計算機アーキテクチャ 150−16
（２００２．１１．２８）

Implementation of FIFO Buffer Using Cache Memory

Khairuddin bin Khalid† and Kiyofumi Tanaka†,††

Conventional caches are not always made use of in some familiar applications because of no locality in memory accesses. To make use of a large cache memory space in a today's processor, we propose a built-in FIFO buffer based on a concept of reconfigurable caches where cache memory space is divided into several partitions dynamically. The FIFO mechanism can be implemented with small additional hardware, avoid memory fragmentation, and improve performance of data accesses.

1. Introduction

Current high performance general-purpose processors are used for a variety of application domains, including scientific, engineering, transaction processing, and decision support. Several quantities characterizations have shown that applications from different domains exhibit different characteristics[1]. As computer systems are used increasingly in wide variety of applications, a "one-size-fits-all" design philosophy will be inadequate. For example, the use of large caches is a common trend across general-purpose systems, sometimes consuming up to 80% of the total transistor budget and up to 50% of the die area[2]. While large caches are effective for a variety of conventional workloads, they are often ineffective for media processing applications because of the streaming nature of the data accesses and the large working sets in these applications.

In response to this observation Ranganathan et. al[3] proposed a new caches organization called reconfigurable caches. Reconfigurable caches allow the on-chip SRAM to be dynamically divided into different partitions that can be assigned to different activities.

When a processor sequentially accesses many data which are located at a certain interval in a memory, those data are brought into the caches and put at a certain interval, which means that many other data that are around the accessed data and would not be touched are also inserted into the cache. This kind of layout wastes memory space and processing time in conventional caches.

In order to prevent the above situation from occurring, it is effective for a processor to have a built-in FIFO buffer and use it instead of the cache when many non-contiguous data are accessed. In this study, we are focusing on application of reconfigurable caches to a FIFO buffer. By implementing one partition in reconfigurable caches as a FIFO buffer, we can avoid fragmentation in caches and improve performance of memory accesses.

2. Conventional Caches

Cache memory is a part of memory hierarchy, that is, an intermediate memory between a processor core and the main memory. To take advantage of a feature of spatial locality which various programs have, memory words are grouped into small blocks or lines. When a caches miss occurs, the processor will then fetch a block including the missing data or instruction from the next level cache or external memory. The block consists of multiple words that are adjacent and carry a high probability of being needed shortly. The cache memory holds such blocks and the contents of it are thus copies of a set of main memory blocks.

2.1 Data Placement Scheme

On block placement in a cache, the simplest scheme is to place a memory block in exactly one location determined uniquely by the memory address of the block. This kind of scheme is called direct mapped. The problem with the scheme is that it lacks flexibility, that is two or more blocks easily conflict with each other for the location.

On the other hand, in fully associative

† School of Information Science, Japan Advanced Institute of Science and Technology
†† "Information and Systems", PRESTO, Japan Science and Technology Corporation(JST)

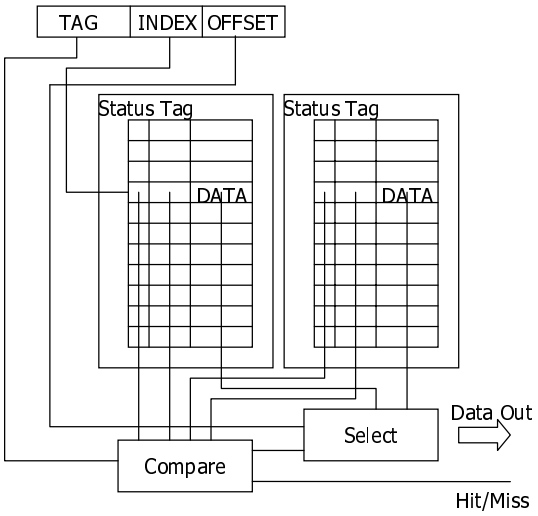

Fig. 1 2-way set associative cache.

scheme, a block can be associated with any entry in the cache, giving more flexibility. However, to find a given block in fully associative cache, all the entries in the cache must be searched because a block can be placed in any entry regardless of the address. Another problem with fully associative cache is that much more hardware is needed, significantly increases the access time and hardware costs.

The middle range of designs between direct mapped and fully associative is set associative. **Fig. 1** depicts the structure of a single-ported 2-way set associative cache. In a N-way set associative cache, there are N data and tag arrays and each of N arrays is referred to as a way. A group consisting of entries with an index is referred as a set. Therefore, a set contains N entries in N-way set associative. A block can be placed in any entry of the corresponding set. The set associative caches represent moderate access time and hardware costs[4].

2.2 Addressing

On the addressing method, the lowest portion of an address, called offset in a block, is used to select an accessed data (byte, word, etc.) in a memory block. The middle portion of an address, index field, is used to select a caches entry consisting of a data and tag. A tag field is used to distinguish between blocks with a different address but the same index field and it is the upper portion of an address.

The tag field of an input address is sent to the comparator(s) in the cache mechanism to determine if there is a match between the tag value and any of the tags read from the tag array. If there is a match, a hit signal is sent to the output and the data is valid to be read. If there is no hit, a miss signal is being sent until the missing block is read out from the next hierarchical cache memory or external memory and filled into the corresponding cache entry.

It is a common trend today that processors have large caches. While some applications use large caches effectively, some don't because of no temporal or spatial locality[5]. To fully utilize large caches, Ranganathan et. al[3] proposed reconfigurable caches described in the next section.

3. Reconfigurable Caches

Paper[3] showed that by enabling caches to be divided into multiple partitions dynamically and use those partitions as lookup tables for instruction reuse, there are 1.04X to 1.20X of improvements in media processing benchmarks. They called this kind of design technique reconfigurable caches. There are several possible applications for reconfigurable caches such as lookup tables for value prediction, memorization, instruction reuse, compiler or compilation controlled memory and prefetched data partition. There is several design aspects that must be considered on designing reconfigurable caches and our option.

3.1 Partitioning

The key challenge in designing a reconfigurable cache is to devise mechanism for dividing the cache memory into different partitions. Our primary design exploits set-associativity in conventional cache organizations. The reconfigurable cache builds on the set-associativity in natural way, as depicted conceptually in **Fig. 2**. That is, we divide the reconfigurable cache into partitions at the granulity of the ways of the set-associative cache as in Fig. 2. The example

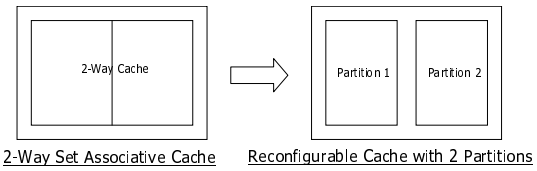

Fig. 2 Conceptual partitioning in 2-way set associative.

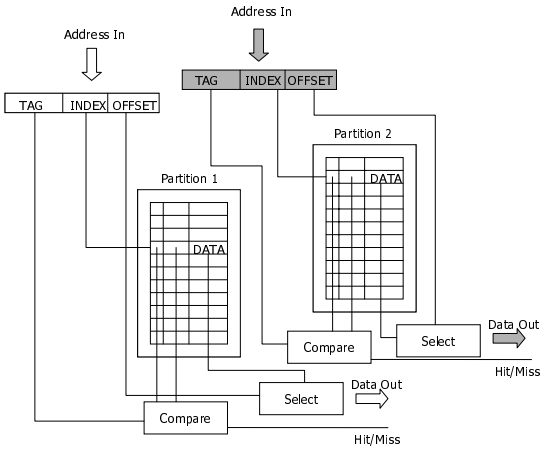

Fig. 3 Reconfigurable cache based on 2-way set associative.

of exploiting the set-associativity for a reconfigurable cache is shown in **Fig. 3**

In our implementation we are going to modify conventional 2-way set associative cache and make it accommodate general feature of a cache in one way and an effect of a FIFO buffer in another way.

3.2 Detection

A reconfigurable cache needs to know when it should act as a normal cache with 2-way associative, and when it should act as a reconfigurable cache with one partition for a normal cache and another partition for a special use. To make this change possible, we are going to provide a special register whose value invokes the reconfiguration. This special register is set by execution of a special instructions. That is, applications themselves control the configuration of the cache according to their decision. Resetting the register restores the cache configuration to the normal cache mode.

3.3 Addressing

In a conventional cache, the middle portion of the memory address (index field) is used to select a set that should be searched. The specified set by the index consists of data, tag, and status (validity, etc.) for each way. Each tag is compared with the upper portion of the address (tag field) to determine whether the entry corresponds to the memory address.

Addressing in a reconfigurable cache differs from that of a conventional one. The method depends on what purpose the special partitions are used for. For example, when a data cache is divided several partitions and one of the partitions is used as a branch or value prediction buffer, the partition would be indexed by a part of an instruction address instead of a data address.

Such use of a partition forces the cache organization to have an extra input port of addresses and also an extra output port since two or more partitions might be accessed simultaneously. We propose an application of a reconfigurable cache, a FIFO buffer, that does not require any extra ports. The structure is described in detail in the next section.

3.4 Data Consistency

In one of conventional caches, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the next level cache or main memory when it is replaced. This kind of scheme is called write-back.

In a reconfigurable cache, after a partition moves to a special purpose, it is necessary to write back the modified blocks belonging to the partition in the main memory before the entries of the blocks are overwritten by new data. There are two options, cache scrubbing and lazy transitioning[3]. The former performs all write-back operations on the partition when the reconfiguration occurs. The latter performs a write-back operation about a block only when the corresponding entry is being overwritten. We use the latter scheme since there is little difference between the scheme and that of conventional caches, which means it does not require much extra hardware.

4. An Application: FIFO Buffer

In general, instruction accesses by a program execution show a tendency of locality enough to have the benefit of an instruction cache. On the other hand, a data cache of the general cache structure has inefficiency against processing that exhibits no locality in data accesses. Therefore, we give the data cache the ability to switch into multiple partitions.

4.1 Motivation and Basic Policy

We are now focusing on an example, that is accesses to data of a certain interval in memory sequentially, where each data is accessed only once. Although they are sequential they are not continuous resulting in fragmentation in

conventional cache memory. The fragmentation is wasting not only space in cache memory but also time to transport data between the cache and external memory, since the cache mechanism assumes a block which includes data that being not accessed. This kind of data accesses can be found often in, for example, database processing. The use of a smaller size of block is one solution to the problem. However it requires much more tag and state memory spaces than that of the usual size of block and cannot receive benefit from space locality.

As for such accesses to sequential data of a certain interval, a memory address of each data is not important for the intention of a program. Only the fact of regular injections of sequential data into the execution of memory access instructions is essential. In order to prevent the above inefficiency of the fragmentation in the cache from occurring, it is effective to use a FIFO buffer instead of a conventional data cache. A characteristic of first-in and first-out is suitable for the sequential accesses taken up now.

Here, the size of a FIFO buffer needs to be large enough to deal with a large data set. Equipping a processor with a large FIFO buffer increases the chip size and costs. Therefore, we implement a FIFO buffer within a partition in reconfigurable cache. This strategy is based on the observation that a partition, or way, in a cache is not small, 4 Kbytes or more in today's microprocessors, and that the cache is not used effectively anyway for the memory accesses in question.

4.2 Structure of Partitions

We build a reconfigurable cache as a FIFO buffer based on a 2-way set associative cache. In our reconfigurable cache, when the value of the special register for the cache configuration is zero, the cache functions as a 2-way set associative cache. When the cache is reconfigured, that is when the register is set, one partition is used as general-purpose data cache and we use the other partition as a FIFO buffer. **Fig. 4** shows the configuration after it is configured as multiple partitions.

The left partition in the figure functions as a direct mapped data cache and the right one as a FIFO buffer. Which partition is accessed from a memory access instruction depends on the physical memory address. For example, the highest bits in an address decide it as in the figure. This means that it is necessary for a compiler or linker to relocate data sets being accessed via the FIFO appropriately and for a virtual memory mechanism to translate the virtual addresses to the corresponding physical addresses. Since both partitions are accessed as data accesses, only a partition results in giving a data to a requesting instruction at a time. In other words, there is no chance for both partitions to be needed at the same time. Therefore, our reconfigurable cache has a single output port common between two partitions, whereas the original reconfigurable caches depicted in Fig. 3 must have the same number of output ports as partitions.

An entry to be read in the FIFO is indicated by a read counter. Similarly, an entry to be written is indicated by a write counter. These counters are automatically increased by one at read or write, respectively. When the value of the read counter is equal to that of the write counter, no valid values exist in the FIFO and a miss signal is asserted. Consequently, the FIFO buffer does not need to be indexed by any address other than the two counters. This means that an extra address input port that the original reconfigurable caches have is not required for our FIFO usage. In addition, the structure does not need any tag matching for searching, which can simplify the hardware of our reconfigurable cache.

4.3 Cooperation with Bus Transfer Mechanism

Memory access instructions (load instructions) take a data out of a FIFO buffer, while data read from external memory are to be inserted into the FIFO buffer. Although our FIFO buffer aims chiefly at avoiding fragmentation in conventional cache, the mechanism exhibits its power only when it absorbs a gap between the speed of execution of load instructions and that of accesses to external memory. One of options is the introduction of a prefetch instruction. Execution of the instruction can inject data to be requested into the FIFO in advance. However, the execution of the instruction passes through pipeline resources in the processor and therefore leads to overheads that cannot be neglected.

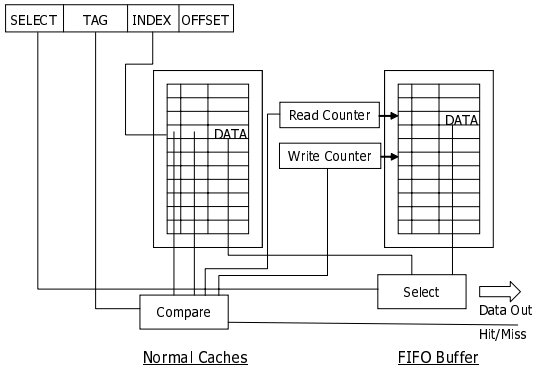

Fig. 4 Reconfigurable cache as FIFO buffer.

Some support mechanism should be provided to make full use of the FIFO buffer. In this study, we assume that there is one of DMA mechanisms outside the processor, called "stride data transfer (SDT)"[6], which can cooperate with the FIFO buffer in the processor. We describe the outline of SDT and relationship between the SDT and the FIFO buffer in this subsection.

One of typical applications which show no locality in data access is database processing. When a column with an attribute in a relation table is looked over, data located at the interval that corresponds to the size of a tuple are accessed sequentially. DRAM is constructed in rectangle array of memory cells. Data in the array is accessed by indicating row and column addresses. After the cells in a row addressed by the row address are latched, the target data is specified by the column address and output. A memory controller divides a memory address from a processor into the row and column address and send them to DRAM one by one. Although general memory controllers can only read and write continuous bits in a row in a burst mode, the DRAM structure can let the memory controller access any bits, for example, bits at a fixed interval, in a row quickly only by receiving corresponding column addresses. The memory controller has only to know the interval and the number of data to generate the column addresses.

The procedure of the stride data accesses is as follows.

(1) Before a program invokes the SDT mechanism, the execution of instructions sets the address space identifier, interval and the number of data to the registers in the memory controller.

(2) When the execution of a load instruction for the target data set meets a FIFO miss, the processor issues a memory request to the memory controller.

(3) When the memory controller finds the matching of the physical address concerned and the value of the address space identifier set in advance, it dispatches a row and column addresses to DRAM and sends a data read from DRAM to the processor. After that, it generates the next column address by adding the previous column address and the value of the interval, dispatches it to DRAM, decrements the value of the register indicating the number of data, and sends a data read to the processor. This process is repeated unless the column address exceeds the size of a row or the number of data gets zero.

(4) Every time the processor receives a data, the data is inserted into the FIFO and the write counter is incremented.

It is possible for FIFO misses to occur when a load instruction is executed before the corresponding data arrives at the FIFO. The processor recognizes this as a FIFO miss and issues a request to the memory controller. The memory controller discards this request when the address is within the ongoing SDT. The memory controller suspends the transfer when the calculated column address exceeds the row size. Then the processor restarts the step (2) and the following steps after all data in the FIFO are consumed

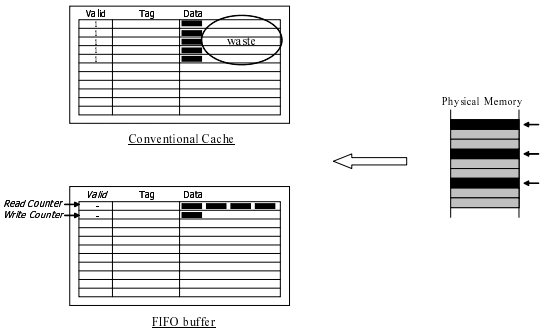

Fig. 5 Efficient use of cache memory space by FIFO against stride data.

By the SDT mechanisms, the FIFO based on cache memory is filled efficiently and speedily as in **Fig. 5**.

5. Conclusion

While the use of large caches is a common trend across general-purpose systems, it is often ineffective for some of applications, for example, media processing and database processing since they might not exhibit any locality in the data accesses.

Reconfigurable caches have already been proposed for the effective use of cache memory space when the conventional caches could not find any locality. In this paper, we proposed an application of reconfigurable caches where one partition is used as a built-in FIFO buffer and described the way of controlling it. The mechanism of the FIFO can be implemented with small additional hardware, prevent fragmentation in a cache memory from occurring, and improve performance of memory accesses. And furthermore, it becomes more effective when it cooperates with a DMA mechanism such as the SDT.

We have completed design of a basic CPU core using VHDL. The integer unit is based on the MIPS instruction set architecture[7] and each instruction is executed through a 5-stage (IF, ID, EX, MEM, WB) pipeline. This CPU core includes an instruction cache and data cache each of which is 2-way set associative. The data dache is now based on simple write-back. We will reorganize the data cache as a reconfigurable cache and evaluate the effectiveness in the next step.

Acknowledgments

In this study, we are using tools provided by university programs of Synopsys, Inc. and Mentor Graphics Corp. We would like to thank both of them.

References

1) K. Diendorf and P. K. Dubey, "How Multimedia Workloads will Change Processor Design." IEEE Computer, Vol.30, No.9, pp.43–45, 1997.
2) J.Henessy, "The Future of System Research." IEEE Computer, Vol.32, No.8, pp.27–33, 1999.
3) Parthasarathy Ranganathan, Sarita Adve and Norman P. Jouppi, "Reconfigurable Caches and Their Application to Media Processing." Proc. of ISCA, pp.214–224, 2000.
4) David A. Patterson and John L. Hennessy, "Computer Organization & Design: The Hardware/Software Interface." Morgan Kaufmann Pub., 1997.
5) Parthasary Ranganathan,Sarita Adve,Norman P. Jouppi, "Perfomance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions." Proc. of ISCA, pp.124–135, 1999.
6) T. Fukawa, K. Tanaka and J. Miyazaki, "The Highly Functional Memory Controller for Main Memory Database." Design Gaia 2002, ARC, 2002 (in Japanese).
7) Gerry Kane and Joe Heinrich, "MIPS RISC ARCHITECTURE." Prentice Hall, 1991.