results show that the hardware overhead of AMI is very small.

Cyclone IV EP4CE115 FPGA has 114,480 logic cells and 3.888 Kbits embedded memory. In the result of the baseline, the usage of logic cells is 54559/114480 = 47.7%. Researchers can use more than the half of logic cells to implement their proposed architecture. The usage of embedded memory is 2585.856/3888 = 66.5%, and thus researchers also use about the 1/3rd of total embedded memory.

5.1.3 Discussion

Like the cache, developers can modify the hardware configuration and estimate the hardware resource usage. Since it can run a Linux OS, developers are able to evaluate the performance of their module with a complex benchmark such as the SPEC CPU2000 benchmark. Moreover, no specific cross-compiler is required to compile applications because Frix employs an x86 soft processor and many general purpose computers have an x86 processor. Hence, Frix is stable environment for developers to evaluate their architectural ideas.

Compared to software-based simulators [13] [14], FPGA-based simulators can complete a simulation much more quickly. in terms of simulation time, Frix is compared with gem5 [13], which is a major software-based full-system simulator. Six applications from the SPEC

CPU2000 INT benchmark suite, which can be run in both environments, are selected to compare total simulation time of the applications. The 4-way 4-word set-associative L1 data cache is embedded in both environments. Although I try to set the architectural parameters the same values in both environments, a few configurations are different. For instance, gem5 simulates a 64bit x86 processor while the processor of Frix is 32bit x86 ISA. Thus, this comparison includes a mite error. Gem5 is compiled with gcc 4.8.2, and the compile option is "-O2" which is the default compile option of gem5. The execution environment of gem5 is Ubuntu 14.04 with Core i7-3770K operating at 3.5GHz and 16GB DRAM.

Figure 5.3 shows the speedup ratio of Frix in comparison with gem5. The result shows that Frix achieves up to 9.38 times higher simulation speed than gem5, and the average simulation speedup ratio is 4.92 times. In only twolf, gem5 simulation speed is slightly faster than Frix. This is because that the execution time of twolf is short and it requires many file read/write operations. Since the SoC does not have DMA function yet, data except the BIOS program stored in the SD card are transferred to the DRAM via ao486 processor. This process is probably a bottleneck. However, all of the other applications achieve significant speedup ratio. For these results, it is obvious that Frix is able to more quickly simulate target hardware than software simulators. As the simulated hardware becomes larger and more complex, the simulation speed gap between the software and Frix expands.

Frix is also a suitable environment for computer education because the RTL source code of Frix is released as open-source. Leaners can understand how a computer system works by reading the source code. Unlike ao486 SoC [11], the proposed SoC can run on the two major vendors' FPGAs because it does not use vendor-dependent IP cores such as Avalon Interconnect and Nios Ⅱ in main functions. In general, many HDL processors are released as open-source, but the source code of peripherals modules such as buses or I/O controller is seldom provided. Frix is the first open-source SoC which includes an x86 soft processor and some peripheral modules, and can run on two major vendors' FPGAs.

5.2 Efficiency of AMI-Based Cache System

In this section, I evaluate the performance of the proposed AMI-based cache system on Frix.

5.2.1 Setup

The proposed cache is implemented on Frix with modifying an original data cache in ao486 processor. The configuration of the original cache is mentioned in Section 5.1.1. The numbers of cache lines can be changed to 255 or 254 with multiplexers.

I evaluate the proposed cache system with 15 configurations which are (Way0, Way1, Way2, Way3) = (0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 0, 2), (0, 0, 1, 1), (0, 0, 1, 2), (0, 0, 2, 2), (0, 1, 1, 1), (0, 1, 1, 2), (0, 1, 2, 2), (0, 2, 2, 2), (1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 2, 2), (1, 2, 2, 2) and (2, 2, 2, 2). Each value means the number of unused cache lines on the way, and thus (0, 0, 1, 2) means that the numbers of cache lines are 256 in Way0 and Way1, 255 in Way2 and 254 in Way3.

5.2.2 Evaluation

In this section, I evaluate the proposed cache system in terms of miss rate, speedup, hardware resource and maximum frequency. These evaluations are done in the same environment mentioned in the Section 5.1.2.

Miss Rate

Figure 5.4 shows the overall miss rates. In this figure, miss rates in each configuration are written in bars. This figure also shows the best-case, written in a red line. The best-case means the least miss rate in the 15 configurations. The gmean is the geometric mean of miss rates in each application. Miss rates become high in order of mcf, parser, gzip, gcc, gap, crafty, bzip2, vortex, perlbmk and twolf. Miss rates are about 20% in mcf, and less than 10% in the other applications.

Figure 5.5 shows the overall miss rate reduction. These results are calculated as the division of each miss rate by that in the conventional cache “0000”, and lower values are better. In the results, the configurations are roughly classified into three categories according to the miss rate: baseline (“0000”), non-skewed (“1111”, “2222”) and skewed (the others). In the most part of these applications, skewed configurations achieve lower miss rates than the other configurations. Though non-skewed configurations do not affect the geometric mean of miss rates, skewed configurations decrease it significantly.

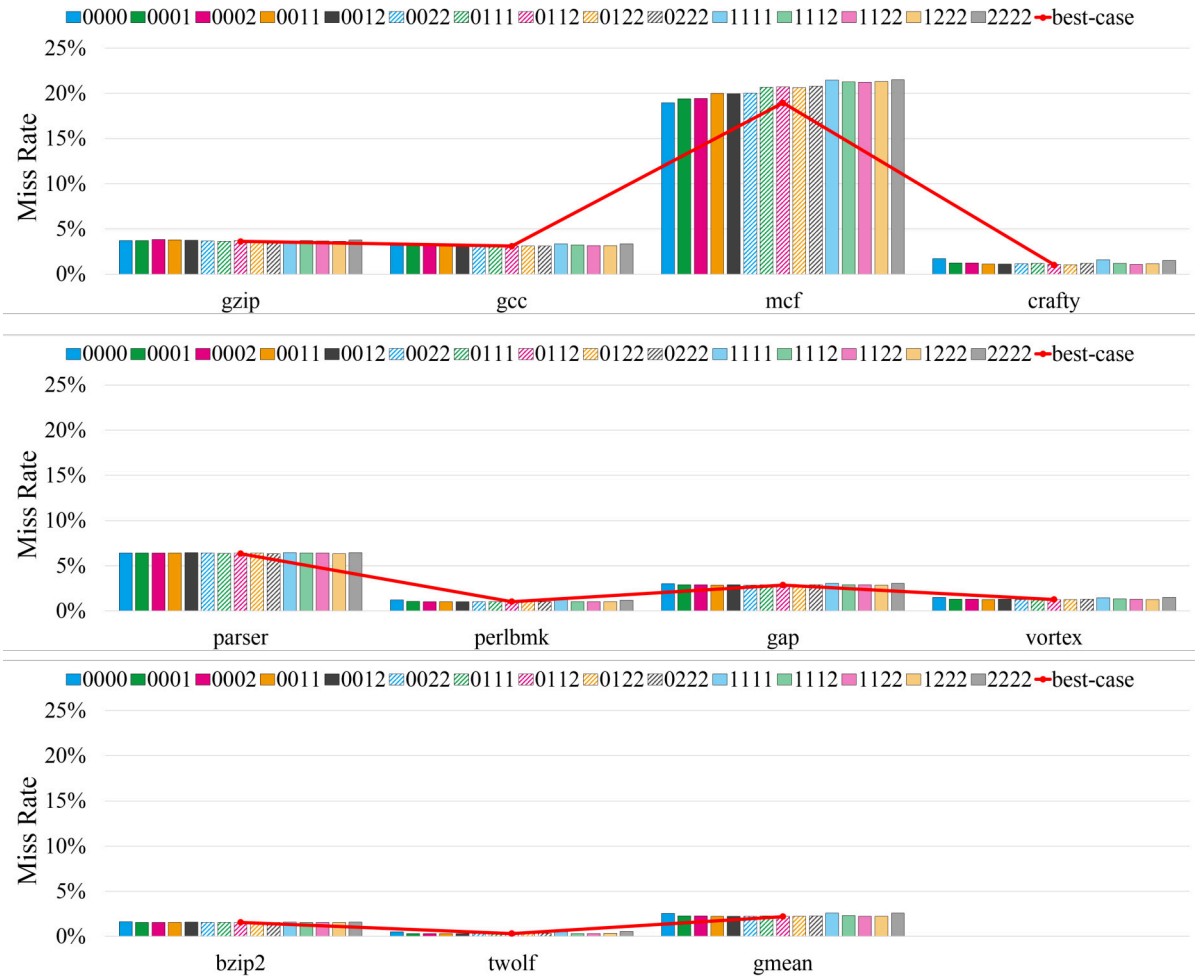


Figure 5.4: Overall miss rates

The result also shows, in crafty, non-power-of-2 indexing functions work very effective. In particular, skewed configurations reduce about 40% of miss rates from baseline. Crafty is an application of playing a chess game. Since the size of the chess board is 8x8, data is managed with power-of-2 stride. In consequence, when cache capacity is power-of-2 value, these datas cause many conflicts. Thereby AMI-based indexing functions reduce conflicts.

In twolf, skewed configurations achieve 40% of miss rates reduction as well. In contrast, miss rates in non-skewed configurations are rather worse than that of the baseline. Thus, the cause of the reduction is considered to be the skewed indexing functions rather than the AMI-based indexing functions.

In mcf, AMI-based indexing functions raise miss rate. In this result, the miss rates increase

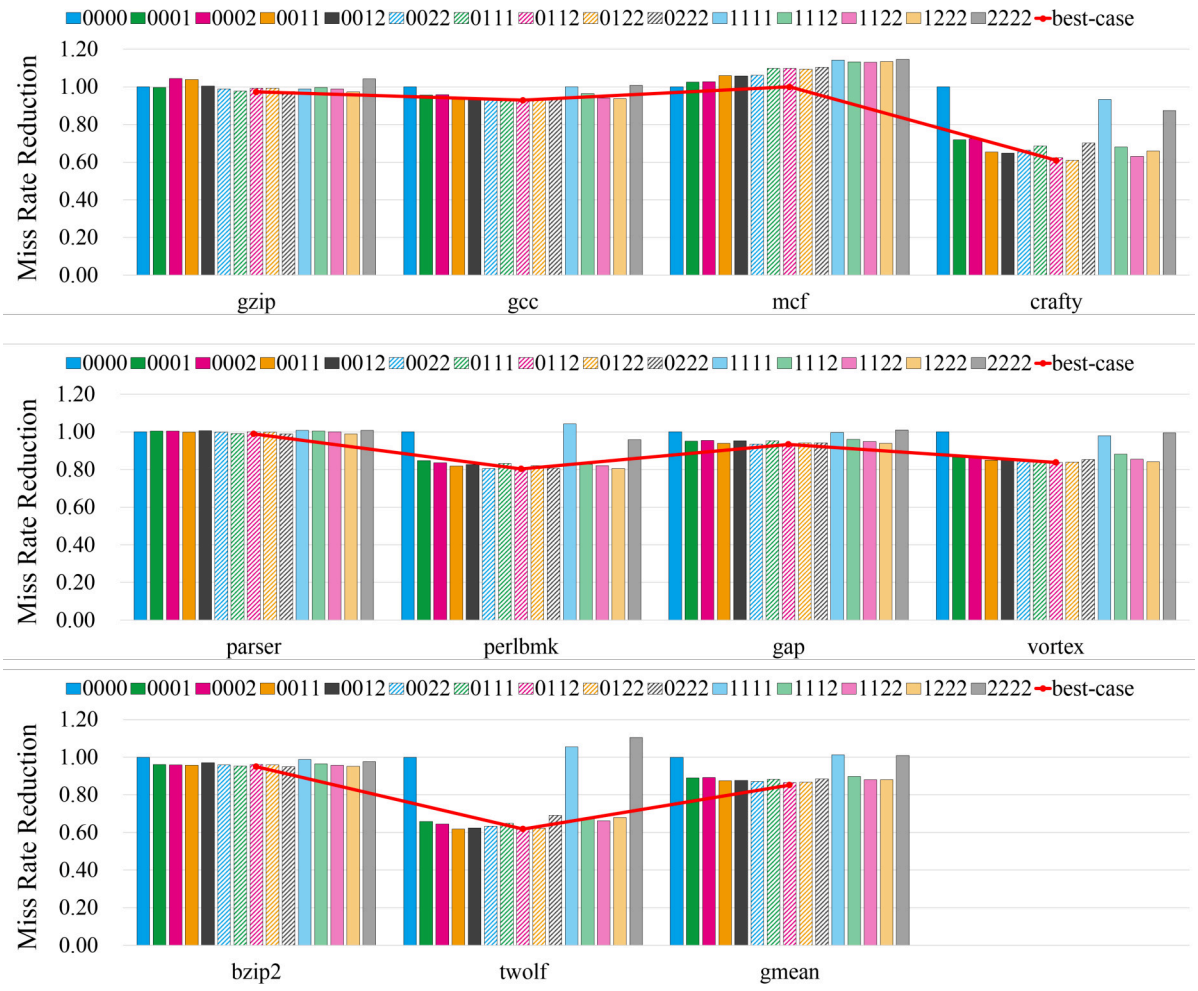


Figure 5.5: Overall miss rate reduction

with the number of ways with non-power-of-2 cache lines. Thus, the miss rate of “0011” is between the one of “0001” and “0111”. Though the cause of these results is not examined in detail, in some applications, the effect of AMI may become rather worse. Even in mcf, the miss rate in the skewed configurations is less than the one of the non-skewed configurations when the numbers/ of AMI-based ways are the same.

The configurations which achieve the highest miss rate reduction are shown in Figure 5.6. For instance, the miss rate in gzip gets the least value in the “0222” configuration. Though the baseline “0000” achieves the least miss rate in mcf, the skewed configurations achieve the least miss rates in the other applications. About 40% reduction is shown in crafty and twolf, and about 15% reduction is shown in perlbmk and vortex. In the other applications, the

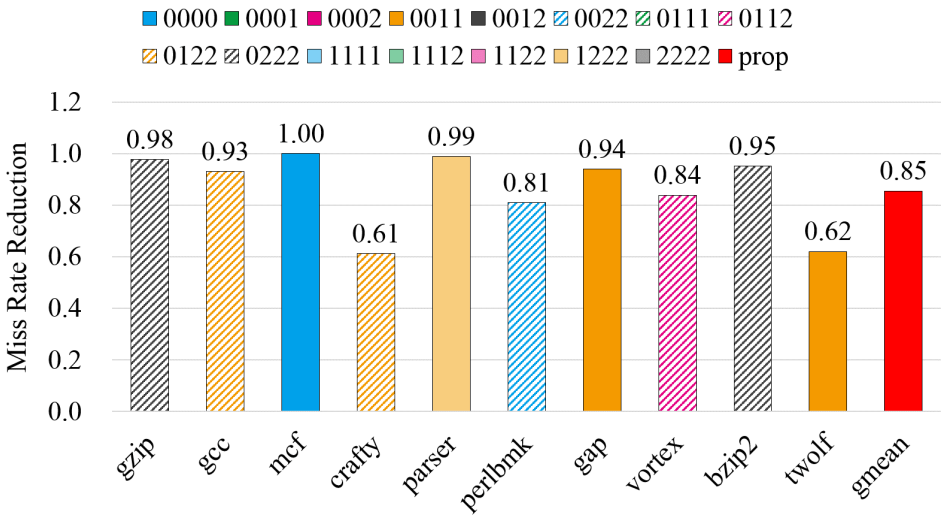


Figure 5.6: Configurations achieving the highest miss rate reduction

reductions are small. The "prop" is the geometric mean of the highest miss reductions, and the proposed cache achieves 0.85 times reduction of miss rate on average.

Speedup

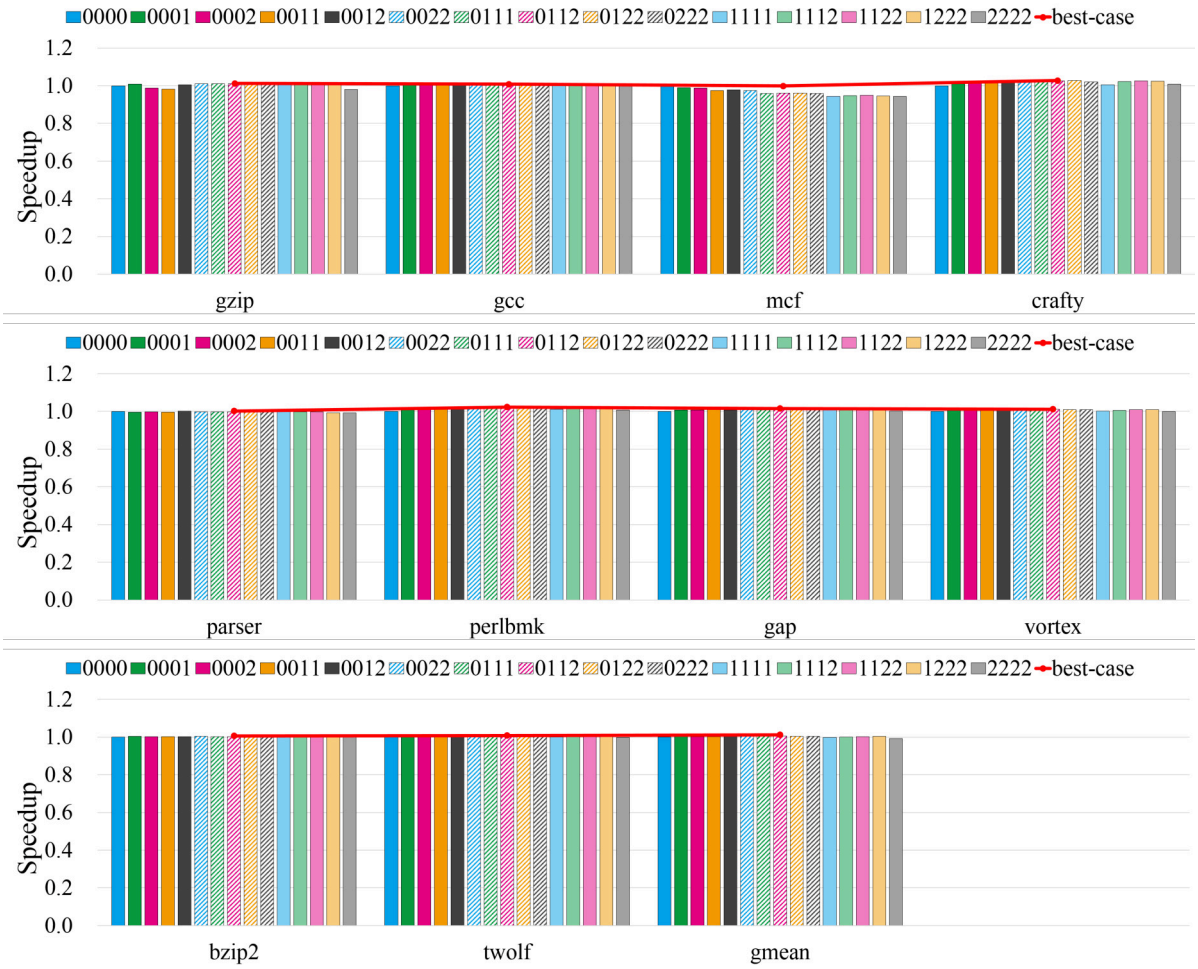
Figure 5.7 shows the overall speedup. This speedup is calculated as the division of execution time in the baseline "0000" by the one in each configuration, and higher values are better. In this evaluation, AMI hardly affects the system performance. This may be because DRAM access latency on this system, which is 10 cycles, is much smaller than that on high performance processors. In Section 5.2.3, I will evaluate the reduction of execution cycles with increasing DRAM access latency.

Figure 5.8 shows the configurations achieving the highest speedup. In mcf, the conventional cache "0000" achieves the highest speedup. In contrast, the caches with skewed indexing functions achieve the highest speedup in the other applications. However, the difference of speedup is very small.

Hardware Resource

Figure 5.9 shows the amounts of occupied logic cells. The baseline includes a conventional cache with 256 cache lines, and the proposal includes my proposed cache with reconfigurable

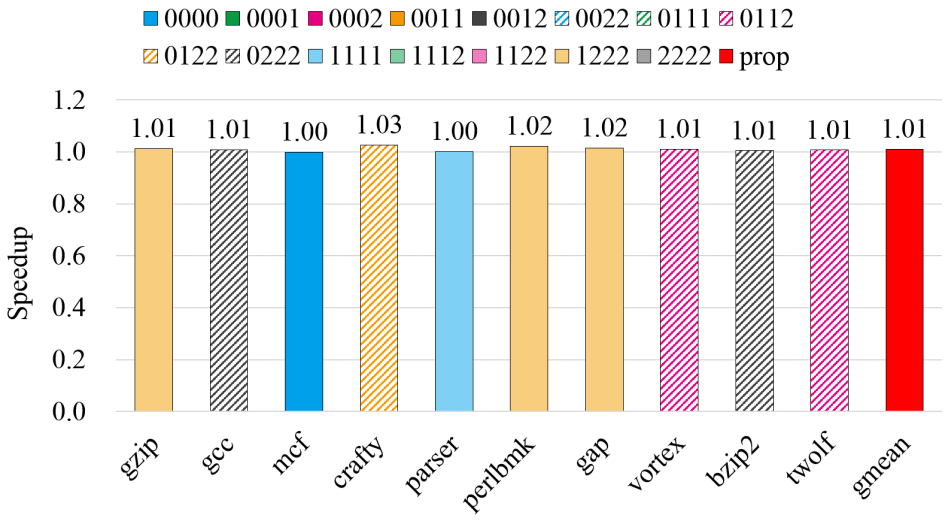
Figure 5.7: Overall speedup



cache lines. In the scale of dcache, the occupied logic cells increase by 1.44 times from baseline. Since the total logic cells in Cyclone IV EP4CE115 FPGA are 114,480, the additional 2000 logic cells are only 1.7% of total resources on the FPGA. Hence, the overhead of logic cells is very small.

Figure 5.10 shows the amounts of occupied memory bits. The memory bits increase by 1024 from baseline because of the additional tag field. In this evaluation, each way in the data cache has 256 cache lines, and so the number of cache lines in the whole cache is 1024. Since extra 1-bit is added to the tag memory in each line, the total additional bits are 1024-bits. These 1024-bits are much smaller than 154,624-bits on the whole cache. Each line has 151-bits, which are composed of 128-bits data, 20-bits tag, valid bit, dirty bit, and NRU bit.

Figure 5.8: Configurations achieving the highest speedup



The memory bits increase by only less than 1.01 times, and hence the overhead of memory bits is still very small.

Maximum Frequency

The maximum frequency is shown in Figure 5.11. The frequency is evaluated with two models: "Slow 1200mV 0C Model" and "Slow 1200mV 85C Model". The difference between these two models is temperature of the FPGA. In the both of two models, the system frequency decreases by 0.95 times. This is because the AMI-based indexing functions increase the latency.

Evaluation Summary

In this section, I evaluated my proposed AMI-based cache system on Frix. According to the results of miss rates, it is found that the proposed cache system achieves high reduction of miss rates in crafty and twolf, and the system reduces miss rate by 15% on average. In this evaluation, 15 configurations are compared, and according to the miss rates, they are classified into baseline, non-skewed and skewed. Non-skewed configurations improve hit rates in the crafty, which manages data with power-of-2 stride and then occurs many conflicts. Moreover, skewed configurations reduce more conflicts in the most part of the benchmark suite.

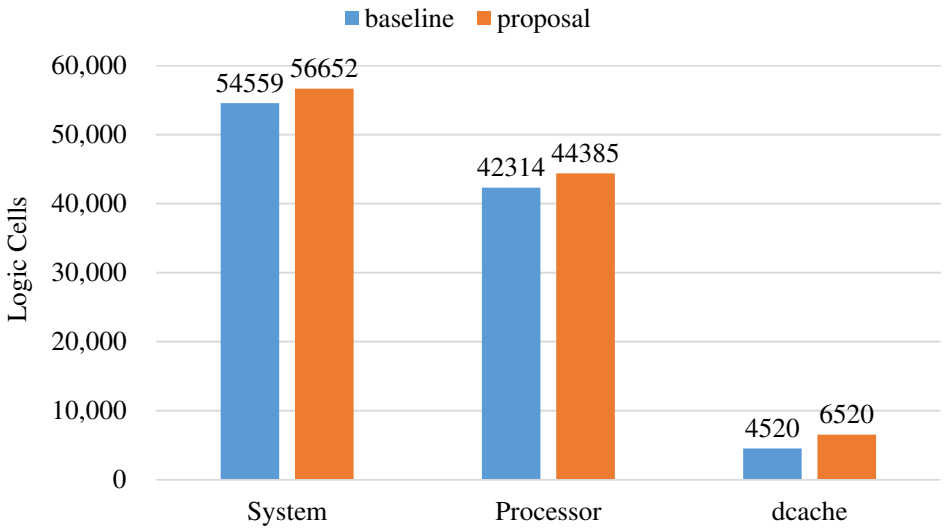


Figure 5.9: Logic cells

In the speedup evaluation, there is almost no difference from the baseline. The cause is considered to be the low DRAM access latency in the Frix. In the next section, I discuss the speedup of the proposed system on the environment with higher DRAM access latency.

The overhead of hardware resource is very small on the proposed system. The occupied memory bits increase by only 1-bit per a cache line. Since a cache line manages so many bits, which are composed of data bits, tag bits, valid bit and so on, the additional 1-bit is negligible. The occupied logic cells increase by 1.44 times from baseline in the scale of dcache.

The maximum frequency decreases by 0.95 times. This is because the AMI-based indexing functions increase the latency.

5.2.3 Discussion

Methods to Improve the System Frequency

In the evaluation, the proposed system reduces the maximum frequency by 0.95 times. Since the AMI-based indexing functions, as described in Section 3.1, is implemented as a cascade of adders, the cache latency significantly increases.

In the modern high performance processors, caches are pipelined in general. When a cache is pipelined, the waiting cycle increases, but the cache can run at a higher frequency. Thus a

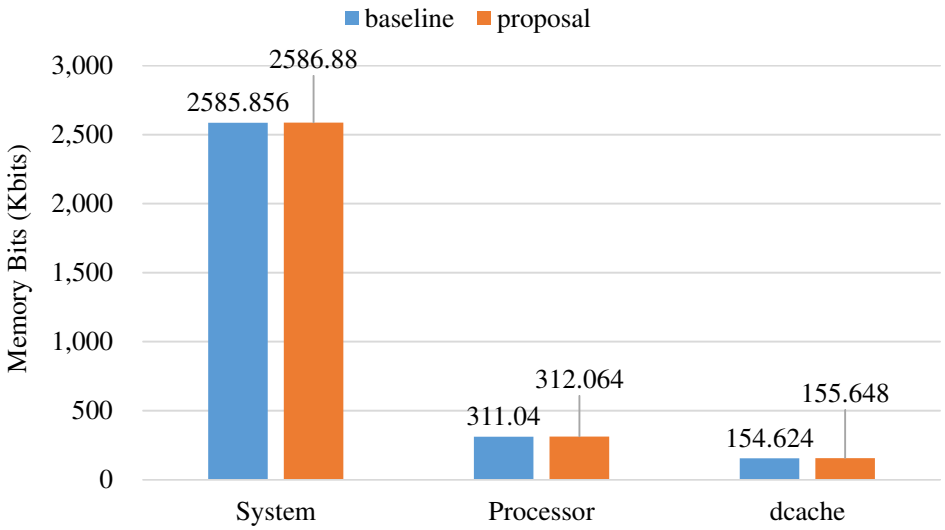


Figure 5.10: Memory bits

pipelined cache with AMI-based indexing functions is a better implementation. In addition, the Figure 3.2 shows that the AMI-based indexing function can be easily pipelined, and the proposed system can run on the environments where the very high frequency is desired.

Speedup Evaluation with Increasing DRAM access Latency

In the evaluation, there is almost no difference in speedup though the miss rate decreases. It is assumed that the DRAM latency on this system, which is 10 cycles, is much smaller than that on high performance processors. In this section, I discuss that if the DRAM latency increases, how high the speedup can achieve.

Since ao486 processor is an in-order scalar processor, when assuming the miss rate of L1 instruction cache is enough small, the execution cycle is roughly calculated as follows:

```
Execution Cycle = Pipeline Cycle
                + (Number of L1D Hit) × L1D Latency
                + (Number of L1D Miss) × DRAM Access Latency
```

I evaluate the execution cycle with increasing DRAM access latency.

In this discussion, the DRAM latency gets value from 10 to 200. Figure 5.12 shows the speedup in terms of execution cycles (not execution time). In crafty, the speedup gets the

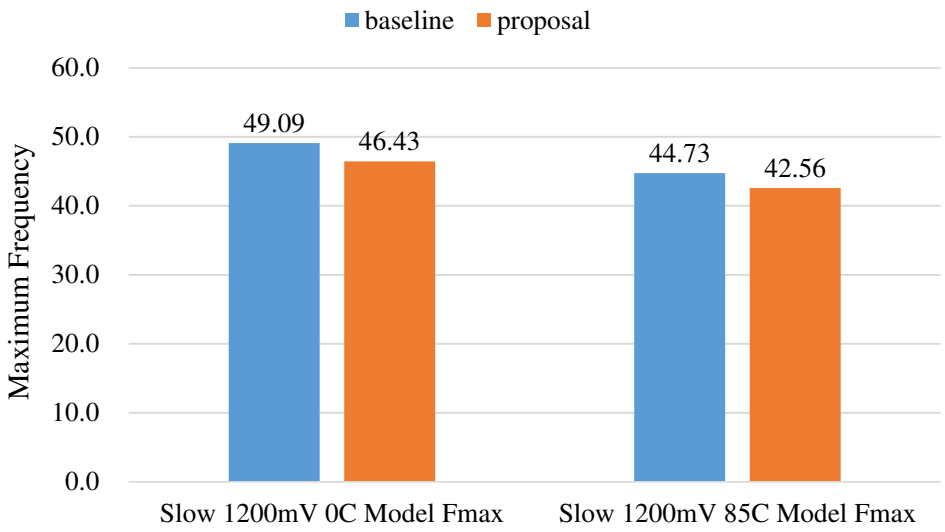


Figure 5.11: Maximum frequency

highest value in these applications. The speedup is about 1.09 times when the DRAM latency is 100 cycles, and about 1.14 times when 200 cycles. This is because the AMI-based indexing functions reduce miss rate in crafty higher than in the other applications. However, the speedup is not high in twolf, which achieves the same miss rate reduction as crafty, and the cause is considered to be the miss rate of twolf is very low. In mcf, the speedup is not changed because the baseline configuration is the best configuration. In geometric mean, 1.027 times speedup is achieved in the 100 cycles latency, and 1.040 times speedup is achieved in the 200 cycles latency.

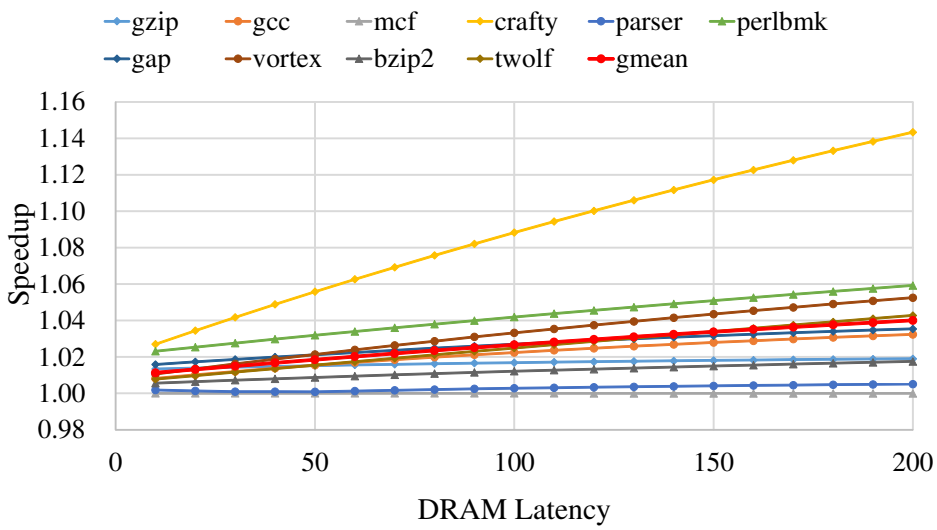


Figure 5.12: Configurations achieving the highest speedup with increasing DRAM access latency

Chapter 6

Conclusion

In this paper, I proposed a cache system which selects the suitable number of cache lines depending on an application. The proposed system uses Arbitrary Modulus Indexing to implement the non-power-of-2 cache lines, and thereby reduce cache conflicts.

I also proposed an IBM PC Compatible SoC which can run general purpose OSs, named Frix. Although Frix is based on ao486 SoC, Frix does not use vendor-dependent IP cores in the main function. As a result, Frix can run on two major vendors' FPGA boards.

I evaluated the usability of Frix and the efficiency of the AMI-based cache system. In order to evaluate these factors, the proposed cache system is implemented on Frix, and the SPEC CPU2000 INT benchmark suite is executed on the system.

Frix can execute the SPEC CPU2000 INT benchmark suite accurately. The evaluation result shows that Frix takes one-fifth simulation time of that of the software simulator on average. It is shown that Frix is a useful environment for computer research.

The AMI-based cache system reduces miss rate in many applications. The skewed indexing functions can reduce more conflicts than the conventional function. The system achieves 0.85 times miss rate reduction on average with low hardware resources. Moreover, supposing the DRAM latency is enough high, the system also achieves up to 1.14 times speedup.

Acknowledgement

本研究を進めるにあたり，終始熱心なご指導を賜りました吉瀬謙二准教授に深く感謝致します．

また，協力して研究に取り組む場を与えてくれた吉瀬研究室の皆様に感謝致します．小林諒平先輩には，研究の進め方から論文の書き方まで，様々なご助言を頂きました．Frix の開発を共同で行った小川愛理さんと味曽野智礼君には，特に感謝致します．二人と共同で研究を行わなければ，このシステムは完成しなかったと感じます．

最後に，綾鷹を毎日販売して頂いた，吉瀬研究室の歴代店長に感謝致します．

First of all, I would like to thank my supervisor, Associate Professor Kenji Kise. His constant support, guidance, and encouragement have been essential for me to complete this Master Thesis.

In addition, I appreciate for all the members at Kise Laboratory. I appreciate Ryohei Kobayashi for giving me many helpful advices for research. I also appreciate Eri Ogawa and Tomohiro Misono for developing Frix with me.

Finally, I really thank the managers on Kise Laboratory for giving me AYATAKA everyday.

Bibliography

[1] J. Diamond, D. Fussell, and S. Keckler, "Arbitrary Modulus Indexing," in *Proceedings of 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2014, pp. 140–152.

[2] M. Kharbutli, D. Solihin, and J. Lee, "Eliminating conflict misses using prime number-based cache indexing," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 573–586, May 2005.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[4] C.-L. Su, C.-C. Teng, and A. Despain, "A study of cache hashing functions for symbolic applications in micro-parallel processors," in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 1994, pp. 530–535.

[5] A. Seznec, "A case For Two-way Skewed-associative Caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, May 1993, pp. 169–178.

[6] F. Bodin and A. Seznec, "Skewed Associativity Improves Program Performance and Enhances Predictability," *IEEE Transactions on Computers*, vol. 46, no. 5, pp. 530–544, May 1997.

[7] T. Givargis, "Improved Indexing for Cache Miss Reduction in Embedded Systems," in *Proceedings of Design Automation Conference (DAC)*, June 2003, pp. 875–880.

[8] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proceedings of 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2010, pp. 187–198.

[9] Sun Microsystems, "UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007," Draft D1.4.3. 2007.

[10] "Inside the Intel Itanium 2 Processor," a Hewlett Packard Technical White Paper, July 2002.

[11] "ao486," https://github.com/alfikpl/ao486.

[12] Altera, "Qsys," http://www.altera.com/products/design-software/fpga-design/quartus-ii/quartus-ii-subscription-edition/qts-qsys.html.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[14] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Design Automation Conference 2011 (DAC'11)*, 2011.

Publication

Conference Papers

1. **Yuki Matsuda**, Ryosuke Sasakawa, and Kenji Kise, "A Challenge for an Efficient AMI-Based Cache System on FPGA Soft Processors", *In Proceedings of Third International Symposium on Computing and Networking (CANDAR)*, December 2015.

2. Eri Ogawa, **Yuki Matsuda**, Tomohiro Misono, Ryohei Kobayashi, and Kenji Kise, "Reconfigurable IBM PC Compatible SoC for Computer Architecture Education and Research", *In Proceedings of 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 65–72*, September 2015.

Others

1. **Yuki Matsuda**, Eri Ogawa, Tomohiro Misono, Ryohei Kobayashi and Kenji Kise, "Frix: Feasible and Reconfigurable IBM PC Compatible SoC", *the 78th National Convention of Information Processing Society of Japan*, March 2016.

2. Susumu Mashimo, **Yuki Matsuda**, and Kenji Kise, "Fast Merge Network for Sorting on FPGA", *the 78th National Convention of Information Processing Society of Japan*, March 2016.

3. Takuma Usui, Susumu Mashimo, **Yuki Matsuda**, Ryohei Kobayashi and Kenji Kise, "A Study of the World's Fastest FPGA-based Sorting Accelerator", *the 78th National Convention of Information Processing Society of Japan*, March 2016.

4. Paniti Achararit, **Yuki Matsuda** and Kenji Kise, "An Approach to Real Time Tennis Ball Speed Analysys on Tablet PC", *the 78th National Convention of Information Processing Society of Japan*, March 2016.

5. **Yuki Matsuda** and Kenji Kise, "A Scalable Computer System with 3 Types of FPGA Boards", *the 77th National Convention of Information Processing Society of Japan, Vol. 1, pp. 171–173*, March 2015.

6. **Yuki Matsuda**, Eri Ogawa, Tomohiro Misono, Naoki Fujieda, Shuichi Ichikawa and Kenji Kise, "MieruSys Project : Developing an Advanced Computer System with Multiple FPGAs", *IEICE Technical Reports RECONF2014-79, pp. 211–216*, January 2015.