This is the accepted manuscript of a paper published in
*Proc. Int'l Symp. on System-on-Chip 2014 (SoC2014)*


# An Implementation of Auto-Memoization Mechanism on ARM-based Superscalar Processor

Yuuki SHIBATA*, Takanori TSUMURA*, Tomoaki TSUMURA* and Yasuhiko NAKASHIMA†

*Nagoya Institute of Technology, Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp
†Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara, Japan
Email: nakashim@is.naist.jp

***Abstract*—We have proposed a processor called Auto-Memoization Processor which is based on computation reuse. Until now, we have implemented the auto-memoization mechanism on a single-issue non-pipelined SPARC processor and studied the processor. The processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically. In addition, the processor can apply computation reuse to the blocks with a little reuse overhead. However, the fine evaluation result of the processor may not guarantee enough practicality. This is because instead of such a simple architecture, superscalar architectures are now widely used for generic processors for PCs, embedded processors, and other various processors. Hence, we examine problems which will be caused in the case of implementing the auto-memoization mechanism on an ARM-based superscalar processor and design the ARM-based Auto-Memoization Processor. For example, one of such problems is that pipeline stalls are caused because of the reuse overhead. To solve this problem, we implement a mechanism for overlapping the reuse overhead and the pipeline execution of the processor. The evaluation result with SPEC CPU95 benchmark suite shows that the ARM-based Auto-Memoization Processor can also achieve speed-up as well as the previous SPARC-based Auto-Memoization Processor. In this paper, we describe the implementation and the evaluation result of the ARM-based Auto-Memoization Processor.**

***Index Terms*—microprocessor architecture, computation reuse, memoization, Auto-Memoization Processor, superscalar.**

## I. Introduction

So far, various speed-up techniques for microprocessors have been proposed. The performance of microprocessors had been controlled by the gate latencies, and it had been relatively easy to speed-up microprocessors by transistor scaling. However, the interconnect delay has been going major, and it has become difficult to achieve speed-up only by higher clock frequency. Therefore, speed-up techniques based on ILP (Instruction-Level Parallelism), such as superscalar or SIMD instruction sets, have been counted on. Traditional speed-up techniques mentioned above are all based on some parallelisms in different granularities.

Meanwhile, we have proposed a processor called Auto-Memoization Processor based on computation reuse [1] [2]. Auto-Memoization Processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically. The processor can apply computation reuse to the blocks with a little reuse overhead. As a speed-up technique, memoization has no relation to parallelism of programs. It depends upon value locality, especially input values of functions or loops. Therefore, memoization can achieve speed-up on the programs which do not have much ILP.

Until now, we have studied Auto-Memoization Processor which is based on a single-issue non-pipelined SPARC architecture. However, the fine evaluation result of the SPARC-based Auto-Memoization Processor may not guarantee enough practicality. This is because instead of such a simple architecture, superscalar architectures are now widely used for generic processors for PCs, embedded processors, and other various processors. Hence, at first, we focus on an ARM-based superscalar architecture which is not only widely used for embedded processors but also being used for server processors. Next, we examine problems which will be caused in the case of implementing the auto-memoization mechanism on an ARM-based superscalar processor. Finally, we solve the problems and design the ARM-based Auto-Memoization Processor. For example, one of such problems is that pipeline stalls are caused because of the reuse overhead. To solve this problem, we implement a mechanism for overlapping the reuse overhead and the pipeline execution of the processor. Another problem is that functions can not be detected as reusable blocks easily, and to solve this problem, we implement the decoders for monitoring function calls and function returns. Moreover, it is also one of such problems that some of inputs of a reusable block are lost before the retire stage, and to solve this problem, we extend pipeline registers. In this paper, we describe the implementation and the evaluation result of the ARM-based Auto-Memoization Processor.

## II. Related Work

A general-purpose reuse technique for single instructions with fully associative reuse buffer was proposed by Sodani [3]. Each reuse buffer entry holds the operand values and the result of an instruction execution. The mem-valid bit and the mem-address fields are maintained to ensure a load instruction can be reused. On a store the mem-address fields are associatively searched and the mem-valid bits are reset.

González [4] evaluated reuse trace memory (RTM) with maximum 256K entries. Each entry is indexed by part of PC (e.g., 8-way set-associative and 16 entries per PC) and

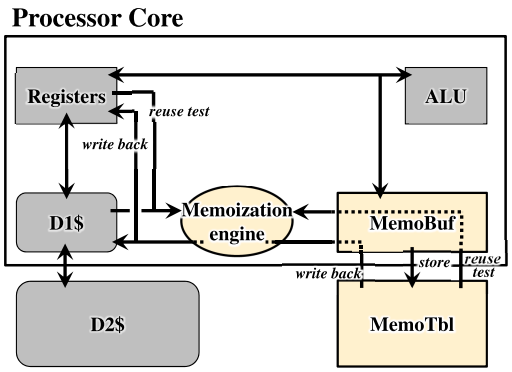



Fig. 1. Structure of Auto-Memoization Processor.

is assumed to hold 8-in 8-out register values and 4-in 4-out memory values. The maximum size of RTM is over 32MB. This work provides the upper bounds achieved with an infinite reuse trace memory.

Costa [5] proposed a reuse scheme which employs a fully associative table that does not include load/store instruction. The program counter and the operand values are associatively compared. This work approaches trace-level reuse by providing an evaluation of a feasible reuse mechanism against [4].

There are also works which focused on only load/store instructions. Yang [6] focused on load-load/store-load redundancy and reuse the address and/or value by maintaining a load-table which holds past addresses and values. Moreover, Onder [7] proposed non-speculative load and store reuse technique. CAM based value-address association structure (VAAS) is maintained for address associations of loaded and stored values present in the physical register file. This structure enables store-to-load and load-to-load forwarding.

In contrast to these studies, Auto-Memoization Processor, which we have proposed, dynamically detects functions and loop iterations as reusable blocks and memoizes them automatically. In addition, Auto-Memoization Processor executes traditional load-modules faster without any software assist.

## III. Research Background

In this section, we describe the previous SPARC-based Auto-Memoization Processor as the background of our study.

### A. Auto-Memoization Processor

**Computation reuse** is a well-known speed-up technique in the software field. It is storing the input sequences and the results of some computation blocks, such as functions, for later reuse and avoiding recomputing them when the current input sequence matches one of the past input sequences. It is called **memoization** [8] to apply computation reuse to computation blocks in programs.

Auto-Memoization Processor [1], which we have proposed, executes traditional load-modules faster without any software assist. There is no need to rewrite or recompile programs. This processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically.

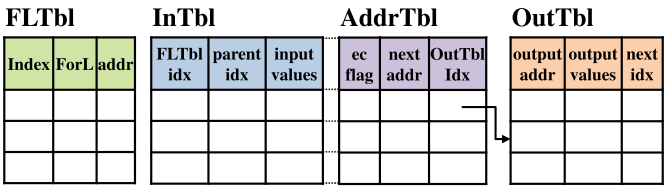



Fig. 2. Structure of MemoTbl.

The structure of Auto-Memoization Processor is depicted briefly in Fig.1. Auto-Memoization Processor consists of the memoization engine, **MemoTbl** and **MemoBuf**. MemoTbl is a set of tables for storing input/output sequences of past executed computation blocks, or instruction regions. MemoBuf works as a write buffer for MemoTbl.

Entering to a memoizable region, the processor refers to MemoTbl and compares the current input sequence with the past input sequences which are stored in MemoTbl. In the following, we call this comparison 'reuse test.' If the current input sequence matches one of the stored input sequences on MemoTbl, the memoization engine writes back the stored outputs, which are associated with the matched input sequence, to the registers and caches. This omits the execution of the region and reduces the total execution time.

If the current input sequence does not match any past input sequence, the processor stores the current inputs and outputs of the region into MemoBuf while executing the region as usual. The input sequence consists of the register/memory values which are read over the region, and the output sequence consists of the values which are written. If the region is a function, its return value is also included in the output sequence. Reaching the end of the region, the memoization engine stores the content of MemoBuf into MemoTbl for future reuse.

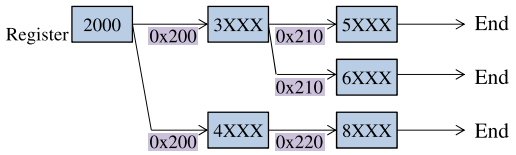
MemoBuf has multiple entries, and each MemoBuf entry corresponds to one input/output sequence. Each MemoBuf entry has a stack pointer (*SP*), a return address (*retOfs*), an input sequence (*Read*), and an output sequence (*Write*). Moreover, MemoBuf has a stack-like structure, and can manage nested computation blocks. *MemoBuf_top*, the stack pointer for MemoBuf, is incremented when Auto-Memoization Processor detects a new computation block, and is decremented when Auto-Memoization Processor reaches the end of a block.

The structure of MemoTbl is shown in Fig.2. MemoTbl consists of four tables:

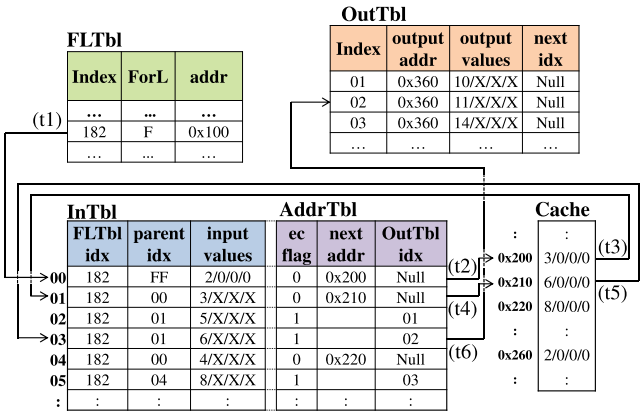
**FLTbl:** for start addresses of computation blocks.
**InTbl:** for input value sets of computation blocks.
**AddrTbl:** for input address sets of computation blocks.
**OutTbl:** for output data sets of computation blocks.

FLTbl, AddrTbl, and OutTbl are implemented with RAM. On the other hand, InTbl is implemented with a ternary CAM (Content Addressable Memory), so that input values can be found fast by associative search.

Each FLTbl line corresponds to a reusable computation block. Each FLTbl entry has whether the block is a function or a loop (*ForL*) and the start address of the block (*addr*).




(a) Tree of input sequences.




(b) Input matching flow on MemoTbl.

Fig. 3. Input sequences and Input matching flow.

Each InTbl entry has an index for FLTbl (*FLTbl idx*), which represents the associated instruction region, or computation block, of the input stored in the entry, and holds input values (*input values*). Because each InTbl entry can hold single cache line, an input sequence over multiple cache lines is registered onto InTbl by using several entries. Hence, each InTbl entry also has an index key for its parent entry (*parent idx*). Incidentally, When a variable is read as an input value, Auto-Memoization Processor stores its whole cache line in an InTbl entry, masking unreferred values with *don't care* bits.

AddrTbl has the same number of entries as InTbl, and each AddrTbl entry corresponds to the InTbl entry which has the same index. Each AddrTbl entry has an input address which should be tested next (*next adr*). Each AddrTbl entry also has a flag (*ec flag*), which shows whether it is the terminal entry of an input sequence, and if it is terminal, it has a valid pointer (*OutTbl idx*), which refers to an OutTbl entry for associated outputs.

Each OutTbl entry has FLTbl idx, addresses (*output addr*) and values (*output values*) of an output sequence. Each OutTbl entry also has an index for next OutTbl entry (*next idx*) because an output sequence may be stored over multiple OutTbl entries.

### B. Execution Mechanism

Now, an input sequence for a certain instruction region can be represented as a sequence of tuples, each of which contains an address and a value. In a certain instruction region, the series of input addresses sometimes branch off from each other. For example, after a branch instruction, what address will be referred next relies on whether the branch was taken or untaken. Therefore, the universal set of the different input sequences for an instruction region can be represented as a multiway input tree. Here, input sequences of a memoizable region are represented as a way from its root to a leaf on this tree. Hence, Auto-Memoization Processor should hold input sequences as a tree structure.

Fig.3(a) shows an example of a tree structure of input sequences for a instruction region. Each node of the tree represents input values, and each edge represents the address which should be referred next. Here, `End` represents the terminal of a sequence. Fig.3(b) shows how the input sequences shown in Fig.3(a) are stored into InTbl/AddrTbl. In this figure, `X` in *input values* represents *don't care* bits in the cache line and will not be tested for computation reuse, and the entry whose *parent idx* is `FF` represents a root entry of a input sequence. Fig.3(b) also shows an input matching flow on MemoTbl as (t1)...(t6).

First, Auto-Memoization Processor reads the values of registers when the start address of a reusable instruction region is detected. Then, the processor searches the root entry whose *parent idx* is `FF` and whose *input values* match the values on the current registers. Now, the line `00` matches (t1). Next, the address of `0x200` is read because *next addr* of the entry `00` in AddrTbl indicates `0x200` (t2). Then, the processor searches the entry whose *parent idx* is `00` and whose *input values* match the values of `0x200` (t3). This process is applied repeatedly until a mismatch of input values occurs or the processor finds the entry whose *ec flag* is set (t4)(t5). In this example, as *ec flag* of the entry `03` in AddrTbl is set, that is, the processor detects the terminal of a input sequence, reuse test for the current input values succeeds. Therefore, the processor can get the output values by using the index of *OutTbl idx* stored in the terminal entry (t6). Finally, the processor writes back the output values to the registers and caches. This omits the execution of the instruction region and reduces the total execution time.

## IV. Superscalar Auto-Memoization Processor

In this section, we describe two problems of the SPARC-based Auto-Memoization Processor and the pipeline execution flow on the ARM-based Auto-Memoization Processor.

### A. Problems of SPARC-based Auto-Memoization Processor

Until now, we have studied Auto-Memoization Processor which is based on the SPARC architecture [9]. However, this SPARC-based Auto-Memoization Processor has two problems.

One problem is that the SPARC architecture is a single-issue non-pipelined architecture. However, single-issue non-pipelined architectures are now not widely used for processors. Recently, superscalar architectures which are multiple-issue pipelined architectures are widely used for generic processors for PCs, embedded processors, and other various processors. Generally, superscalar architectures have more complex hardware structures than single-issue non-pipelined architectures. Therefore, we have to implement the auto-memoization mechanism on a superscalar processor and evaluate the performance of the superscalar-based Auto-Memoization Processor. This

implementation enables us to evaluate the practical performance of Auto-Memoization Processor.

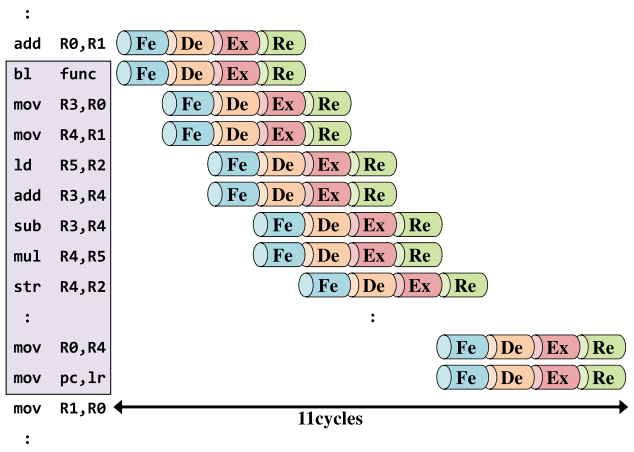
The other problem is that the auto-memoization mechanism of the SPARC-based Auto-Memoization Processor depends on characteristics of the SPARC architecture. For example, SPARC instruction set which is derived from RISC lineages, SPARC ABI, and register windows are convenient for the auto-memoization mechanism (e.g., a mechanism for detecting computation blocks as reusable blocks). In other words, it is unknown whether the auto-memoization mechanism can be implemented with base architectures other than the SPARC architecture or not. To solve this problem, we adopt the ARM architecture [10] and implement the auto-memoization mechanism on the ARM-based superscalar processor. The ARM architecture is also categorized into RISC architecture, but the ARM architecture has more complex instructions than the SPARC architecture. In addition, the ARM architecture is not only widely used for embedded processors but also being used for server processors [11]. Therefore, if the ARM-based Auto-Memoization Processor can be implemented, the practicality of Auto-Memoization Processor can be more generally confirmed. Moreover, we can acquire some knowledge about merging the superscalar which is one of the ILP-based methods and the memoization mechanism.

### *B. Pipeline Execution Flow on ARM-based Auto-Memoization Processor*

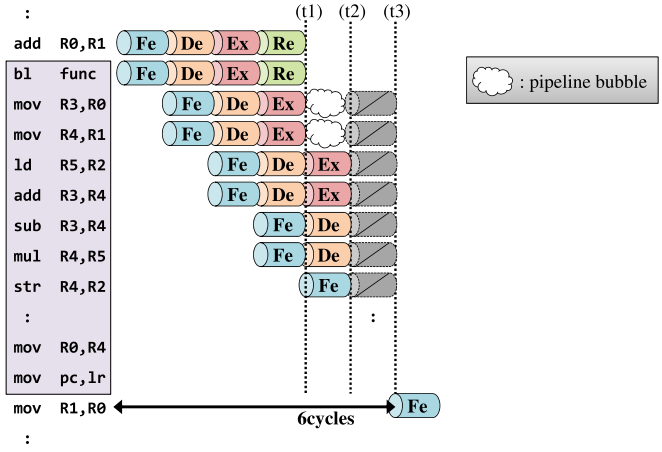
Fig.4 shows three different situations of the pipeline execution flow on the ARM-based Auto-Memoization Processor. In the following, we describe each situation.

*1) Pipeline execution without Memoization:* First, Fig.4(a) shows a situation of the pipeline execution flow without memoization. Here, for simplicity, assume that ARM-based Auto-Memoization Processor equips two pipelines and each pipeline has four stages, *Fe* (fetch), *De* (decode), *Ex* (execution) and *Re* (retire). In addition, assume that each stage is executed in 1 cycle, and a pipeline stall is not caused by a cache miss and so on. In this figure, the instruction (`bl func`) represents the instruction for a function call and the instruction (`mov pc,lr`) represents the instruction for a function return, that is, the region from (`bl func`) to (`mov pc,lr`) is a function region. This figure shows that the processor executes the function without pipeline stalls. Incidentally, 11 cycles are required for the function execution.

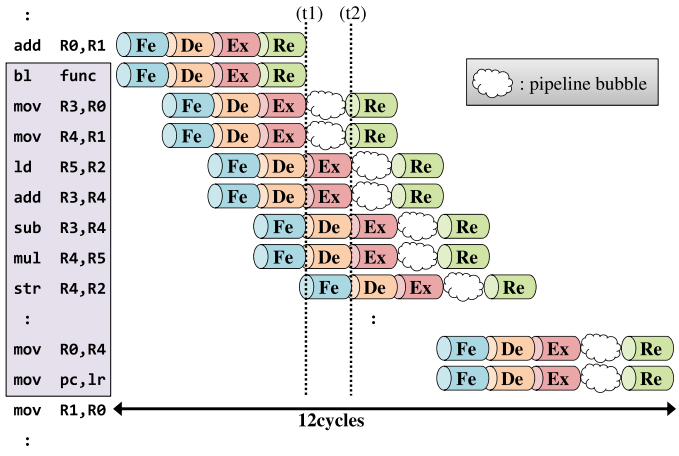
*2) Pipeline execution when succeeding in reuse test:* Next, Fig.4(b) shows a situation of the pipeline execution flow in the case reuse test succeeds. When the ARM-based Auto-Memoization Processor detects an instruction for a function call, the processor tries reuse test for applying computation reuse to the detected function. At this time, the values which should be compared as inputs of the function have to be written back to the registers and caches in order to be used for trying reuse test correctly. When the instruction for a function call is retired, it is guaranteed that the processor already finishes executing instructions prior to the instruction for a function call. In other words, it is guaranteed that the values which will be compared as inputs of the function are already written back to the registers and caches. Therefore, we design that the processor tries reuse test when an instruction for a function call is retired (t1). Then, the processor executes the instructions in the function with stalling the retire stage while trying reuse test. This is because input values which should be compared




(a) Pipeline execution flow without memoization.




(b) Pipeline execution flow when succeeding in reuse test.




(c) Pipeline execution flow when failing in reuse test.

Fig. 4. Pipeline execution flow.

in reuse test can be rewritten if the processor retires the instructions in the function while trying the reuse test. This rewrite of input values can cause false computation reuse of the function.

Then, if the processor succeeds in reuse test, the processor writes back the stored outputs, which are associated with the matched input sequence, to the registers and caches (t2). At the same time, the instructions in the pipeline should be flushed. This is because the instructions in the pipeline belong to the function whose execution is omitted by computation reuse, and these instructions have to be invalidated. At this time, by flushing the pipeline, pipeline bubbles are assumed to appear in each cycle after the processor tries to fetch the instruction (`mov R3,R0`). These pipeline bubbles may negate the performance gain by computation reuse. Therefore, we will describe a method for solving this problem in section VI-B.

Then, the processor fetches the instruction of the return destination (`mov R1,R0`) and continues to execute instructions (t3). In Fig.4(b), by omitting the function execution, 6 cycles are required for the function execution. This situation shows that the execution time can be reduced by 5 cycles as compared with the case without memoization shown in Fig.4(a).

*3) Pipeline execution when failing in reuse test:* Finally, Fig.4(c) shows a situation of the pipeline execution flow in the case reuse test fails. As mentioned above, the processor tries reuse test when the instruction for a function call is retired (t1). Then, the processor executes the instructions in the function with stalling the retire stage while trying reuse test. Here, if the processor fails in reuse test, the retire stage is restored from stall and the processor continues to execute instructions (t2). In Fig.4(c), 12 cycles are required for the function execution. This situation shows that the execution time increases by 1 cycle as compared with the case without memoization shown in Fig.4(a). This is because the processor executes instructions with stalling retire stage while trying reuse test.

## V. Implementation

In this section, we describe the base architecture of the ARM-based Auto-Memoization Processor and the hardware extension for memoization.

### A. Base Architecture

Fig.5 shows the brief structure of the ARM-based Auto-Memoization Processor. The pipeline stages are as follows;

**IA** (Instruction Address)
: The address whose instruction should be executed next is calculated.

**IF** (Instruction Fetch)
: Two consecutive instructions are fetched from the instruction cache. In addition, the destination of a branch instruction is predicted with *g-share*.

**ARM-D** (Arm-Instruction Decode)
: Fetched instructions are decoded. In addition, an instruction for a function call and an instruction for a function return are detected by the memoization decoder (*M1-D*) which is described in section V-B.

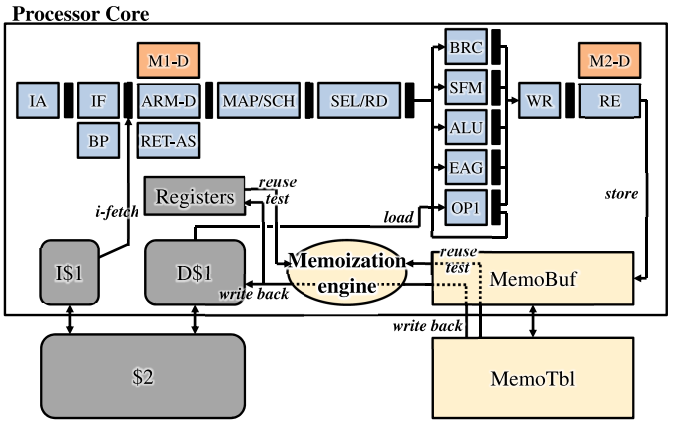



Fig. 5. Structure of the ARM-based Auto-Memoization Processor.

**MAP/SCH** (Register mapping/Schedule)
: Logical registers of instruction operands are mapped into the reorder buffer which is combined with an instruction window and physical registers.

**SEL/RD** (Select and Read)
: The instructions which do not depend on the other instructions in the reorder buffer are issued out-of-order.

**IE** (Instruction Execution)
: The base architecture has five-parallel execution stages in the pipeline.
  - **BRC** For branch instructions.
  - **SFM** For shift operation instructions.
  - **ALU** For arithmetic instructions.
  - **EAG** For address calculation.
  - **OP1** For load instructions and store instructions.

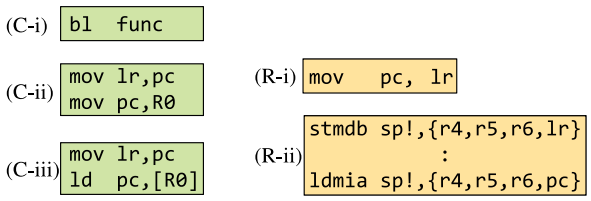
**WR** (Writeback)
: The result of the instruction which is executed at each instruction execution stage is written back to the reorder buffer.

**RE** (Retire)
: The instruction whose execution is finished is retired in order. In addition, an instruction for a function return is detected by the memoization decoder (*M2-D*) which is described in section V-B.

### B. Hardware Extension for Memoization

In this section, we discuss the additional hardware units for memoization.

*1) Decoders for Memoization:* Auto-Memoization Processor must monitor function calls and function returns in order to detect functions as reusable instruction regions. In the ISA of SPARC, it is defined that a call instruction is '`call`' and a return instruction is '`ret`.' Therefore, the SPARC-based Auto-Memoization Processor can easily detect functions by only monitoring these two instructions. On the other hand, in the ISA of ARM, specific instructions, which should be used for a function call and a function return, are not defined. Here, Fig.6 shows the call codes and return codes in the ARM binaries.




(a) Call codes. (b) Return codes.

Fig. 6. Call codes and return codes in ARM binaries.

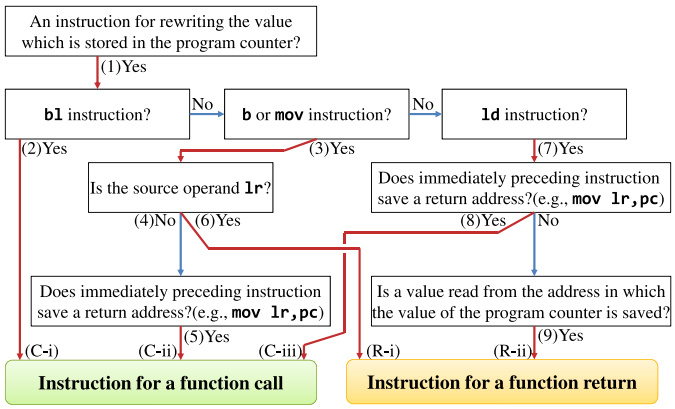



Fig. 7. How to detect function calls and function returns.

As shown in this figure, there are several call codes and return codes. Therefore, the ARM-based Auto-Memoization Processor can not detect functions by merely monitoring specific instructions in the same way as the SPARC-based Auto-Memoization Processor. Hence, we add the decoders (*M1-D* and *M2-D*) for monitoring function calls and function returns.

Fig.7 shows how to detect function calls and function returns. Now, we describe how these decoders detect the call codes and the return codes shown in Fig.6. First, an instruction for rewriting the value which is stored in the program counter is monitored by *M1-D* (1). Depending on the monitored instruction, how the decoders detects the call codes and the return codes is divided into three patterns. If the monitored instruction is `bl` (2), the ARM-based Auto-Memoization Processor detects the instruction as an instruction for a function call (C-i). Next, if the monitored instruction is `b` or `mov` (3), the source operand of the instruction is checked. In the case the source operand is other than *link register* (`lr`) where a return address is held (4), the immediately preceding instruction is checked. If the checked instruction is an instruction which saves a return address (e.g., `mov lr,pc`) (5), the processor detects the monitored instruction as an instruction for a function call (C-ii). On the other hand, in the case the source operand is *link register* (`lr`) (6), the processor detects the instruction as an instruction for a function return (R-i). Finally, if the monitored instruction is `ld` (7), the immediately preceding instruction is checked. If the checked instruction is an instruction which saves a return address (e.g., `mov lr,pc`) (8), the processor detects the monitored instruction as an instruction for a function call (C-iii). Incidentally, after the return address is saved in the stack at the function prologue, the return address can be rewritten in the program counter by a load instruction at the function epilogue. In order to detect such a load instruction as an instruction for a function return, the address in which the value of the program counter is saved at function prologue is recorded in *M2-D*. After that, if a load instruction for reading the value of the program counter from the address is found (9), the processor detects an instruction as an instruction for a function return (R-ii). By using these decoders, the ARM-based Auto-Memoization Processor can detect functions as reusable instruction regions.

*2) Pipeline Register for Memoization:* In order to apply computation reuse to the functions which are detected as reusable instruction regions, Auto-Memoization Processor has to store inputs and outputs of each function in MemoBuf and MemoTbl. Here, the inputs and outputs are a set of values and addresses which are read and written in the function. The ARM-based Auto-Memoization Processor stores the inputs and outputs at the retire stage. This is because a problem is caused if the processor tries to store the inputs and outputs at the stages prior to the retire stage. The problem is that in the case the executed instructions are flushed by branch prediction misses and so on, large overhead will be caused by invalidating the MemoBuf entries corresponding to the flushed instructions. Therefore, it is not realistic that the processor tries to store the inputs and outputs at the stages prior to the retire stage. However, even if the inputs and outputs will be stored at the retire stage, another problem occurs. The problem is that some of inputs of a function can not be stored in MemoBuf at the retire stage. Specifically, source operand values of a load instruction can not be stored as inputs of a function at the retire stage because the values are lost before the retire stage.

Here, we describe why some of inputs of a function are lost before the retire stage and how to solve the problem. For example, when the instruction (`ld R0,[R1]`) is processed at the instruction execution stage, the value which is stored in the address indicated by `R1` is read into the physical register corresponding to logical register `R0`. Then, the value in the physical register is written into logical register `R0` at the retire stage. At this time, the address value which is stored in source operand `R1` has been lost from the reorder buffer because the address value is not required for retiring its instruction. Therefore, the address value can not be stored in MemoBuf as the input of the function at the retire stage. In order to solve the problem, we extend pipeline registers. Specifically, we design that values of source operands are held and sent to the next stage at the instruction execution stage, the writeback stage, and the retire stage as well as the other stages. The values of source operands are held in the extended pipeline registers and sent to the retire stage through the extended pipeline registers. Thereby, all inputs of functions can be stored at the retire stage.

TABLE I
SIMULATION PARAMETERS.

| | |
|---|---|
| MemoBuf | 64 KBytes |
| MemoTbl CAM | 256 KBytes |
| Comparison (register and CAM) | 1 cycle / 64Bytes |
| Comparison (Cache and CAM) | 2 cycles / 64Bytes |
| Writeback (MemoTbl to Reg. / Cache) | 1 cycle / 64Bytes |
| L1 I-cache | 16 KBytes |
| line size | 64 Bytes |
| ways | 4 ways |
| miss penalty | 8 cycles |
| L1 D-cache | 32 KBytes |
| line size | 64 Bytes |
| ways | 4 ways |
| miss penalty | 8 cycles |
| L2 cache | 2 MBytes |
| line size | 64 Bytes |
| ways | 4 ways |
| miss penalty | 40 cycles |
| pipeline stage | |
| IA (Instruction Address) | 1 insn / cycle |
| IF (Instruction Fetch) | 2 insns / cycle |
| ARM-D (ARM-Instruction Decode) | 2 insns / cycle |
| MAP/SCH (Register Mapping/Schedule) | 2 insns / cycle |
| SEL/RD (Select and Read) | 2 insns / cycle |
| IE (Instruction Execution) | 1 insn / cycle |
| WR (Writeback) | 1 insn / cycle |
| RE (Retire) | 2 insns / cycle |
| Reorder Buffer | 32 entries |

## VI. PERFORMANCE EVALUATION

We have implemented the auto-memoization mechanism on a simulator for an ARM-based superscalar processor. This section describes the evaluation results with the simulator.

### A. Simulation Environment

We have developed an in-house simulator of an ARM-based superscalar processor and implemented the auto-memoization mechanism on the simulator. The simulation parameters are shown in TABLE I. The on-chip CAM for InTbl in MemoTbl is modeled on DC18288 [12] (64Bytes $\times$ 4K lines). We assume the latency for input matching between CAM and the register as 1cycle / 64Bytes, and the latency between CAM and main memory as 2cycles / 64Bytes. In this paper, we have evaluated the performance of the ARM-based Auto-Memoization Processor with applying computation reuse to only functions. This is because we have not still implemented the mechanism for memoizing loop iterations.

### B. Evaluation with SPEC CPU95

We have evaluated the performance of following two processors;

- (N) No-memoization processor (baseline).
- (A) ARM-based Auto-Memoization Processor.

Here, no-memoization processor (N) represents the ARM-based superscalar processor without memoization mechanism. Workloads are benchmark programs in SPEC CPU95 INT suites with 'train' dataset. The benchmark programs are compiled by gcc version 4.1.1 with '-O2 -msoft-float -march=armv4' option, and linked statically. In this section, we compare the performance of (N) with (A).

TABLE II
REDUCTION RATE OF THE EXECUTED INSTRUCTION COUNTS.

| 099.go | 124.m88ksim | 129.compress | 130.li | 132.ijpeg | 134.perl | 147.vortex |
|---|---|---|---|---|---|---|
| 6.4% | 29.0% | 1.7% | 10.8% | 1.6% | 7.5% | 39.2% |

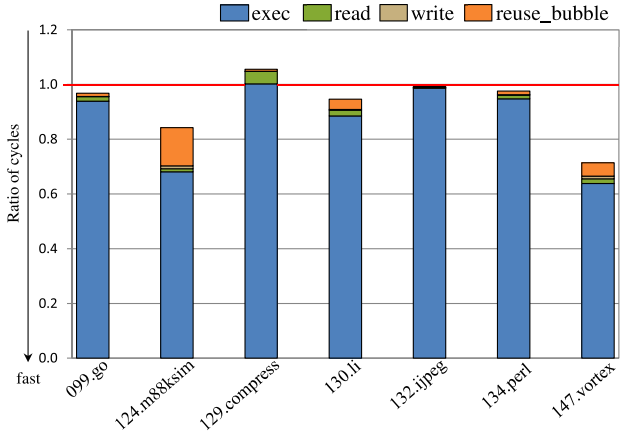



Fig. 8. Ratio of cycles (SPEC CPU95 INT).

At first, TABLE II shows the reduction rate of the executed instruction counts with each benchmark program on (A). This result shows that by omitting the execution of functions, the executed instruction counts are reduced with all programs as compared with (N). Here, (A) reduces the executed instruction counts by 13.7% in average. Therefore, the execution cycles will be reduced and the IPC will be increased.

Next, the ratio of execution cycles of (A) are shown in Fig.8. Each bar in this figure is normalized to the number of the execution cycles of (N). The legend in the figure shows the breakdown items of total cycles. They represent the executed instruction cycles ('**exec**'), the comparison overhead between CAM and registers or the caches ('**read**'), the write-back overhead ('**write**'), the pipeline bubble penalty which is described in section IV-B ('**reuse_bubble**'). Incidentally, 'exec' also includes the first-level and the second-level cache miss penalties and the cycles which are required for waiting until the busy cache controller becomes idle in order to try reuse test.

As shown in this figure, (A) reduces the execution cycles by 28.6% in maximum and by 7.9% in average. These reduction rates are almost the same as the SPARC-based memoization processor. This result means that the ARM-based Auto-Memoization Processor can also achieve speed-up as well as the SPARC-based Auto-Memoization Processor. However, 'reuse_bubble' cycles occupy no less than about 20% of total cycles with 124.m88ksim. This means that the pipeline bubbles negate the performance gain by computation reuse. Here, we discuss how we solve this pipeline bubble problem. The ARM-based Auto-Memoization Processor executes instructions in a function while trying reuse test for the function. Therefore, when the processor fails in the reuse test, the processor can continue to execute the function without flushing instructions

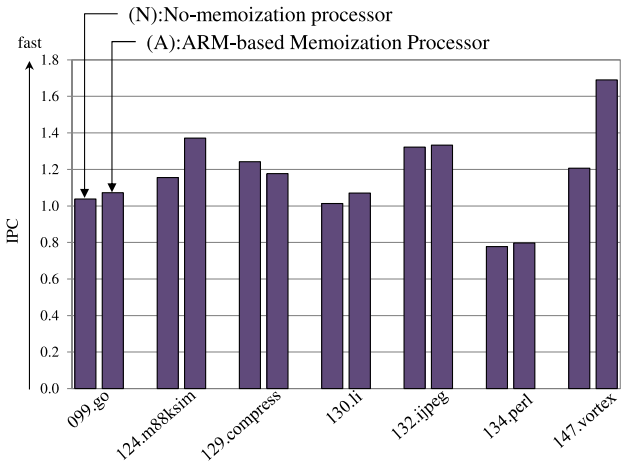



Fig. 9. IPC (SPEC CPU95 INT).

in the pipeline. On the other hand, when the processor succeeds in the reuse test, the instructions in the pipeline must be flushed. At this time, if the processor has executed instructions from the return destination while trying reuse test, the instructions in the pipeline need not to be flushed. Therefore, we consider a prediction mechanism for reuse test. The prediction mechanism is similar to a mechanism of a branch predictor. When the processor predicts that reuse test will succeed with this mechanism, the processor executes instructions from the return destination while trying the reuse test. Incidentally, this prediction is tried in the decode stage because an instruction for a function call can be detected first in the stage. Then, in the case the processor succeeds in the reuse test actually, the processor can continue to execute the following instructions without flushing instructions in the pipeline. Therefore, if the prediction for reuse test succeeds, the processor can restrain causing pipeline bubbles.

Finally, the IPC of (A) are shown in Fig.9. Each IPC is represented by two bars. The left-bar plots the IPC of (N) and the right-bar plots the IPC of (A). Now, the IPC is generally used for evaluating the performance of superscalar processors. However, if we calculate the IPC of the (A) on the basis of counts of the instructions which are executed actually, we can not evaluate speed-up correctly. This is because (A) reduces the executed instruction counts by omitting function executions and outputs the same execution results as the processor without the auto-memoization mechanism. Therefore, we have calculated the IPC on the basis of counts of the instructions which should be executed primarily, including the omitted instructions. With many programs such as 124.m88ksim and 147.vortex, the IPC of (A) is higher than (N). However, with 129.compress, the IPC of (A) is lower than (N). This is because in 129.compress, there are few functions to which computation reuse can be applied and only the search overhead increases as we can see in TABLE II and Fig.8. Therefore, in future, we will have to implement the mechanism for avoiding memoizing unsuitable computation blocks. This mechanism which is called *overhead filter* have been already implemented on the SPARC-based Auto-Memoization Processor [2], and we know that the performance degradation by increasing the search overhead can be restrained with this mechanism.

In conclusion, the performance of the ARM-based Auto-Memoization Processor(A) is better than the no-memoization processor (N) as a whole. The result shows that the performance of the ARM-based Auto-Memoization Processor can also increase as well as the SPARC-based Auto-Memoization Processor.

The hardware implementation cost for the auto-memoization mechanism is not small, and the power consumption will also increase. However, energy consumption will not increase so much because the execution time is reduced. Cache misses and cache accesses are also reduced by eliminating some function execution, and this can restrain the increase of the energy consumption. We intend to examine the hardware cost and the energy consumption of ARM-based Auto-Memoization Processor in detail.

## VII. Conclusions

In this paper, we described the implementation and the evaluation result for the ARM-based superscalar Auto-Memoization Processor. Through the implementation, we can acquire some knowledge about implementing the auto-memoization mechanism on an ARM-based superscalar processor. In addition, the evaluation result with SPEC CPU95 benchmark suite shows that the ARM-based Auto-Memoization Processor can also achieve speed-up as well as the SPARC-based Auto-Memoization Processor.

One of our future works is to implement a method for restraining the performance degradation which is caused by the pipeline bubbles. Other future works are to implement the mechanism for memoizing loop iterations and the mechanism for avoiding memoizing unsuitable computation blocks.

## References

[1] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima, "Design and evaluation of an auto-memoization processor," in *Proc. Parallel and Distributed Computing and Networks*, Feb. 2007, pp. 245–250.

[2] K. Kamimura, R. Oda, T. Yamada, T. Tsumura, H. Matsuo, and Y. Nakashima, "A speed-up technique for an auto-memoization processor by reusing partial results of instruction regions," in *Proc. 3rd Int'l. Conf. on Networking and Computing (ICNC'12)*, Dec. 2012, pp. 49–57.

[3] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proc. 24th Annual Int'l Symp. on Computer Architecture (ISCA-24)*, Jun. 1997, pp. 194–205.

[4] A. González, J. Tubella, and C. Molina, "Trace-level reuse," in *Proc. Int'l Conf. on Parallel Processing*, Sep. 1999, pp. 30–37.

[5] A. T. Costa, F. M. G. França, and E. M. C. Filho, "The dynamic trace memorization reuse technique," in *PACT*, 2000, pp. 92–99.

[6] J. Yang and R. Gupta, "Load redundancy removal through instruction reuse," in *Int'l Conf. on Parallel Processing*, Aug. 2000, pp. 61–68.

[7] S. Önder and R. Gupta, "Load and store reuse using register file contents," in *ICS'01*, 2001, pp. 289–302.

[8] P. Norvig, *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1992.

[9] *UltraSPARC III Cu User's Manual*, Sun Microsystems, May 2002.

[10] ARM Limited, *"ARM Architecture Reference Manual"ARM DDI 0100E*, 2000.

[11] AMD Corporation, *AMD Opteron A1100 Processor*, 2014.

[12] MOSAID Technologies Inc., *Feature Sheet: MOSAID Class-IC DC18288*, 1st ed., Feb. 2003.