Evaluation of Cache Memory as FIFO Buffer

KHAIRUDDIN BIN KHALID[†] and KIYOFUMI TANAKA[†,††]

Conventional caches are not always effective when a running program exhibits little locality in memory accesses. To utilize a large cache memory in today's processors, we propose a built-in FIFO buffer based on a concept of reconfigurable caches where a cache memory is dynamically divided into several partitions. The FIFO mechanism is 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 quantity 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 in the near future. 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 example, 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 caches organization called reconfigurable caches. Reconfigurable caches allow a on-chip cache memory to be dynamically divided into multiple partitions that can be assigned to different activities when a running application has little locality of data accesses.

When a processor sequentially accesses many data which are located at a certain interval in a memory, those data are brought into a cache 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 cache space. In order to prevent the 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 a cache and improve performance of memory accesses.

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

2. Reconfigurable Caches

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[4]. To fully utilize large caches, Ranganathan et. al[3] proposed 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.

2.1 Partitioning

The key challenge in designing a reconfigurable cache is to devise mechanism for dividing the cache memory into different partitions dynamically. Our primary design exploits set associativity in conventional cache organizations.

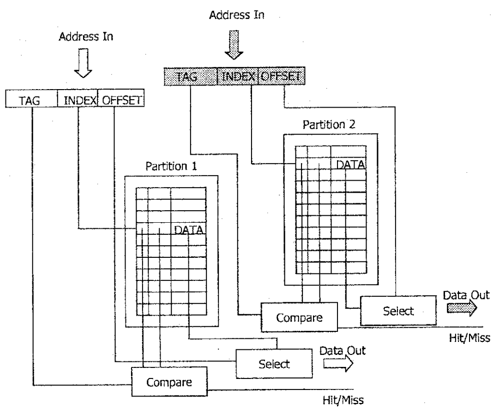

Fig. 1 Reconfigurable cache based on 2-way set associative.

The reconfigurable cache can be built on set associativity in natural way. That is, we divide a cache memory into partitions at the granularity of ways in a set associative cache. The example of exploiting the set associativity for a reconfigurable cache is shown in **Fig. 1**.

In our implementation we are going to modify a 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.

2.2 Detection

A reconfigurable cache needs to know when it should act as a normal cache 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 actually possible, we select software-controlled approach. We are going to provide a special register whose value invokes the reconfiguration. This special register is set by execution of the special instructions. That is, applications themselves control the configuration of the cache according to their decision. Resetting the register restores the cache configuration to a normal cache mode.

2.3 Addressing

In a conventional cache, a middle portion (index field) of a memory address 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 (tag field) of the address to determine whether the entry corresponds to the 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 partition 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 a 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.

2.4 Data Consistency

In one of conventional caches, when a write occurs, the new value is written only to a 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 dirty blocks in the partition into the next level cache or main memory.

There are two options, cache scrubbing and lazy transitioning in$^{3)}$. The former performs all write back operations on the partition at the same time that the reconfiguration occurs. The latter performs a write back of a block only when the entry is being overwritten, that is when a replacement is processed. 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. In our usage as FIFO, data accesses to a cache memory always search all partitions and the partitioning makes sense only when a replacement is processed. Therefore, data consistency is naturally maintained.

3. 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 a data cache the ability to switch to multiple partitions one of which is used as a FIFO buffer and cope with data accesses with no locality.

3.1 Motivation and Basic Policy

We are now focusing on an example, that is sequential accesses to data of a certain interval in memory, where each data is accessed only once. Although they are sequential they are not continuous resulting in fragmentation in a 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 as a unit which might include data being not accessed. This kind of data accesses can be found often in, for example, database query 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 (16-32 bytes) of block and cannot receive benefit from spatial locality.

As for such accesses to sequential data of a certain interval, a memory address of each data itself does not have much importance for the intention of a program in many cases. Only the fact of regular and in-order injections of sequential data into the execution of memory access instructions is essential. Conventional data caches cause inefficiency against such accesses since it depends particularly on the structure of blocks and data addresses. In order to prevent inefficiency caused by 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.

The size of FIFO needs to be large enough to deal with a large data set, especially when supported by a direct memory access mechanism described in section 3.3. A large FIFO can bridge a gap in speed between inside and outside of a processor. However, equipping a processor with a large memory dedicated to the FIFO buffer can easily increase the chip size and costs. Therefore, we implement a FIFO buffer within a partition in a reconfigurable cache. This strategy is based on the observations that a partition, or way, in a primary 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.

3.2 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 register is set, one partition is used as a general-purpose data cache and the other as FIFO. **Fig. 2** shows the configuration after it is configured as multiple partitions.

The left partition in the figure functions as a direct mapped cache and the right one as a FIFO. 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. This means that it is necessary for a compiler or linker to relocate data sets being accessed via FIFO appropriately and for a virtual memory to translate the virtual addresses to the corresponding physical ones. 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 simultaneously. Therefore, our reconfigurable cache has a single output port common between two partitions, whereas the original reconfigurable caches must have the same number of output ports as partitions.

An entry to be read in 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 entries exist in FIFO and a miss signal is asserted. Consequently, FIFO does not need to

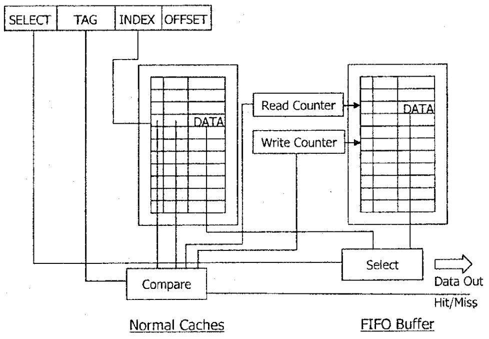

Fig. 2 Reconfigurable cache as FIFO buffer.

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 usage. In addition, the structure does not need any tag matching for searching, which can simplify the hardware.

Another feature is that only entries that are actually used as FIFO entries invade the partition. Entries in the FIFO partition that have not been pointed to by the write counter remain as normal cache ones. Therefore, all the partition is not occupied when a data set is smaller than the partition size. The surviving entries have nothing to do with data consistency since all partitions are searched in every data accesses. In order to leave as many surviving entries as possible, there is an option that the write counter goes back to a top of the partition when entries which the read counter has passed are found. This can make the size of FIFO as small as possible, although the effectiveness of this scheme depends on a speed gap between read and write operations and the write pointer might not be able to increase when it meets the read pointer after going around. Our implementation does not select this option because of the simplicity of circuits.

3.3 Cooperation with Bus Transfer Mechanism

Sequential memory access instructions (load instructions) take a data out of FIFO, while data read from an external memory are to be inserted into FIFO. Although our FIFO buffer aims chiefly at avoiding fragmentation in a conventional cache, the mechanism exhibits its power only when it absorbs a gap between the speed of execution of load instructions and latency of accesses to an external memory. One of options is the introduction of a prefetch instruction. Execution of the instruction can inject data to be requested into FIFO in advance. However, the execution of the instruction passes through pipeline resources and therefore leads to overheads.

Some support mechanism should be provided to make full use of FIFO. In this study, we assume that there is one of DMA mechanisms outside a processor, called "stride data transfer (SDT)"[5], which can cooperate with FIFO in the processor. We describe the outline of SDT and relationship between the SDT and FIFO.

One of typical applications which show no locality about data access is database processing. When columns with an attribute in a relation table is looked over, data located at the interval that corresponds to the tuple size are accessed sequentially. DRAM is constructed in rectangle array of memory cells where data 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 sends them to DRAM one by one. Although general memory controllers can only read and write continuous and adjacent 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 requested 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 for the target data set finds a valid entry in FIFO, it is served by FIFO and the read counter is increased by one. When it 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 and the value of the address space identifier, it dispatches row and column addresses to DRAM and then 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 interval, dispatches it to DRAM, decrements the value of the register indicating the number of data, and then sends a read data 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 FIFO and the write counter is increased.

It is possible for FIFO misses to occur when an load instruction is executed before the corresponding data arrives at FIFO. The processor recognizes this as a 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 FIFO are consumed.

4. Evaluation

We have designed a CPU core in VHDL. By using this VHDL descriptions and a VHDL simulator (ModelSim), we performed RTL simulation. In this section we will first describe the base of our evaluation of reconfigurable caches and FIFO buffer. Then we present the benchmark programs, and show results.

4.1 Processor and cache Models

The integer unit of the baseline system is a load/store architecture that executes MIPS1 instructions, subset of MIPS32, through a five-stage pipeline (IF,ID,EX,MEM,WB). For the instruction cache, we assume that it is an ideal instruction cache (thus no I-cache misses). The data cache is a 2-way set associative cache whose block size is 16 byte. The total capacity of the data cache is 8 KBytes. Data replacement is base on LRU. The data cache is accessed in MEM stage of the pipeline. In the RTL simulation, we made a miss penalty take 12 clock cycles. We equipped this basic data cache with the ability to function as a FIFO buffer in the way. When the FIFO configuration is simulated, we assumed that a memory system implemented the SDT mechanism as described in section 3.3.

4.2 Benchmarks

We used two programs that includes simple loop structure to evaluate the effectiveness of the FIFO buffer mechanism.

Program 1

The first program computes the summation of data at some stride in an integer array. The codes is as follows.

```
#define N
#define STRIDE
main(){
  int i,sum=0;
  int vector[N];
  for(i=0; i<N; i+=STRIDE)
    sum += vector[i];
}
```

The program was executed for various N values and for STRIDE whose value is 2 or 4.

BenchMark Program 2

The second program is something like a query processing in a database. The array TAB is scanned and a member "year" of each tuple is checked. The codes is as follows.

```
#define N
main(){
  struct {
    int age;
    int year;
#if BODY
    int height;
    int weight;
#endif
  } TAB[N];
  int i,hit=0;
  for(i=0; i<N; i++)
    if(TAB[i].year == 2003)
      hit++;
}
```

The program was executed for various N values and for BODY whose value is 0 or 1.

4.3 Results

As for program 1, we obtained results when STRIDE was 2 or 4 that are shown in **Table 1** and **Table 2**, respectively. When STRIDE was 2, cache misses occurred every two successive accesses to the array in a normal data cache. Similarly, when STRIDE was 4, they occurred every four accesses. On the other hand, in FIFO method, the pipeline waited a return from a memory only when it referred to the first integer, and could get all the succeeding integers without stalling. The number of cycles that the method took depended on the number of accesses (N/STRIDE), not on STRIDE as seen in the results. The FIFO method could reduce about 15 to 18% of the execution time when STRIDE was 2, and 30 to 33% when STRIDE was 4.

The results of program 2 when the number of members in a tuple was 2 or 4 are shown in **Ta-**

Table 1 Results of Program 1 (STRIDE=2).

N	Cache	Cycles	Reduction[%]
64	Normal	976	15.10
	FIFO	829	
128	Normal	1920	16.80
	FIFO	1597	
256	Normal	3808	17.70
	FIFO	3133	
512	Normal	7584	18.20
	FIFO	6205	
1024	Normal	15136	18.40
	FIFO	12349	
2048	Normal	30253	18.60
	FIFO	24637	

Table 2 Results of Program 1 (STRIDE=4).

N	Cache	Cycles	Reduction[%]
128	Normal	1184	30.00
	FIFO	829	
256	Normal	2336	31.60
	FIFO	1597	
512	Normal	4640	32.50
	FIFO	3133	
1024	Normal	9248	32.90
	FIFO	6205	
2048	Normal	18464	33.10
	FIFO	12349	
4096	Normal	36896	33.20
	FIFO	24637	

Table 3 Results of Program 2 (BODY=0).

N	Cache	Cycles	Reduction[%]
32	Normal	972	16.50
	FIFO	812	
64	Normal	1916	17.50
	FIFO	1580	
128	Normal	3804	18.10
	FIFO	3116	
256	Normal	7580	18.40
	FIFO	6188	
512	Normal	15132	18.50
	FIFO	12332	
1024	Normal	30236	18.60
	FIFO	24620	

Table 4 Results of Program 2 (BODY=1).

N	Cache	Cycles	Reduction[%]
32	Normal	1180	31.20
	FIFO	812	
64	Normal	2332	32.20
	FIFO	1580	
128	Normal	4636	32.80
	FIFO	3116	
256	Normal	9244	33.10
	FIFO	6188	
512	Normal	18460	33.20
	FIFO	12332	
1024	Normal	36892	33.30
	FIFO	24620	

ble 3 and **Table 4**, respectively. When BODY was 0, cache misses occurred every two successive accesses to the tuple array in a normal data cache. Every four tuple accesses caused a cache miss when BODY was 1. The results are almost the same as those of program 1. Therefore, The FIFO method achieved similar improvement as seen in the results.

5. Conclusion

While the use of large caches is common across general-purpose systems, it is often ineffective for some applications, for example, media processing and database since they might exhibit little locality in data accesses.

Reconfigurable caches aim at the effective use of cache memory space when conventional caches cannot 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. Furthermore, it becomes more effective when it cooperates with a DMA mechanism such as the SDT.

Simulations using the CPU core we designed showed that the execution time could be reduced by 33% for programs that included a simple loop and accessed an array at some interval.

Acknowledgments

In this study, we are using tools provided by university programs of Synopsys, Inc. and Model Technology, Inc. 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) P.Ranganathan, S.Adve and N.P.Jouppi, "Reconfigurable Caches and Their Application to Media Processing." Proc. of ISCA, pp.214–224, 2000.
4) P.Ranganathan, S.Adve, N.P.Jouppi, "Perfomance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions." Proc. of ISCA, pp.124–135, 1999.
5) T.Fukawa, K.Tanaka and J.Miyazaki, "The Highly Functional Memory Controller for Main Memory Database." IPSJ SIG Notes, ARC, Vol.2002, No.112, pp.77–82, 2002.