This paper is author's private version of the paper published as follows:
Proc. 1st Int'l Symp. on Computing and Networking (*CANDAR'13*), pp.378-374

CAM Size Reduction Method for Auto-Memoization Processor by considering Characteristics of Loops

Yuuki SHIBATA*, Kazutaka KAMIMURA*, Tomoaki TSUMURA*,
Hiroshi MATSUO* 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, and merged it with speculative multi-threading based on value prediction into a mechanism called Parallel Speculative Execution. The processor has table called Reuse Table for registering inputs and outputs of computation blocks. The table is implemented by a ternary CAM, and input sequences are stored onto the table. The past model cannot efficiently use Reuse Table entries because the purging algorithm for Reuse Table is too simple. This paper proposes a new algorithm for purging useless entries based on characteristics of loops. We also propose a new storing and searching filter against useless entries for loops. The evaluation result with SPEC CPU95 benchmark suite shows that these mechanisms allow Reuse Table to be implemented with a smaller CAM without any performance degradation.**

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

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]. The processor has table called Reuse Table for registering inputs of computation blocks. The table is implemented by a ternary CAM, and input sequences are stored onto the table. 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.

We have also proposed a mechanism called Parallel Speculative Execution. It predicts the inputs for a reusable loop iteration, and additional shadow cores execute the iteration speculatively. The shadow cores store the results of the speculative executions onto Reuse Table. If the value prediction for inputs is correct, the stored outputs can be reused by the main core and execution time will be reduced.

However, some input values of a loop, such as an iterator variable of the loop, will generally change monotonously. Therefore, Reuse Table entries for an loop iteration executed by the main core will not be reused in future, but the traditional purging algorithm for Reuse Table does not distinguish such entries and other entries. Hence, the past model can not efficiently use Reuse Table.

In this paper, we propose a new algorithm for purging useless entries based on characteristics of loops. We also propose a new storing and searching filter against useless entries for loops. These mechanisms allow Reuse Table to be implemented with a smaller CAM without any performance degradation, and the energy consumption of Auto-Memoization Processor can be reduced.

II. Related Work

Studies for extracting ILPs with speculative executions based on value prediction have been proposed by Lipasti et al.[3] and Wang et al.[4] Many speculative multi-threading (SpMT) models also have been proposed. They have multiple processors or cores, and run threads speculatively using predicted value sets. In an SpMT model, a speculative thread will generally squashed when its input values are overwritten by the main thread.

Roth et al.[5] have proposed *register integration*. It is a mechanism for reusing the results of squashed instructions by

writing back the past register mapping. It is shown that the model can provide performance improvements of up to 11.5%.

Some hybrid methods of computation reuse and value prediction have been also studied. Wu et al.[6] have proposed a speculative multi-threading supported by computation reuse. In the model, the compiler identifies instruction regions for reuse or value prediction. At runtime, if a region cannot be reused, the processor predicts the outputs of the region, and speculatively executes its following instructions using the predicted values. Hence, if the value prediction fails, the speculative executions should be squashed, and it costs additional hardware and overhead for the squash.

Molina et al.[7][8] have proposed a combination model of speculative thread and non-speculative thread. The execution results of speculative thread are stored into a FIFO , and non-speculative thread picks up instructions from the FIFO. If the current source operands and the stored operands are same, the non-speculative thread reuses the execution results and skips execution.

In contrast to these studies, Parallel Speculative Execution which we have proposed is a non-symmetric SpMT model based on value prediction, and uses computation reuse technique. Our model has two advantages over [6]. The one is that compiler-assist is not required for computation reuse. The other is that there is no need to squash speculative executions. Molina's model [7] is similar to our model. However, our model can reuse some instruction regions which require memory read as their inputs.

III. 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**[9] to apply computation reuse to computation blocks in programs.

Auto-Memoization Processor, 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.

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 past input sequences which are stored in MemoTbl. If the current input sequence matches one of the stored input sequences

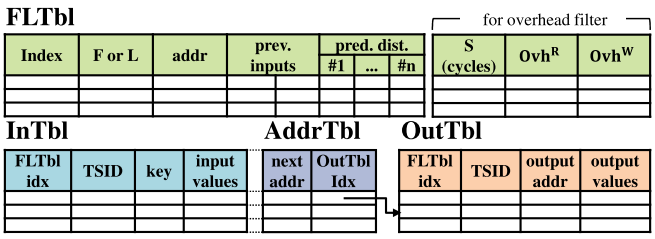

Fig. 1. Structure of MemoTbl.

on MemoTbl, the memoization engine writes back the stored outputs, which are associated with the 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.1. MemoTbl consists of four tables:

FLTbl: for start addresses of instruction regions.

InTbl: for input data sets of instruction regions.

AddrTbl: for input address sets of instruction regions.

OutTbl: for output data sets of instruction regions.

In the following, we call the three tables other than FLTbl 'IAO-tables.' In addition, we call each IAO-tables entry for loop 'loop entry' and each IAO-tables entry for function 'function entry.'

FLTbl, AddrTbl, and OutTbl are implemented with RAM. On the other hand, InTbl is implemented with a ternary CAM (Content Addressable Memory), so that inputs 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 III-C. The fields for computation reuse hold whether the block is a function or a loop (*F or L*) and the start address of the block (*addr*). Each FLTbl entry holds previous two input sequences for predicting next input sequences (*prev. inputs*) and core-IDs which are

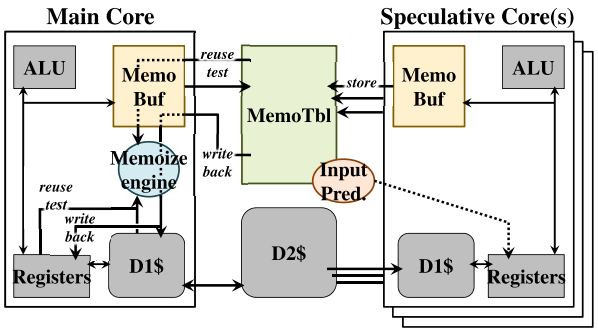

Fig. 2. Structure of Parallel Speculative Execution.

assigned to the execution of the loop iterations (*pred. dist.*) for Parallel Speculative Execution which will be explained later in section III-B. The fields for the overhead filter hold the execution cycles (*S*) and the past reuse overhead (*Ovh*).

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*).

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 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 is stored over multiple OutTbl entries.

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 former execution. Hence, memoization can not be applied to loops because some input values of a loop, such as an iterator variable, will monotonously change in general.

Now, we installed some speculative multi-threading cores called **SpC** (speculative cores) to Auto-Memoization Processor. These cores help the unsuitable regions for memoization mentioned above. 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. 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 methods, 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 blocks, reuse overhead may outweigh the eliminated execution cycles by reuse. This will go for some blocks which have many input values to be tested, and all tiny blocks. Hence, Auto-Memoization Processor has a structure which estimates the effect of reuse, and avoids memoizing unsuitable computation blocks. With the execution cycles S of the block, the processor calculates the performance gain in terms of omitted cycles as $S - Ovh^R - Ovh^W$ where Ovh^R and Ovh^W represent search/writeback overheads for the computation block respectively. If this value is negative, applying memoization will decrease the performance, and the processor stops reusing the block.

D. Purging Mechanisms for MemoTbl

Auto-Memoization Processor has two mechanisms for purging entries from MemoTbl. **TSID Purge** is a purging mechanism which selects victim entries based on LRU and purges the entries from IAO-tables. Auto-Memoization Processor has a ring counter for holding the current time stamp. When an entry is newly-stored or used for applying computation reuse, the current value of the ring counter is copied to *TSID* field of the entry. In addition, every time a certain number of new entries are stored into IAO-tables, the value of the ring counter is incremented and all entries which have the same time stamp as the updated ring counter value are purged from IAO-tables.

FLID Purge is the other mechanism for purging all entries which are associated with a selected computation block when FLTbl or IAO-tables overflows. In the case FLTbl overflows, the computation block which is estimated to have least benefits from computation reuse on the basis of overhead filter is selected as the victim block and all entries which are associated with the block are purged from MemoTbl. On the other hand, in the case IAO-tables overflows, the computation block currently being executed is selected as the victim and all entries which are associated with the block are purged from IAO-tables.

IV. Efficient Purging Algorithm Based on Characteristics of Loops

In this section, we will propose a new purging algorithm for Reuse Table based on characteristics of loops. We will also propose a new storing and searching filter against the loop entries which are not expected to be reused.

A. Loop Iteration Purge

In general, some input values of a loop, such as an iterator variable of the loop, will monotonously change. Therefore, a loop entry corresponding to an iteration executed by the main core should not be reused in future. In the following, we call the loop entries which are not expected to be reused

'useless loop entries.' Now, the traditional purging algorithm for Reuse Table does not distinguish useless loop entries and other entries. Thus, beneficial entries can be purged for storing useless loop entries. To solve this problem, we propose a new algorithm for purging useless loop entries from IAO-tables. In the following, we call this purging algorithm '**loop iteration purge**.'

B. Loop Iteration Storing Filter

As mentioned above, useless loop entries can be purged from IAO-tables by loop iteration purge. However, there are loop entries which will not be reused even once if they are stored into IAO-tables. Such entries are divided into following four types.

- (A) Loop entries which are stored by the main core.
- (B) Loop entries which are stored by an SpC after the main core starts executing the same iteration.
- (C) Loop entries which are stored by an SpC when the SpC start executing an outer loop in the nested loop and has executed the inner loop in the nested loop.
- (D) Loop entries which are stored by an SpC when the main core has already broken out the loop.

(A)-type entries are not expected to be reused even if the entries are kept in IAO-tables. This is because the iterator value of a loop generally changes monotonously and each iteration of loop is never executed with the same input sequence.

(B)-type entries are not expected to be reused with the same reason as for (A)-type entries. This is because the iteration which the main core has already started is never executed with the same input sequence.

(C)-type entries are not expected to be reused because the input values of the inner loop usually depend on the calculation in the outer loop, and the main core will usually fail to reuse the inner iteration when the main core fails to reuse the outer iteration.

(D)-type entries are not expected to be reused because the entries are associated with the loop iterations which exceed the limit of the loop count and can not be executed by the main core.

For these reasons, we propose a new mechanism for avoiding storing these four types of entries into IAO-tables. In the following, we call this mechanism '**loop iteration storing filter**.' Loop iteration storing filter and loop iteration purge enable MemoTbl to be efficiently used because useless loop entries will not be stored in IAO-tables.

C. Loop Iteration Searching Filter

Previous Auto-Memoization Processor searches MemoTbl even if the execution result for the iteration which will be executed next by the main core are not stored in IAO-tables. However, as mentioned above, a loop entry can be reused only if the entry is stored by an SpC before the main core starts executing the corresponding iteration. Hence, we propose a new mechanism for avoiding searching IAO-tables by managing which iterations have been executed by SpCs and which have not yet. In the following, we call this mechanism '**loop iteration searching filter**.' This mechanism enables to reduce search overhead, keeping hit rate of reuse for loops.

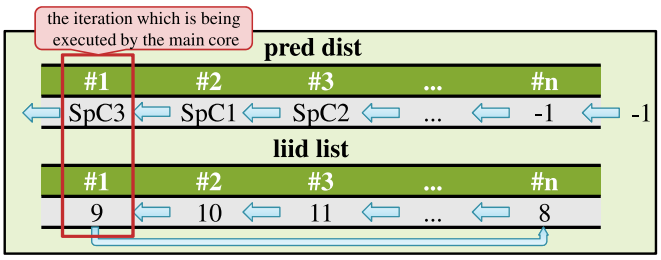

Fig. 3. Structures of *pred. dist.* and *liid list*.

V. Implementation

This section describes how we implement the three new mechanisms.

A. Hardware Extension for Loop Iteration Purge

We will discuss the hardware extension for loop iteration purge. To implement loop iteration purge, each loop iteration and the loop entry for the iteration must be associated with each other. Auto-Memoization Processor assigns some upcoming iterations to each SpC without overlap, and manages the mapping from core-IDs of SpCs to the iterations using *pred. dist.* When an SpC finishes a loop iteration, by using *pred. dist.*, the SpC can know the iteration is how much prior to the current iteration which the main core is now executing. We define an unique ID called 'iteration ID' for each iteration, and we have added a new set of fields called *liid list* in FLTbl for storing iteration IDs. Auto-Memoization Processor can distinguish an iteration from other iterations by examining the iteration IDs in *liid list*. In addition, we have also added the new field *liid* in InTbl and OutTbl for storing the iteration ID. By comparing the iteration ID of the iteration executed by the main core with iteration IDs of the loop entries stored in IAO-tables, the main core can find the loop entries which should be purged from IAO-tables.

Fig.3 shows the structures of *pred. dist.* and *liid list*. In this figure, the leftmost field in *pred. dist.* corresponds to the iteration which is being executed by the main core, and SpC3 is stored there. When the main core has executed a loop iteration or omitted execution of a loop iteration, each core-ID which is stored in *pred. dist.* is shifted left. Meanwhile, each field in *liid list* is also shifted left. Then, the iteration corresponding to the rightmost field in *pred. dist.* have not been assigned to any cores yet. In this figure, '-1' is stored in the rightmost field in *pred. dist.* This value means that the execution of the iteration have not been assigned to any cores yet. In addition, the iteration ID which is pushed out from the leftmost field is set to the rightmost field in *liid list* and reused to limit the maximum bit width for iteration IDs.

Next, we discuss the execution model of loop iteration purge with the storing and purging flow of loop entries shown in Fig.4. When SpC1 stores a loop entry, the state of MemoTbl is as shown in Fig.4 (X). After that, when the main core purges

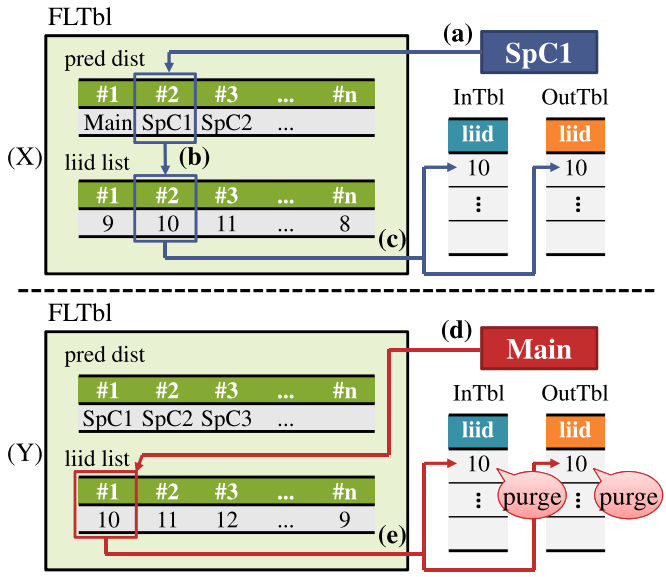

Fig. 4. Storing and purging flow of loop entries.

the loop entry which is stored by SpC1, the state of MemoTbl becomes as shown in Fig.4 (Y). First, when SpC1 stores the loop entry, SpC1 searches the *pred. dist.* fields for the core-ID of SpC1 to fetch the ID of the iteration which was executed speculatively by SpC1 (a). Then, SpC1 fetches the iteration ID '10' from *liid list* (b) and stores the ID into *liid* field of the entry which is stored in IAO-tables by SpC1. Next, assume that the main core completes the execution of the iteration which has ID '9.' Then, as mentioned before, each field in *pred. dist.* and *liid list* is shifted left, and *pred. dist.* and *liid list* become as shown in (Y). Next, assume that the main core omits the execution of the loop iteration which has ID '10' by computation reuse. Then, the main core fetches the iteration ID from the leftmost field of *liid list* for purging the loop entry which was reused (d). Finally, the main core purges the loop entry which has the same ID as the fetched ID. Incidentally, when the main core fails to reuse the loop iteration which has ID '10,' the main core purges the loop entry similarly.

B. Hardware Extension for Loop Iteration Storing Filter

We discuss how Auto-Memoization Processor avoids storing four types of loop entries in IAO-tables described in section IV-B.

To avoid storing (A)-type entries, the main core has to determine whether each executed instruction region is a function or a loop. If the main core finishes an loop iteration while *ForL* indicates that it is a loop entry, the main core avoids storing the entries into IAO-tables.

To avoid storing (B)-type entries, an SpC has to determine whether the main core has already started executing the iteration which has been executed speculatively by the SpC, or not. If an SpC stores loop entries while the core-ID of the SpC is in the leftmost field in *pred. dist.* or the core-ID of the SpC is not in any fields in *pred. dist.*, the SpC avoids storing the loop entries. This is because the SpC finds that the main core has already started executing the iteration which has been executed speculatively by the SpC.

To avoid storing (C)-type entries, SpCs have to distinguish an outer loop in the nested loop from inner loops. Hence, we have added the new field *start flag* in MemoBuf for determining whether SpCs have started Parallel Speculative Execution for loops or not. When an SpC starts Parallel Speculative Execution for a loop, *start flag* of the MemoBuf entry which is pointed by *MemoBuf_top* is set. Next, entering to the inner loop, *MemoBuf_top* is incremented. Then, the SpC do not set *start flag* of the entry which is pointed by *MemoBuf_top*. This is because the SpC has already started Parallel Speculative Execution for the loop outside of the inner loop. If an SpC finishes a loop iteration while *start flag* of the MemoBuf entry which is pointed by *MemoBuf_top* is set, the SpC stores the loop entries into IAO-tables. This is because the SpC finds that the loop is the outermost loop where the SpC has started Parallel Speculative Execution. On the other hand, if an SpC finishes a loop iteration while *start flag* of the MemoBuf entry which is pointed by *MemoBuf_top* is not set, the SpC avoids storing the loop entries into IAO-tables. This is because the SpC finds that the loop is an inner loop.

To avoid storing (D)-type entries, SpCs have to determine whether or not the main core is still executing the loop which the SpCs are now executing. Hence, we have added the new field *exec flag* in FLTbl for determining it. When the main core starts executing a loop, *exec flag* corresponding to the loop is set. If the backward branch is not-taken and the main core breaks out of the loop, *exec flag* corresponding to the loop is cleared. If an SpC finishes a loop iteration while *exec flag* corresponding to the loop is not set, the SpC avoids storing the loop entries.

C. Hardware Extension for Loop Iteration Searching Filter

To implement loop iteration searching filter, the main core has to determine whether loop entries for the loop iteration which will be executed next by the main core have been stored yet by one of the SpCs or not. Hence, we have added the new set of fields *search flag* in FLTbl for determining it. When an SpC finishes a iteration, the field of *search flag* which indicates the iteration is set. In addition, every time the main core completes the execution of an iteration, *search flag* is shifted left as *pred. dist.* and *liid list*. If the leftmost field in *search flag* is set, the main core searches MemoTbl. This is because loop entries for the iteration which will be executed next by the main core are already stored in IAO-tables by SpCs. On the other hand, if the leftmost field in *search flag* is not set, the main core avoids searching MemoTbl because SpCs have not completed executing the iteration speculatively yet.

VI. Performance Evaluation

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

TABLE I
SIMULATION PARAMETERS

MemoBuf	64 KBytes
MemoTbl CAM modeled on DC18288 (L-latency)	128 KBytes
MemoTbl CAM modeled on eFlexCAM (L-latency)	8 KBytes
MemoTbl CAM modeled on eFlexCAM (S-latency)	8 KBytes
L-latency	
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 cycles/32Bytes
S-latency	
Comparison (register and CAM)	3 cycles/32Bytes
Comparison (Cache and CAM)	4 cycles/32Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32Bytes
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

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 performance of the new model proposed in this paper. The simulation parameters are shown in TABLE I. The cache structure and the instruction latencies are based on SPARC64-III[10]. To confirm whether the performance degradation with a smaller CAM is suppressed with proposed mechanisms or not, we have evaluated a model with each two types of CAMs. One is modeled on DC18288[11] and we define the size of the CAM as 128KBytes (32Bytes $\times$ 4K lines). The other is modeled on eFlexCAM[12] and we define the size of the CAM as 8KBytes (32Bytes $\times$ 256 lines). The latencies of these two types of CAMs are defined on the assumption that the clock of the processor is about 2GHz, and is 10-times faster than the 128KBytes CAM, and 4-times faster than the 8KBytes CAM. We call the access latency for the 128KBytes CAM 'L-latency' and the access latency for the 8KBytes CAM 'S-latency.' We have also evaluated the model that the size of the CAM is 8KBytes and the latency is L-latency. By comparing the change in the performance which is caused by the difference in the size of CAMs, we can evaluate the benefits from the proposed model.

B. Execution Cycles with SPEC CPU95

We have evaluated the new proposed model by executing are 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. The evaluation results are shown in Fig.5. We have evaluated following six models,

- (N) No-memoization model. (baseline)
- (M_{128K}) Previous memoization model with 128KBytes CAM.
- (M_{8K}) Previous memoization model with 8KBytes CAM and L-latency.
- (P_{8K}) The model based on (M_{8K}) with proposed loop iteration purge and loop iteration storing filter.
- (C_{8K}) The model based on (P_{8K}) with proposed loop iteration searching filter.
- (C'_{8K}) The model based on (C_{8K}) with S-latency.

and Fig.5 shows the execution cycles of these models. Each bar in Fig.5 is normalized to the number of executed cycles of (N) the model without memoization.

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 the shared second-level data cache miss penalties ('**D$1**', '**D$2**'), and the register window miss penalty ('**window**') respectively.

First, in many benchmark programs such as 147.vortex, 101.tomcatv and 107.mgrid, the performance of (M_{8K}) is lower than (M_{128K}). This means that some computation blocks which can be reused with the previous model (M_{128K}) was not really reused with (M_{8K}). This is because the number of CAM entries is decreased and beneficial entries which are expected to be reused tend to be purged with (M_{8K}).

On the other hand, with many programs such as 101.tomcatv and 107.mgrid, the proposed model (P_{8K}) provides almost the same performance as the previous model (M_{128K}). This means that in many programs, the usage efficiency of MemoTbl increases and the performance degradation with the 8KBytes CAM is suppressed by purging useless loop entries and avoiding storing useless loop entries.

However, with 130.li, 134.perl and 147.vortex, the performance degradation with the 8KBytes CAM can not be much suppressed. Among them, with 130.li and 147.vortex, the proposed model (P_{8K}) can not help because the most of IAO-tables entries are function entries. Next, with 134.perl, it takes much time before a function entry is reused after it is stored into IAO-table. Thus, these functions can not be reused and the performance degradation by reducing the CAM size can not be suppressed.

Next, in all programs, the proposed model (C_{8K}) provides almost the same performance as the proposed model (P_{8K}) or more higher performance than it. Above all, 124.m88ksim, 129.compress, 130.li and 134.perl are improved by the reduction of 'read' cycles. This reason is that the main core avoids searching when loop entries for the iteration which will be executed next by the main core are not stored.

Moreover, 101.tomcatv, 102.swim, 103.su2cor, 104.hydro2d and 146.wave5 are improved by the reduction of 'exec' cycles. This is because overhead filter allows computation reuse to be applied to these loops more frequently by searching IAO-tables only if loop entries for the iteration which will be executed next by the main core are stored.

As mentioned above, we confirmed that the performance degradation with the 8KBytes CAM is suppressed in (C_{8K}) although the latency is the same as 128KBytes CAM. If more

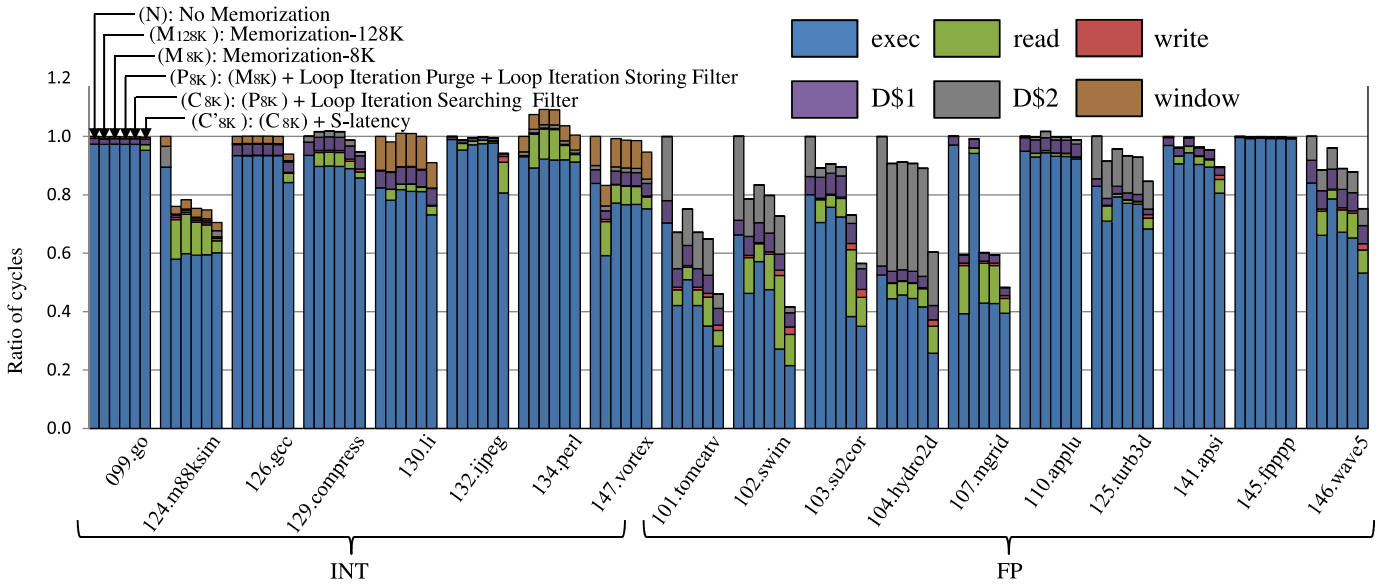

Fig. 5. Ratio of cycles (SPEC CPU95).

actual S-latency is used, the search overhead will decrease and the performance will be improved. The result of the proposed model with S-latency is shown as (C'_{8K}) in Fig.5. As we can see in the result, the proposed model (C'_{8K}) has less 'read' cycles in many programs as expected, and not only 'read' but also 'exec' is considerably reduced. This is because that the overhead filter allows computation reuse to be applied to instruction regions more frequently, and the reuse hit rate raises with (C'_{8K}).

In conclusion, the performance of the proposed model (C_{8K}) is better than the previous model (M_{128K}) as a whole. The model (C_{8K}) improves the maximum speed-up from 40.5% to 40.6%, and the average from 9.6% to 10.6% although the size of CAM is reduced to 1/16. In addition, the model (C'_{8K}) reduces the cycles by 58.4% in maximum and by 20.0% in average. These results show that the proposed mechanisms allow Auto-Memoization Processor to be implemented with a smaller CAM without performance degradation.

C. Hardware Implementation Costs

We estimate the additional hardware costs for the proposed model with the 8Kbytes CAM. When Auto-Memoization Processor has three SpCs, n in Fig.3 should be 8 because Auto-Memoization Processor assigns two iterations to each core. Hence, the width of *liid list* is $8 \times log_2 8 = 24$bit and the width of *liid* is $log_2 8 = 3$bit. Moreover, each of *exec flag* and *start flag* is 1bit and *search flag* is 8bit. When the width of each additional field is multiplied by the number of lines, the additional hardware cost is about 1.2KBytes in total.

VII. Conclusions

In this paper, we proposed a new algorithm for purging useless loop entries based on characteristics of loops. We also proposed a new storing and searching filter against useless entries for loops. The evaluation result with SPEC CPU95 benchmark suite shows that these mechanisms allow Reuse Table to be implemented with a smaller CAM without any performance degradation.

One of our future works is to investigate how an energy consumption changes by implementing Auto-Memoization Processor with a smaller CAM.

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] T. Ikegaya, T. Tsumura, H. Matsuo, and Y. Nakashima, "A Speed-up Technique for an Auto-Memoization Processor by Collectively Reusing Continuous Iterations," in *Proc. 1st Int'l. Conf. on Networking and Computing (ICNC'10)*, Nov. 2010, pp. 63–70.

[3] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proc. 29th Annual ACM/IEEE Int'l Symp. on Microarchitecture (MICRO-29)*, Dec. 1996, pp. 226–237.

[4] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *Proc. 30th Annual ACM/IEEE Int'l Symp. on Microarchitecture (MICRO-30)*, Dec. 1997, pp. 281–290.

[5] A. Roth and G. S. Sohi, "Register integration: A simple and efficient implementation of squash reuse," in *Proc. 33rd Annual ACM/IEEE Int'l Symp. on Microarchitecture (MICRO-33)*, Dec. 2000.

[6] Y. Wu, D. Chen, and J. Fang, "Better exploration of region-level value locality with integrated computation reuse and value prediction," in *Proc. 28th Annual Int'l Symp. on Computer Architecture (ISCA-28)*, Jul. 2001, pp. 98–108.

[7] C. Molina, A. González, and J. Tubella, "Trace-level speculative multithreaded architecture," in *Proc. 20th IEEE Int'l Conf. on Computer Design: VLSI in Computers and Processors (ICCD'02)*, Sep. 2002.

[8] ——, "Compiler analysis for trace-level speculative multithreaded architectures," in *Proc. 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, Jun. 2005.

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

[10] *SPARC64-III User's Guide*, HAL Computer Systems/Fujitsu, May 1998.

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

[12] eSilicon Corporation, *HiSilicon Licenses eSilicon's 40nm Silicon-Proven TCAMs for High-Performance Network Chips*, Dec. 2011.