Hinting for Auto-Memoization Processor based on Static Binary Analysis

Takanori TSUMURA[*], Yuuki SHIBATA[*], Kazutaka KAMIMURA[*‡],
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
[‡]Currently with DENSO CORPORATION, 1-1, Showa-cho, Kariya, Aichi, Japan

***Abstract*—We have proposed a processor called Auto-Memoization Processor which is based on computation reuse, and merged it with speculative multi-threading based on value prediction into a mechanism called Parallel Speculative Execution. The processor dynamically detects functions and loop iterations as reusable blocks, and registers their inputs and outputs into the table called Reuse Table automatically. Then, when the processor detects the same block, to apply computation reuse to the block, the processor compares the current input sequence with the previous input sequences registered in Reuse Table. In this paper, we propose a hinting technique for Auto-Memoization Processor based on static binary analysis. The hint indicates two distinctive types of input for loop bodies. One input type is unchanging value. When applying computation reuse to a loop, the processor can skip comparing such unchanging inputs with the values on Reuse Table. The other input type is unmonotonous changing value. The loops which have unmonotonous changing inputs will not benefit from computation reuse, and the processor can stop applying useless computation reuse to such loop iterations. By hinting these types of input to the processor, the overhead of Auto-Memoization Processor can be reduced. The result of the experiment with SPEC CPU95 benchmark suite shows that the hinting technique improves the maximum speedup from 40.6% to 51.8%, and the average speedup from 11.9% to 16.5%.**

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. Recently, multi-core processors equipped with two or more cores attract a great deal of attention. They are now in wide use from generic processors for PCs to embedded processors[1].

Generally, a program forms a poset, or a lattice. It has a length along time axis, and has a width (i.e. parallelism) orthogonal to time axis. Traditional speed-up techniques mentioned above are all based on some parallelisms in different granularities. In other words, their approaches aim to increase performance by shrinking the width of the program lattice.

On the other hand, we have proposed a processor called Auto-Memoization Processor based on computation reuse[2]. In contrast to traditional speed-up techniques for microprocessors, memoization, or computation reuse, tries to shrink the length of the program lattice. As a speedup 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.

So far, we have proposed some speed-up techniques for an Auto-Memoization Processor[3]. All of them are based on the characteristics of reusable blocks which are analyzed dynamically along the program execution. However, dynamic analysis has a problem. Specifically, only after an execution of a reusable block has been completed, it can be analyzed and its characteristic can be used for speedup.

In this paper, to reduce the overhead of Auto-Memoization Processor, we propose a hinting technique based on static binary analysis. The static analysis can extract the characteristics considering all instructions in a program before program execution is started. Therefore, the hints generated by static binary analysis are available just after the program execution is started.

II. Research Background

In this section, we describe Auto-Memoization Processor and Parallel Speculative Execution 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**[4] to apply computation reuse to computation blocks in programs.

Memoization is originally a programming technique for speed-up, and brings good results on expensive functions[5]. However, it requires rewrite of target programs, and the traditional load-modules or binaries can not benefit from memoization. Furthermore, the effectiveness of memoization is influenced much by programming styles. Rewriting programs using memoization occasionally makes the programs slower. This is because software implemented memoization costs considerable overheads.

On the other hand, **Auto-Memoization Processor**[2], which we have proposed, can execute traditional load-modules faster with low overheads and without rewriting binary program. Auto-Memoization Processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically. A region between an instruction with a callee label and `return` instruction is detected as a memoizable function. A region between a backward branch instruction and its branch target is detected as a memoizable loop iteration.

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 arguments are also included in the input sequence, and 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.1. MemoTbl consists of four tables:

FLTbl: for start addresses of instruction regions.

InTbl: for input value sets of instruction regions.

AddrTbl: for input address sets of instruction regions.

OutTbl: for output data sets of instruction regions.

FLTbl, AddrTbl, and OutTbl are implemented with RAM. On the other hand, InTbl is implemented with a ternary CAM

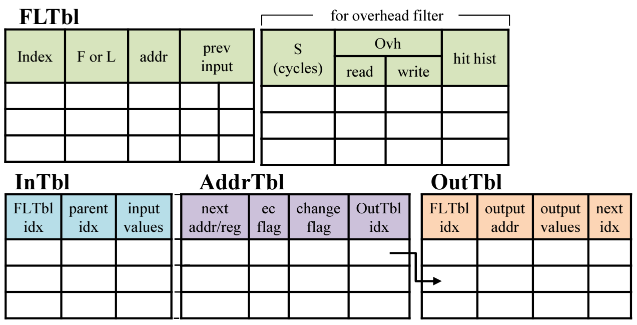

Fig. 1. Structure of MemoTbl.

(Content Addressable Memory), so that input values can be found fast by associative search.

Each FLTbl line corresponds to a reusable computation block. One FLTbl entry has two groups of fields, the one is for computation reuse and the other is for the overhead filter which will be explained later in section II-C. The fields for computation reuse hold whether the block is a function or a loop (*ForL*) and the start address of the block (*addr*). Each FLTbl entry holds previous two input sequences for predicting next input sequences (*prev. inputs*) for Parallel Speculative Execution which will be explained later in section II-B. The fields for the overhead filter hold the execution cycles (*S*), the past reuse overhead (*Ovh*) and the reuse frequency (*hit hist*).

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 holds an index key for its parent entry (*key*). As input values are stored by single cache line, computation reuse is not applied to reusable blocks whose inputs are on non-cacheable memories. 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 or a register ID which should be tested next (*next adr/reg*). In addition, each AddrTbl entry has two flags. One shows whether its entry is the terminal entry of an input sequence, or not (*ec flag*). The other shows whether the value stored in the address or the register indicated by *next adr/reg* is overwritten after being registered into AddrTbl, or not (*change flag*).

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.

When Auto-Memoization Processor tries reuse test, the processor searches FLTbl for the entry whose *Index* corresponds to the current computation block. Next, the processor searches InTbl for the entry whose *FLTbl idx* corresponds with *Index* and whose *input values* match the current input values. Then, the processor refers to the values stored in the address or the register indicated by *next addr/reg* of the AddrTbl

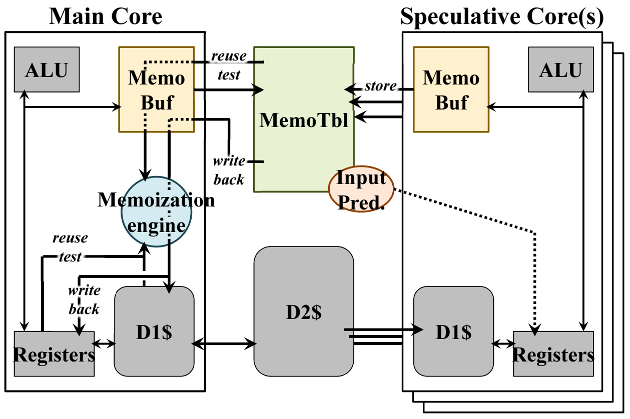

Fig. 2. Structure of Parallel Speculative Execution.

entry corresponding to the found InTbl entry, and compares them with the current input values. This process is applied repeatedly until the processor finds the entry whose *ec flag* is set. If the values stored in the address or the register indicated by *next addr/reg* is never overwritten after being registered into InTbl, the processor need not to compare the current input with the corresponding past input. Hence, Auto-Memoization Processor skips such useless comparison if *change flag* of an entry in AddrTbl is unset.

B. Parallel Speculative Execution

As a matter of course, memoization can omit the execution of an instruction region only if the current input values for the region match completely with the input values which are used in a former execution. Hence, memoization can not be applied to loop iterations because some input values of a loop, such as its iterator variable, will monotonously change in general. To apply computation reuse for loop iterations, we installed some speculative multi-threading cores called **SpC** (speculative cores) to Auto-Memoization Processor. Speculative execution with these cores helps to apply computation reuse to loop iterations. Fig.2 shows the structure of Auto-Memoization Processor with three SpCs.

Each SpC has its own MemoBuf and a first level data cache. The second level data cache and MemoTbl are shared between all cores. While the main core executes a memoizable instruction region, SpCs execute the same region using predicted inputs, and store the results into shared MemoTbl. SpCs preferentially execute the region whose reuse frequency stored in *hit hist* is high. This is because that when the results of speculative executions for an instruction region are frequently reused, the instruction region is supposed to be suitable for Parallel Speculative Execution. The inputs are predicted by stride prediction using the last two input sets stored in *prev. inputs* field of FLTbl. If the input prediction was correct, the main core can omit intended execution by reusing the result by one of SpCs. Unlike as other SpMT techniques, even if the input speculation proves to be incorrect later, the main core need not to pay a cost for any back-out management. It only fails reuse test and executes the region as usual. This extension can omit the execution of instruction regions whose inputs show monotonous change.

C. Overhead Filter

For some reusable regions, reuse overhead may outweigh the eliminated execution cycles by reuse. This will go for some regions which have many input values to be tested, and all tiny regions. Hence, Auto-Memoization Processor should estimate the effect of reuse, and avoid memoizing unsuitable instruction regions. This can reduce useless reuse test and will contribute to good performance.

Assume that M represents the number of successful reuses of a certain region for recent T times tries ($0 \leq M \leq T$). The value of M can be retrieved from *hit hist.* field in FLTbl, because *hit hist.* is implemented with a shift register and holds the history of success or failure of the last T reuse tests. Here, the value of T is the width of the register. With the execution cycles S of the region, which can be also retrieved from FLTbl, the reduced cycles can be represented as follows;

$$M \cdot (S - Ovh^R - Ovh^W) \qquad (1)$$

where Ovh^R and Ovh^W represent search/writeback overheads for the region respectively.

Ovh^R also costs when reuse test fails. This overhead can be calculated as follows.

$$(T - M) \cdot Ovh^R \qquad (2)$$

Here, if the *overhead* (2) is larger than the *expected eliminated cycles* (1), the computation region cannot be suitable for reuse. Now, we define the difference between (1) and (2) as $Gain$ (3). The additional small logic calculates whether $Gain$ goes positive or negative, and decides the suitability of computation regions.

$$Gain = M \cdot (S - Ovh^W) - T \cdot Ovh^R \qquad (3)$$

III. Hinting with Static Binary Analysis

In this section, we propose a hinting technique for Auto-Memoization Processor, and static analysis for generating the hints.

A. Extracting Characteristics of How Input Values Change

In general, how an input value of a loop changes across iterations may have a certain characteristic. This characteristic can affect the performance of computation reuse for loops. For example, as described in section II-B, Auto-Memoization Processor uses the characteristic that some input values of a loop will change monotonously. Considering the characteristic, the processor predicts input values for following loop iterations, and omits the executions of the loop iterations with applying Parallel Speculative Execution. In addition to such monotonous change, an input of a loop can have some characteristics; for example, an input value may not change across iterations, and another input value may change erratically. Nevertheless, Parallel Speculative Execution mechanism predicts that all input values of a loop will change monotonously, and tries to apply computation reuse to the loop. Therefore, the processor tries reuse test for the loop which has unpredictable input values. In such a case, reuse test hardly ever succeeds, and search overhead for reuse test continues to increase in vain. This overhead is one of the factors of performance degradation of Auto-Memoization Processor. Hence, by extracting characteristics of loop inputs with binary analysis, we aim to apply computation reuse for loops efficiently by using the characteristics of how input values change. In the following, we call the input whose value will not change across iterations '**unchanging input**' and the input whose value changes erratically '**unmonotonous input**.'

Now, the characteristics, which can be extracted, of a loop input will vary depending on which execution path is selected by the results of branch instructions. Dynamic analysis can only extract the characteristics along a certain selected execution path, and such characteristics can not be used on other execution paths. On the other hand, static analysis can extract the characteristics considering all instructions in all execution paths of a loop. In this paper, we propose a technique for detecting loops which have unchanging inputs or unmonotonous inputs with static binary analysis, and a technique for hinting the analysis results to Auto-Memoization Processor.

B. *Unchanging Input Values across Iterations*

As mentioned in section II-A, Auto-Memoization Processor skips a comparison of input values when the values stored in the address which should be tested next are not overwritten after being registered into AddrTbl. However, the processor does not skip the comparison of input values stored in registers. This is because the values stored in the registers are overwritten more frequently than the values stored in memories, and the overhead for setting *change flag* will largely increase. However, some loops have unchanging inputs stored in the registers. Now, Fig.3 shows an example code which includes such a loop, and loop1 in the code is the loop which has an unchanging input. Here, SPARC which is the base architecture of Auto-Memoization Processor has a branch delay slot. Therefore, the iteration region of loop1 is between the instruction at address 0x20000 and 0x20010 the delay slot of the branch instruction at address 0x2000c. When loop1 is analyzed, it is found that %o3 is an input because it is used as a source for the add instruction at 0x20010, and it is also found that %o3 is not written along the execution of loop1. Therefore, %o3 proves to be an unchanging input. We analyzed the benchmark programs in SPEC CPU95 and found that each loop in the programs has two unchanging inputs on average. Hence, we propose a static analysis for detecting unchanging register inputs. Such unchanging register inputs being annotated in binaries, Auto-Memoization Processor can omit comparing the unchanging inputs and reduce the overhead of reuse test for loop iterations. In addition, more loops can be decided as suitable for memoization, and more

```
00020000<loop1>:
  20000 add %o1, 5, %o1
  20004 inc %l2
  20008 cmp %l2, 10
  2000c bne 20000 <loop1>
  20010 add %o1, %o3 ,%o1
     :
00030000<loop2>:
  30000 add %o1, %l2, %o1
  30004 smul %l2, %l2, %o3
  30008 inc %l2
  3000c cmp %l2, 10
  30010 bne 30000 <loop2>
  30014 add %o1, %o3 ,%o1
```

Fig. 3. Assembly code of loops.

execution cycle can be reduced. This is because that the reuse test overhead for loops is reduced. In other words, the right side of eq. (3) becomes smaller and the value of $Gain$ will be positive in more cases.

C. *Erratically Changing Input Values*

As mentioned in section II-B, if input prediction was correct, the main core can omit intended execution by reusing the result by one of SpCs. However, the main core cannot omit the execution of loops which has unmonotonous inputs because input prediction fails. Here, "unmonotonous" means "changing erratically." Now, loop2 in Fig.3 is an example of the loop which has an unmonotonous input. Here, %o1 is one of the input of this loop, and the values in %l2 and %o3 are added to the value in %o1 every loop iteration. Besides, these values in %l2 and %o3 change every loop iteration. Hence, %o1 is assumed to be an unmonotonous input because the change amount of the values in %o1 is different every loop iteration. Therefore, the stride prediction for %o1 will fail. Hence, we propose a static analysis for finding such unmonotonous inputs. Loops which have unmonotonous inputs being annotated in binaries, Auto-Memoization Processor can avoid applying computation reuse to such loops, and omit useless reuse test.

IV. Static Binary Analysis and Binary Annotation

In this section, we describe how to analyze a binary program in order to detect the two types of inputs described in section III, and we also describe how Auto-Memoization Processor utilizes the analysis results.

A. *How to Analyze a Binary Program*

To apply computation reuse for loops efficiently, loops which have unchanging inputs or unmonotonous inputs need to be detected with static binary analysis. First of all, to detect such loops, all loop blocks in a binary program should be found. As mentioned in section II-A, Auto-Memoization Processor detects a region between a backward branch instruction and its branch target as a loop block. Therefore, backward branch instructions need to be found by analyzing a binary program first. A backward branch instruction is a branch instruction whose target address is lower than the own address of the branch instruction. Hence, by investigating the target indicated in the operand of each branch instruction, the branch proves to be whether a backward branch or not. If a backward branch instruction is detected, its branch target address is defined as the start address of the loop and the address of the instruction following the backward branch instruction (i.e. the branch delay slot) is defined as the end address of the loop. In this way, all loops in the binary program are detected.

After all loops are detected, all instructions in the binary program are analyzed again to search the inputs and outputs of each loop. Now, for picking up unchanging register inputs, the registers whose values are read but not written in a loop need to be found. On the other hand, for picking up unmonotonous register inputs, the destination registers of instructions whose source values are overwritten in a loop need to be found. However, it is difficult to find instructions whose source values are overwritten in a loop. Therefore, in this paper, we regard a value on a register, which is a source of an instruction and also is the destination of multiple arithmetic instructions, as an unmonotonous register input.

On SPARC-V8 processors, eight global registers (GRs), eight output registers (ORs), eight local registers (LRs), and eight input registers (IRs) can be used in a function, and 32 floating-point registers can be also used. Therefore, the inputs and outputs in 64 registers should be analyzed for each loop.

Now, we describe an example of the analysis flow by using the assembly code shown in Fig.3. First, to detect backward branch instructions, the binary program is analyzed, and the instruction at address 0x2000c and the instruction at address 0x30010 are detected as backward branch instructions. Therefore, it is found that loop1 is from the instruction at address 0x20000 to the instruction at address 0x20010 and loop2 is from 0x30000 to 0x30014. Next, to extract characteristics of inputs of the loops, the binary program is analyzed again. Then, after loop1 is detected at address 0x20000, each register used in loop1 is investigated, and it is found that the operation code of the instruction at address 0x20000 is 'add.' Subsequently, the source and the destination register are checked, and it is found that %o1 is an input and an output of loop1, because %o1 is used as a source and the destination of add instruction. Similarly, the operand registers of the instruction at address 0x20004 are checked, and it is found that %l2 is also an input and an output of loop1. On the other hand, %o3 is an input of loop1, because %o3 is indicated as a source of the instruction at address 0x20010. After the operand registers of the instruction at address 0x20010 is checked, the analysis about loop1 finishes. As a result of the analysis of loop1, it is found that %o3 is an unchanging register input. Next, after loop2 is detected at address 0x30000, each register used in loop2 is investigated, and it is found that %o1 is an input and an output of loop2. In addition, as %l2 is also an input, it is found that %o1 is the destination register of an instruction whose source operands are registers.

```
     :
  1fffc  reuse_unchange %o3
00020000<loop1>:
  20000  add %o1, 5, %o1
  20004  inc %l2
  20008  cmp %l2, 10
  2000c  bne 20000 <loop1>
  20010  add %o1, %o3 ,%o1
     :
  2fffc  reuse_complex
00030000<loop2>:
  30000  add %o1, %l2, %o1
  30004  smul %l2, %l2, %o3
  30008  inc %l2
  3000c  cmp %l2, 10
  30010  bne 30000 <loop2>
  30014  add %o1, %o3 ,%o1
```

Fig. 4. An example for inserting dedicated instructions.

Similarly, the operand registers of the instruction at address 0x30014 are checked, and it is found that %o1 is indicated as a source and the destination of the instruction and %o3 is indicated as a source of the instruction. Therefore, it is found that %o1 is the destination register of two instructions whose source operands are registers. After the operand registers of the instruction at address 0x30014 are checked, the analysis about loop2 finishes. As a result of the analysis of loop2, %o1 is regarded as an unmonotonous register input.

B. How to Utilize Static Analysis Results

Characteristics of input values of each loop in a binary program can be extracted as the result of static analysis. A hinting technique, which is based on results of compiler's flow-analysis, for computation reuse was proposed [6],[7]. In this technique, to utilize the analysis results, some instructions are annotated. Similarly, to utilize the analysis results on Auto-Memoization Processor, loops should be annotated in a binary program. Specifically, we define some dedicated instructions, and annotates loops which have unchanging inputs or unmonotonous inputs by inserting the dedicated instructions just before the loops. We extend SPARC ISA by adding two dedicated instruction *reuse_unchange* and *reuse_complex*. The *reuse_unchange* instruction indicates that a loop following the *reuse_unchange* instruction has unchanging register inputs. In addition, unchanging register inputs are indicated in operand fields of the *reuse_unchange* instruction. In mnemonic form, registers written behind the *reuse_unchange* instruction indicate unchanging register inputs. When a loop is found just after Auto-Memoization Processor detects a *reuse_unchange* instruction, the processor skips the comparison for the registers indicated in the operands of the *reuse_unchange* instruction. On the other hand, the *reuse_complex* instruction indicates that a loop following the *reuse_complex* instruction has unmonotonous register inputs. When the processor detects a *reuse_complex* instruction, the processor does not try reuse

TABLE I
SIMULATION PARAMETERS.

MemoBuf	64 KBytes
MemoTbl CAM	128 KBytes
MemoTbl small CAM	8 KBytes
Comparison (register and CAM)	9 cycles/32 Bytes
Comparison (Cache and CAM)	10 cycles/32 Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32 Bytes
D1 cache	32 KBytes
line size	32 Bytes
ways	4 ways
latency	2 cycles
miss penalty	10 cycles
D2 cache	2 MBytes
line size	32 Bytes
ways	4 ways
latency	10 cycles
miss penalty	100 cycles
Register windows	4 sets
miss penalty	20 cycles/set

test for the loop following the *reuse_complex* instruction. In the current implementation, these instructions are inserted in not a binary code but an assembly code. Inserting into a binary code is one of our future works. Now, Fig.4 shows an example for inserting the dedicated instructions into the assembly code shown in Fig.3. Here, a *reuse_unchange* instruction is inserted just before the start address of loop1 for indicating loop1 has the unchanging register %o3. This *reuse_unchange* instruction is executed before the execution of loop1, and it can be found that %o3 does not change in loop1. Therefore, when the processor tries reuse test for loop1, the processor skips the comparison of %o3 for reducing the search overheads. On the other hand, the *reuse_complex* instruction is inserted just before the start address of loop2 for indicating that loop2 has some unmonotonous register inputs. Therefore, when this *reuse_complex* instruction is executed before the execution of loop2, the processor does not try reuse test for loop2 for reducing the search overheads which are caused in the case Auto-Memoization Processor fails reuse test.

V. PERFORMANCE EVALUATION

We have implemented the above-mentioned software and hardware extension on a processor simulator with auto-memoization structures. This section describes the evaluation results with the simulator.

A. Simulation Environment

We have developed a single-issue SPARC-V8 simulator equipped with the auto-memoization mechanisms and three SpCs. In this section, we will discuss the practicality of the proposed hinting technique in this paper. The simulation parameters are shown in TABLE I. The cache structure and the instruction latencies are based on SPARC64-III[8]. The on-chip CAM for InTbl in MemoTbl is modeled on DC18288[9] (32Bytes $\times$ 4K lines). The latencies of the CAM are defined on the assumption that the clock frequency of the processor is about 2 GHz, and is 10-times higher than the CAM.

B. Execution Cycles with SPEC CPU95

We have evaluated the proposed hinting technique by executing all benchmark programs in SPEC CPU95 suites with 'train' dataset. All benchmark programs are compiled by gcc version 3.0.2 with '-msupersparc -O2' option, and linked statically. We have evaluated following four processors;

(N) No-memoization processor. (baseline)
(M) Previous Auto-Memoization Processor.
(U) Auto-Memoization Processor with the hinting technique for skipping the comparison of unchanging register inputs.
(C) Auto-Memoization Processor based on (U) with the hinting technique for stopping computation reuse for the loop which has an unmonotonous input.

and Fig.5 shows the execution cycles of these processors. Each bar is normalized to the number of executed cycles of (N). The legend in Fig.5 shows the breakdown items of total cycles. They represent the executed instruction cycles ('**exec**'), the comparison overhead between CAM and the registers or the caches ('**read**'), the writeback overhead ('**write**'), the first-level and shared second-level data cache miss penalties ('**D$1**', '**D$2**'), and the register window miss penalty ('**window**') respectively.

First, with many benchmark programs, the performance of (U) Auto-Memoization Processor with the hinting technique is higher than (M) previous Auto-Memoization Processor. Especially, with 124.m88ksim, 134.perl, 107.mgrid and 125.turb3d, 'read' is reduced. This means that the search overhead is reduced by skipping a comparison of unchanging register inputs. On the other hand, with some programs such as 101.tomcatv and 102.swim, 'exec' is reduced. This means that Auto-Memoization Processor omits the execution of some loop iterations. This is because search overhead of (U) becomes smaller than that of (M), and Auto-Memoization Processor can apply computation reuse for some loops which are decided as unsuitable for memoization on (M).

However, with 130.li, the performance of (U) is lower than that of (M) because 'read' is increased. This means that some loops in 130.li can be also decided as suitable for memoization, but reuse test for them cannot succeed.

Secondly, the performance of (C) Auto-Memoization Processor with hinting technique is higher than that of (U) with 129.compress, 134.perl, 107.mgrid, 125.turb3d, and 141.apsi. Especially, 'read' of 129.compress is reduced without increasing 'exec.' This means that the search overhead is reduced by stopping computation reuse for the loops which have unmonotonous inputs. On the other hand, with many benchmark programs such as 124.m88ksim, 101.tomcatv and 102.swim, the performance of (C) provides almost the same performance as (U). This means that most of the loops which have unmonotonous inputs can not be detected because the criteria for the unmonotonous inputs are inaccurate. Meanwhile, with 104.hydro2d, the performance of (C) is lower than that of (U) because 'exec' is increased. This means that computation reuse for some loops can be applied on (U), but computation reuse

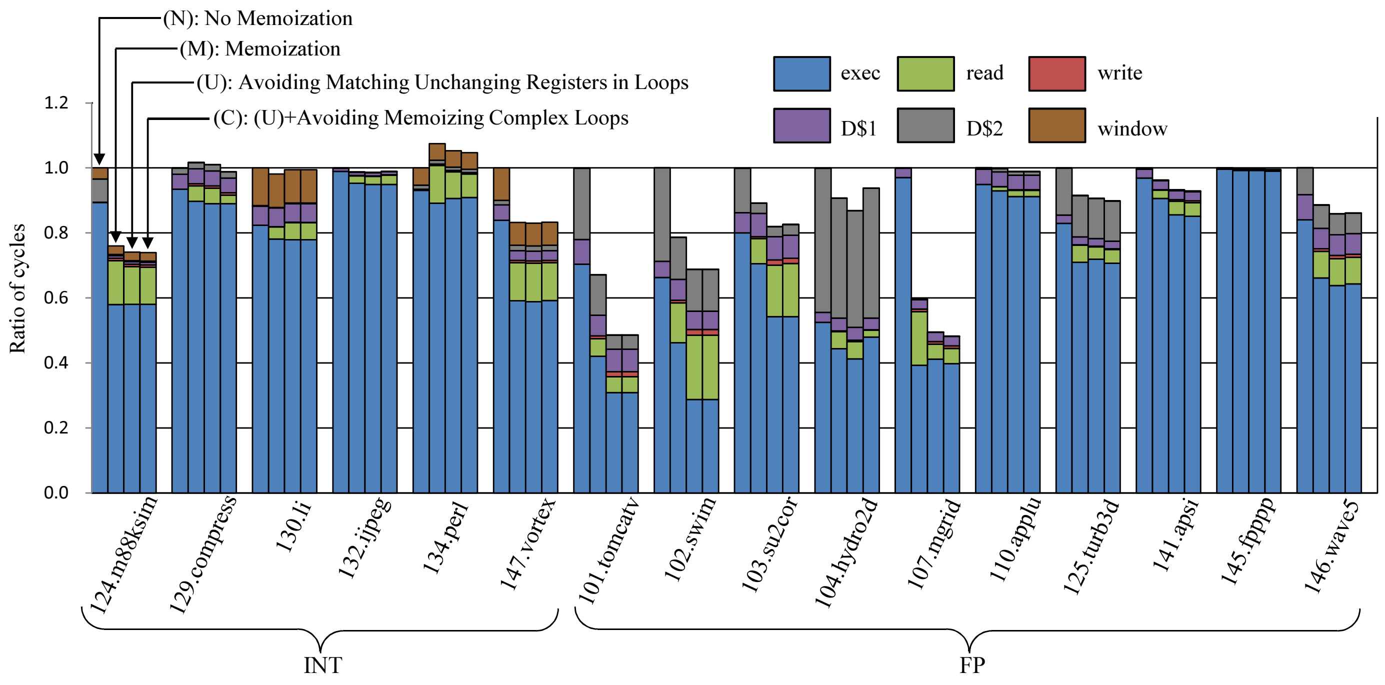

Fig. 5. Evaluation(SPEC CPU95).

for the loops cannot be applied on (C). This reason is that the characteristics, which are used on the proposed hinting technique, of loop inputs are extracted from all instructions in loops. Therefore, if only a part of instructions in loops are executed because of branch instructions, the extracted characteristics may be incorrect. In this case, the stride prediction for the inputs which are detected as unmonotonous inputs may succeed. This means that 104.hydro2d has the loop which does not have unmonotonous inputs and can be applied computation reuse. However, on (C), computation reuse is not applied to the loop. Therefore, the performance of (C) is lower than that of (U).

In conclusion, the performance of Auto-Memoization Processor with the hinting technique is better than the previous Auto-Memoization Processor as a whole. The previous Auto-Memoization Processor (M) reduces the cycle by 40.6% in maximum and by 11.9% in average while Auto-Memoization Processor with the hinting technique (U) reduces the cycles by 51.4% in maximum and by 16.7% in average, and Auto-Memoization Processor with the hinting technique (C) reduces the cycle by 51.8% in maximum and by 16.5% in average. Comparing (C) with (M), (C) reduces the cycle by 27.6% in maximum and by 5.3% in average.

VI. Conclusions

In this paper, we proposed a hinting technique for Auto-Memoization Processor based on static binary analysis. First, we described a static analysis for detecting unchanging input values and erratically changing input values in loops. We also described a hinting technique, which is based on the static analysis results, for reducing reuse overheads. By annotating unchanging inputs, useless reuse test was reduced, and by annotating loops which have erratically changing inputs, useless computation reuse was restrained. Through an evaluation with SPEC CPU95 benchmark suite programs, it is found that Auto-Memoization Processor with the hinting technique improves the maximum cycle reduction ratio from 40.6% to 51.8%, and the average ratio from 11.9% to 16.5%. One of our future works is to improve a static analysis for detecting loop inputs which change erratically. This enables Auto-Memoization Processor to avoid increasing the search overheads which are caused in the case the processor fails reuse test for such loops.

References

[1] ARM Ltd, *The ARM Cortex-A9 Processors*, Sep 2007.
[2] T. Tsumura *et al.*, “Design and evaluation of an auto-memoization processor,” in *Proc. Parallel and Distributed Computing and Networks*, Feb. 2007, pp. 245–250.
[3] K. Kamimura *et al.*, “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.
[4] P. Norvig, *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1992.
[5] J. Huang and D. J. Lilja, “Exploiting basic block value locality with block reuse,” in *Proc. 5th Int’l Symp. on High-Performance Computer Architecture (HPCA-5)*, Jan. 1999, pp. 106–114.
[6] ——, “Exploring sub-block value reuse for superscalar processors,” in *Proc. 2000 Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT’00)*, 2000, pp. 100–107.
[7] D. A. Connors *et al.*, “Hardware support for dynamic activation of compiler-directed computation reuse,” in *Proc. 9th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, 2000, pp. 222–233.
[8] *SPARC64-III User’s Guide*, HAL Computer Systems/Fujitsu, May 1998.
[9] MOSAID Technologies Inc., *Feature Sheet: MOSAID Class-IC DC18288*, 1st ed., Feb. 2003.