This paper is author's private version of the paper published as follows:
Proc. 3rd Int'l Conf. on Networking and Computing (ICNC'12), pp.49-57

DOI: 10.1109/ICNC.2012.17

A Speed-up Technique for an Auto-Memoization Processor by Reusing Partial Results of Instruction Regions

Kazutaka KAMIMURA*, Ryosuke ODA*, Tatsuhiro YAMADA*,
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 an auto-memoization processor based on computation reuse. The auto-memoization processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically. In the past model, computation reuse cannot be applied if the current input sequence even differs by only one input value from the past input sequences, since processing results will differ. This paper proposes a new partial reuse model, which can apply computation reuse to the early part of a reusable block as long as the early part of the current input sequence matches one of the past sequences. In addition, in order to acquire sufficient benefit from the partial reuse model, we also propose a technique that reduces the searching overhead for memoization table by partitioning it. The result of the experiment with SPEC CPU95 suite benchmarks shows that the new method improves the maximum speedup from 40.6% to 55.1%, and the average speedup from 10.6% to 22.8%.**

***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.

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]. The SPARC T4[2] with eight cores, the Opteron[3] with 16 cores, and the TILE64[4] with 64 cores are available now, and many-core processors such as the TILE-Gx processor[5] with 100 cores are planned to be shipped.

A program generally 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 an auto-memoization processor based on computation reuse[6][7]. 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 has a potential for breaking through the stone wall against which the speedup techniques based on ILP have been up.

The auto-memoization processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically. In the past model, computation reuse cannot be applied if the current input sequence even differs by only one input value from the past input sequences, since processing results will differ.

In this paper, we propose a new partial reuse model, which can apply computation reuse to the early part of a reusable block as long as the early part of the current input sequence matches one of the past sequences. In addition, in order to acquire sufficient benefit from the partial reuse model, we also propose a technique that reduces the searching overhead for memoization table by partitioning it.

II. Related Work

Studies for extracting ILPs with speculative executions based on value prediction have been proposed by Lipasti et al.[8] or Wang et al.[9] 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.[10] 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%.

Tuck et al.[11] have proposed *single fetch path multi-threaded value prediction*. In this prediction model, whenever the executing thread is stopped by cache misses, a new speculative thread, which uses the predicted value as the load value, is spawned and executed. After that, whenever a fetched load value matches the predicted value, this prediction model writes back the outputs of the speculative thread, which corresponds to the predicted value, to the cache.

Some hybrid methods of computation reuse and value prediction have been also studied. Wu et al.[12] 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.[13][14] have proposed a combination model of speculative thread and non-speculative thread. The execution results of speculative thread are stored into the FIFO called a *look ahead buffer*, 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.

Some compiler technologies for SpMT models have been also studied. Bhowmik et al.[15] have proposed a general compiler framework for a wide variety of SpMT architectures. This compiler considers data dependencies, control dependencies and thread size together and exploits parallelism from not only loops but also non-loop regions.

Li et al.[16] have proposed a cost estimation model for SpMT systems. This model estimates the overhead of SpMT quantitatively and computes the effect of SpMT. The computational result can be used to determine whether SpMT leads to a good performance or a bad performance.

Gao et al.[17][18] have proposed *loop recreation*. It is a technique for decreasing inter-iteration dependencies by restructuring a loop. The compiler constructs a data dependency graph of a loop and solves a min-cut problem on the graph.

In contrast to these studies, the parallel speculative execution model we have proposed is a non-symmetric SpMT model based on value prediction, and uses computation reuse technique. Our model has two advantages over [12]. The one is that there is no need to be assisted by compiler for computation reuse. The other is that there is no need to squash speculative executions. Molina's model [13] is similar to our model. However, our model can reuse some instruction regions which require memory read as their inputs.

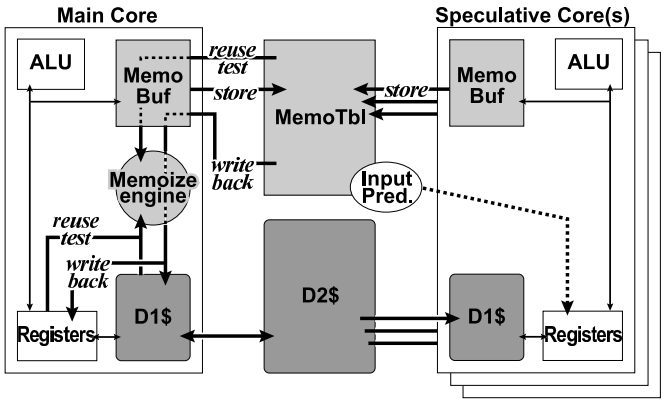

Fig. 1. Structure of Auto-Memoization Processor.

III. Research Background

In this section, we describe the auto-memoization processor which we have proposed, and its behavior 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**[19] 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[20]. However, it requires rewrite of target programs, and the traditional load-modules or binaries cannot benefit from memoization. Furthermore, the effectiveness of memoization is influenced much by programming styles. Rewriting programs using memoization occasionally makes the programs slower. Memoization costs a certain overhead because it is implemented by software.

On the other hand, the **auto-memoization processor**, which we have proposed, makes traditional load-modules faster without any software assist. There is no need to rewrite or recompile programs. The auto-memoization processor dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically. However, a loop iteration, which uses its iterator variable as one of its inputs, never benefits from memoization. Hence, we have installed some speculative cores to our auto-memoization processor for reusing loop iterations. The brief structure of the processor is shown in Fig.1.

The 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. MemoBuf works as a write buffer for MemoTbl.

```
1  int a = 3, b = 4, c = 8;
2  int calc(x){
3    int tmp = x + 1;
4    tmp = tmp + a;
5    if(tmp < 7)
6      tmp = tmp + b;
7    else
8      tmp = tmp + c;
9    return(tmp);
10 }
11 int main(void){
12   calc(2); /* x = 2, a = 3, b = 4 */
13   b = 5; calc(2); /* x = 2, a = 3, b = 5 */
14   a = 4; calc(2); /* x = 2, a = 4, c = 8 */
15   a = 3; calc(2); /* x = 2, a = 3, b = 5 */
16   return(0);
17 }
```

Fig. 2. A sample code.

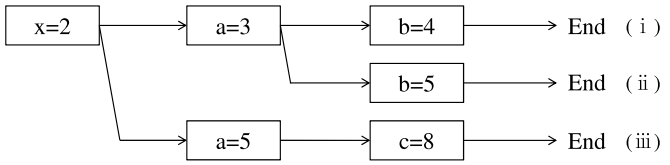

Fig. 3. Tree of input sequences.

Entering to a memoizable region, the processor refers to MemoTbl and compares the current input sequence with former input sequences which are stored in MemoTbl. If the current input sequence matches one of the stored input sequences on MemoTbl, the memoization engine writes back the stored outputs, 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 inputs and the 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, whose entries correspond to input/output sequences. A MemoBuf entry has a stack pointer (*SP*) and a return address (*retOfs*). A MemoBuf entry also has an input sequence (*Read*) and an output sequence (*Write*).

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, the auto-memoization processor should hold input sequences as a tree structure.

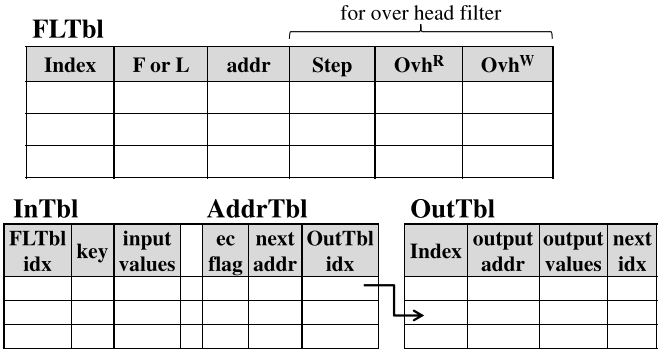

Fig. 4. Structure of MemoTbl.

For example, if the processor executes the sample program shown in Fig.2, the tree structure of input sequences for the function `calc` will be formed as shown in Fig.3. Each node of the tree represents input values, and each edge represents a relationship that the input values represented by the connected nodes are referred to sequentially. Here, `End` represents the terminal of a sequence and each input sequences (i), (ii) and (iii) corresponds to the function call at line 12, 13 and 14 respectively in the sample code. In the input sequence (i) and (ii), the variable `b` is read as the third input, whereas the variable `c` is read in the input sequence (iii). This is because that the results of branch instruction at line 5 differs, owing to that the value of the second input variable `a` changes.

Next, let us see about the structure of MemoTbl shown in Fig.4. 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.

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 matching can be done 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 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*). The fields for the overhead filter hold the execution cycles of the region (*Step*) and its 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. Each InTbl entry can hold single cache line, and an input sequence over multiple cache lines is registered onto InTbl by using several entries. An InTbl entry holds an index key for parent entry (*key*) and input values (*input values*). When a variable is read as an input

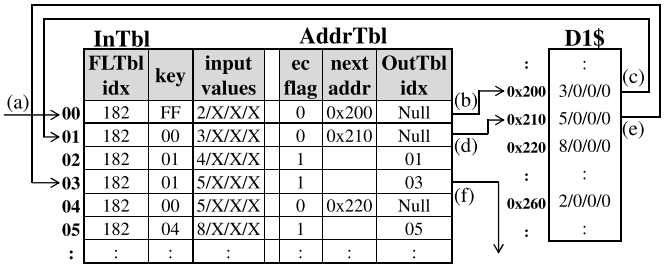

Fig. 5. Input matching flow on MemoTbl.

value, its whole cache line is stored in an InTbl entry, masking unreferred values with *don't care* bits.

An AddrTbl entry has an input address which should be tested next (*next adr*). AddrTbl has the same number of entries as InTbl entries, and each AddrTbl entry corresponds to the InTbl entry which has the same index. An AddrTbl entry 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.

An OutTbl entry has addresses (*output addr*) and values (*output values*) of an output sequence. An OutTbl entry also has an index for next OutTbl entry (*next idx*) because an output sequence is stored over multiple OutTbl entries.

As mentioned before, the auto-memoization processor also has some speculative cores. While the main core executes a memoizable computation block, speculative cores execute the same block using predicted inputs, and stores the results into MemoTbl. The inputs are predicted by stride prediction using the last two input sequences stored in FLTbl. If the input prediction succeeds, the main core can omit intended execution by reusing the result registered by one of the speculative cores.

Here, the bandwidth of MemoTbl does not matter. The speculative cores have their own MemoBuf, and their write access to MemoTbl will shift off each other because speculative executions are issued sequentially. Hence, there occur little conflicts while accessing to MemoTbl.

B. Execution Mechanism

Fig.5 shows how the input sequences shown in Fig.3 are registered onto InTbl/AddrTbl. Here, `X` in *input values* represents a *don't care* nibble in the cache line, and will not be tested for computation reuse. Furthermore, `0x200`, `0x210` and `0x220` in *next addr* correspond to the variable `a`, `b` and `c` in Fig.2. In addition, Fig.5 also shows an input matching flow on MemoTbl as (a)...(f). This flow represents the reuse test for the function call at line 15 in Fig.2. First, the processor reads the values of registers when the start address of the reusable instruction region is detected. Then, the processor searches the root entry whose *key* is `FF` and whose *input values* match the values on the current registers. Now, (a) the line `00` matches. Next, (b) the address of `0x200` is read because *next addr* of the entry `00` in AddrTbl indicates `0x200`. Then, (c) the processor searches the entry whose *key* is `00` and whose *input values* match the values of `0x200`. This process is (d)(e) applied repeatedly until a mismatch of input values occurs. In this example, all reuse tests for the current input values succeed. Therefore (f) the processor can get the output values by using the index of *OutTbl idx* stored in the terminal entry. 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.

Meanwhile, accessing MemoTbl causes overhead inevitably. Through input matching, searching InTbl, referring AddrTbl, and reading caches cost a certain time. When input matching has succeeded, outputs of the reusable block should be written back from OutTbl. This also costs some cycles. We call these two kinds of overheads '**reuse overheads**.'

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, the auto-memoization processor has a structure which estimates the effect of reuse, and avoids memoizing unsuitable computation blocks. With the execution cycles $step$ of the block, the processor calculates the performance gain in terms of omitted cycles as

$$step - Ovh^R - Ovh^W \tag{1}$$

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.

IV. Applying Partial Computation Reuse

In this section, we will propose a new processor model which can reuse instruction regions partially.

A. Execution Model of Partial Reuse

In the past model, computation reuse is abandoned if the current input sequence even differs by only one input value from the past input sequences, since processing results will differ. In addition, search overheads for such instruction regions are wholly added to the execution cycles. However, execution results for the early part of a reusable region should be same as long as the early part of the current input sequence matches one of the past input sequences. For example in Fig.2, when the function `calc` is called at line 12, the value of the variable `b` is `4`. On the other hand, when the function `calc` is called at line 13, the value of the variable `b` is `5`. Therefore, traditional auto-memoization processor fails to reuse `calc` because the input values do not match completely any of the past input sequences. In fact, the return values of these function calls are different, since the return value of the second function call is `11` while the value of the first function call is `10`. However, between these two function calls, values of the argument `x` and the global variable `a` are unchanged. Therefore, when `calc` is called at line 13, the processor can start execution from at line 6, if it can write

back the value 6 to the local variable tmp. Similarly, when only the values of variable x are same, the processor can start execution from at line 4 if it can write back the value 3 to the local variable tmp. Consequently, even if the processor fails input matching, the execution of the instruction region can be partially omitted by writing back the intermediate results which correspond to the matched inputs. Hence, we propose a new partial reuse model, which can apply computation reuse to the early part of the reusable region as long as the early part of the current input sequence matches one of the past input sequences stored in MemoTbl. This method enables the auto-memoization processor to omit not only functions or loop iterations but also more smaller instruction regions, and some speedup will be achieved. In the following, we call this new reuse method '**partial computation reuse**' and the traditional reuse method '**full computation reuse**.'

B. Reuse Table Partitioning

As mentioned before, the processor acquires ability to reuse instruction regions partially. However, partial computation reuse can reduce less execution cycles per reuse as compared with full computation reuse. Hence, partial computation reuse will be easily influenced by the reuse overhead. Incidentally, the search overhead can be reduced by reducing CAM's access latency because InTbl is constructed by CAM. The access latency of the CAM depends on its depth[21], and we also propose a technique for reducing reuse overhead by partitioning MemoTbl into multiple small tables. Given the number of partition N, the traditional InTbl, which has the width of 32Bytes and the depth of 4K, is horizontally partitioned into N sub-InTbls. Each sub-InTbl has the width of 32Bytes and the depth of $4\text{K}/N$. Likewise, AddrTbl is horizontally partitioned into N sub-AddrTbls because each AddrTbl entry is correspond to an InTbl entry which has the same index.

To store input entries efficiently over multiple sub-InTbls, overconcentration of input entries in one sub-InTbl should be avoided. Therefore, when storing a new entry, it should be stored into the least crowded sub-InTbl. However, all input entries which have same parent entry should be stored into the same sub-InTbl. This feature enables the auto-memoization processor to distinguish whether the next input entry exists or not on sub-InTbls by just searching through single sub-InTbl.

As mentioned in the previous section, the overhead filter stops applying computation reuse to the instruction regions which are considered as unsuitable for computation reuse. Now, reducing the search overhead may lead to increasing the number of instruction regions whose performance can be improved by applying computation reuse. Therefore, with reuse table partitioning the overhead filter will less frequently stop applying computation reuse to instruction regions, and the reuse hit rate for such instruction regions will raise.

V. Implementation

This section describes an implementation of the new reuse model.

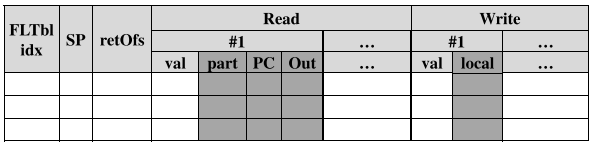

Fig. 6. Structure of expanded MemoBuf.

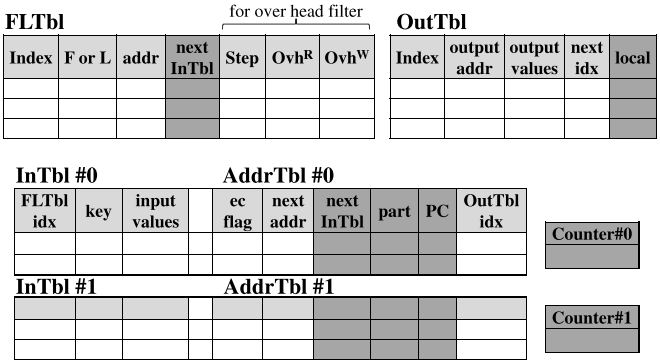

Fig. 7. Structure of expanded MemoTbl.

A. Hardware Extension

For partial computation reuse and reuse table partitioning, we have added some fields on MemoBuf and MemoTbl. The fields are shown with shade in Fig.6 and Fig.7. We will discuss the extensions for these two techniques separately.

1) Extension for Partial Computation Reuse: To achieve partial computation reuse, the new partial reuse model has to write back intermediate results of a function. In addition, it has to restart execution from the intermediate instruction in the function. Hence, we have added the new field *PC* for storing a program counter value, which represents where to be restarted, into *Read* fields in MemoBuf and AddrTbl in MemoTbl. Since outputs from partial computation reuse, or intermediate results, include the value of local variables of the functions, these values should be stored on the new partial reuse model. Hence, we have added the new 1-bit flag field *local* for distinguish whether the output values are global values or local values. With partial computation reuse, the output values correspond to the region, where input values are same so far, should be written back. For this reason, such output values are stored into MemoBuf while executing the function, but will be modified before reaching the end of the function. Therefore, the memoization engine stores the current partial outputs into OutTbl whenever a new input value is read. On the other hand, the memoization engine stores all input values into MemoTbl when the processor reaches the end of the function as usual. Therefore, MemoBuf should remember indices to OutTbl entries until the processor reaches the end of the function. Hence, we have added the field *Out* for storing indices to OutTbl entries into *Read* fields in MemoBuf. Now, AddrTbl also has to hold indices to OutTbl entries, but we do not need to add new field since the *OutTbl idx* in AddrTbl can be used for this purpose. In the traditional model, this field is used only in the terminal entry of each input sequence

to hold the index to its corresponding output sequence for full computation reuse. However, an output entry may be purged from OutTbl before the corresponding input entry is purged. Hence, we have added the new 1-bit flag field *part* for distinguishing whether the output values are valid or not.

2) Extension for Reuse Table Partitioning: To achieve reuse table partitioning, the new reuse model has to store input entries over multiple sub-InTbls, keeping a correct tree structure of input sequences. In addition, to avoid searching all sub-InTbls, input entries which have the same parent entry should be stored into same sub-InTbl. Therefore, each AddrTbl entry has to remember the index to sub-InTbl entry which should be accessed next. Hence, we have added the new field *next InTbl* for these indices. Furthermore, to avoid overconcentration of input entries in one sub-InTbl, we have added the counter *Counters* for storing how many entries are stored into the sub-InTbl.

In the new reuse model, each input sequence should have an output sequence for full computation reuse and some output sequences for partial computation reuse in OutTbl. Thus, it may decrease in performance due to running out of free space on OutTbl caused by storing more numerous output entries into OutTbl than the traditional model. To avoid this problem, we set a limit that partial computation reuse is applied only to functions, and not to loops. The execution cycles, which can be reduced by applying partial computation reuse to loop iterations, should be very few. Hence, this limit will not affect the performance. Furthermore, to avoid running out of free space on OutTbl, we increased the OutTbl size. OutTbl is implemented with not CAM but RAM, so increasing the size of OutTbl is not so serious. We will mention the assumed size of OutTbl in section VI.

B. Registering to MemoTbl

In the new reuse model, while executing functions, the memoization engine stores the inputs and the outputs of the function into MemoBuf as the traditional model. Then, if a local variable is modified, the output is stored into MemoBuf and its *local* flag is set. In addition, when a new input value is read, the current output entries on MemoBuf are stored into MemoTbl. Then, the index of the OutTbl entry and the current value of *PC* are stored into the newest input entry of MemoBuf. Reaching the end of the function, the memoization engine stores each output entry, if whose *local* flag is not set, on MemoBuf into MemoTbl. Then, the memoization engine refers the counters of sub-InTbls, and stores the new entry on the least crowded sub-InTbl, and sets *next InTbl* of the parent entry in AddrTbl to an index to the new entry.

C. Searching through MemoTbl

Fig.8 shows the state of MemoTbl when the processor completes the execution of the line 12 of the program shown in Fig.2, and also shows an input matching flow on MemoTbl as (a)...(g). Let us see this flow in the new reuse model.

When the function `calc` is called at line 13, the processor searches the root entry, whose *key* is `FF` and whose *input values* match the values of the current registers, through InTbl#0 which is indicated by *next InTbl* in FLTbl. Now, (a) the line `00` matches. Then, the processor notices that it is able to apply partial computation reuse because *part* of the entry `00` in AddrTbl is set as valid. Therefore, the processor holds this entry index `00` until the end of input matching. Next, (b) the address of `0x200` is read because *next addr* of the entry `00` in AddrTbl indicates `0x200`, and (c) the processor searches the entry, whose *key* is `00` and whose *input values* match the value of `0x200`, through InTbl#1 which is indicated by *next InTbl* of the entry `00` in AddrTbl. Then, *part* of the entry `10` in AddrTbl is also valid, and the processor updates the entry index, which has been stored for partial computation reuse, from `00` to `10`. This is because the output indicated by `10` omits more execution cycles than the former output indicated by `00`. After that, (d)(e) the processor fails full computation reuse because next input value `b` does not match.

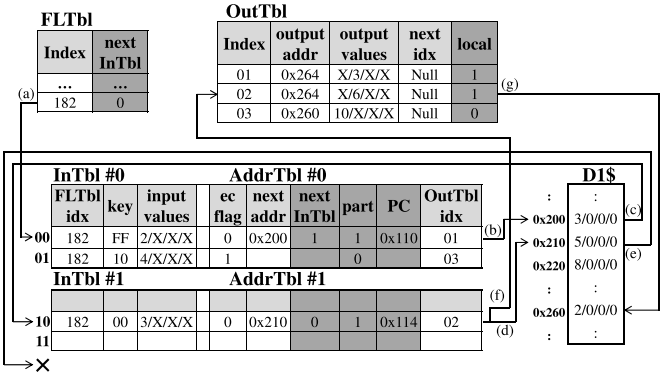

Fig. 8. Input matching flow on expanded MemoTbl.

Then, the new reuse model applies partial computation reuse by using the stored index `10`. The processor first (f) gets the output sequence from OutTbl by using the index of *OutTbl idx* stored in the entry `10` in AddrTbl, (g) writes back the output sequence to the registers and caches, and restart execution from `0x114` which is indicated by the entry `10` in AddrTbl.

VI. Performance Evaluation

A. Simulation Environment

We have developed a single-issue SPARC-V8 simulator equipped with the auto-memoization mechanisms and the three speculative cores. In this section, we will discuss the performance of the new reuse 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[22]. The on-chip CAM for InTbl in MemoTbl is modeled on DC18288[23] (32Bytes $\times$ 4K lines). On the other hand, the small CAM for sub-InTbl is modeled on eFlexCAM[24](32Bytes $\times$ 256 lines). For the reuse table partitioning model, we constructed InTbl with 16 small CAMs. The latencies of these two types of CAMs are defined on the assumption that the clock of the processor is about 2 GHz, and is 10-times faster than the CAM for the traditional model, and 4-times faster than the small CAM for the new reuse model.

TABLE I
SIMULATION PARAMETERS

MemoBuf	64 KBytes
MemoTbl CAM	128 KBytes
MemoTbl small CAM	8 KBytes
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 cycles/32Bytes
Comparison (register and small CAM)	3 cycles/32Bytes
Comparison (Cache and small 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

TABLE II
REDUCED EXECUTION CYCLES. (SPEC CPU95)

	Mean	Max
(M) Traditional memoization model	**10.6%**	**40.6%**
(P) Partial reuse model	11.0%	40.7%
(S) Reuse table partitioning model	22.5%	55.1%
(C) Hybrid model of (P) and (S)	**22.8%**	**55.1%**

In the new reuse model, we increased the size of OutTbl from 4K lines to 32K lines.

B. Execution Cycles with SPEC CPU95

We have evaluated the execution cycles. Workloads are all benchmark programs in SPEC CPU95 suites and are executed with 'train' dataset. All benchmark programs are compiled by gcc version 3.0.2 with '-msupersparc -O2' options, and linked statically. The evaluation results are shown in TABLE II and Fig.9. We have evaluated following five models,

(N) No-memoization model (baseline)
(M) Traditional memoization model
(P) Partially reusing model
(S) Reuse table partitioning model
(C) Hybrid model of (P) and (S)

and Fig.9 shows the execution cycles of these models. Each bar is normalized to the number of executed cycles of (N) the model without memoization.

The legend in Fig.9 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, note that the new reuse model (P) reduces the 'exec' of some benchmark programs such as 124.m88ksim, 134.perl, 147.vortex and 141.apsi. This means that some instruction regions, which are not reused with the traditional model, are reused by partial reuse with the new reuse model (P).

```
000317f0<killtime>:
   317f0: sethi %hi(0x236800), %o1
      :
   31814: ld [ %o2 + %o4 ], %o0
      :
   31834: bpos 31814 <killtime+0x24>
   31838: add %o2, 4, %o2
      :
   3189c: ld [ %o2 + %o4 ], %o1
      :
   318c4: retl
   318c8: nop
```

Fig. 10. An assembly code of killtime function.

Especially, in the execution cycles of 124.m88ksim, not only 'exec' but also 'read' is reduced. Fig.10 shows the assembly code of the most partially reused function in 124.m88ksim: `killtime`. In this program, the `bpos` instruction at `0x31834` is a branch instruction, and the instruction region from `0x31814` to `0x31834` is a loop. With the traditional model (M), the processor fails to apply computation reuse to the function `killtime`, and executes it from `0x317f0` as usual. Next, the processor tries to reuse the loop and causes a certain search overhead. On the other hand, in the new reuse model (P), the execution of instruction region from `0x317f0` to `0x3189c` is omitted by applying partial computation reuse to the function `killtime`. Because of this, the total search overhead is reduced.

However, for the benchmark programs other than 124.m88ksim, 134.perl, 147.vortex and 141.apsi, we cannot see the benefit from new reuse model (P), because the overhead filter stops applying both full computation reuse and partial computation reuse to the most of functions.

Next, note that the new reuse model (S) reduces the 'read' of some benchmark programs such as 124.m88ksim, 134.perl and 107mgrid. This means that the search overhead is reduced by partitioning reuse table. Also, in the results of many programs, not only 'read' but also 'exec' is considerably reduced. As we expected in section IV-B, this is because that the overhead filter stops applying computation reuse to instruction regions less frequently, and the reuse hit rate for instruction regions raises with the new reuse model. On the other hand, 'exec' of 147.vortex is increased and the performance of the program declines. This reason is that more instruction regions are registered onto MemoTbl, and instruction regions which are reused with the traditional model may be purged away from MemoTbl before they are reused. The usage efficiency of InTbl could be decreased by partitioning InTbl into multiple sub-InTbl, but the evaluation results show that the efficiency is not decreased for most of benchmark programs. We have verified that the performance degradation by partitioning InTbl is about 0.5% in average. This means that even the naive policy that the new entry is stored into the sub-InTbl with the fewest

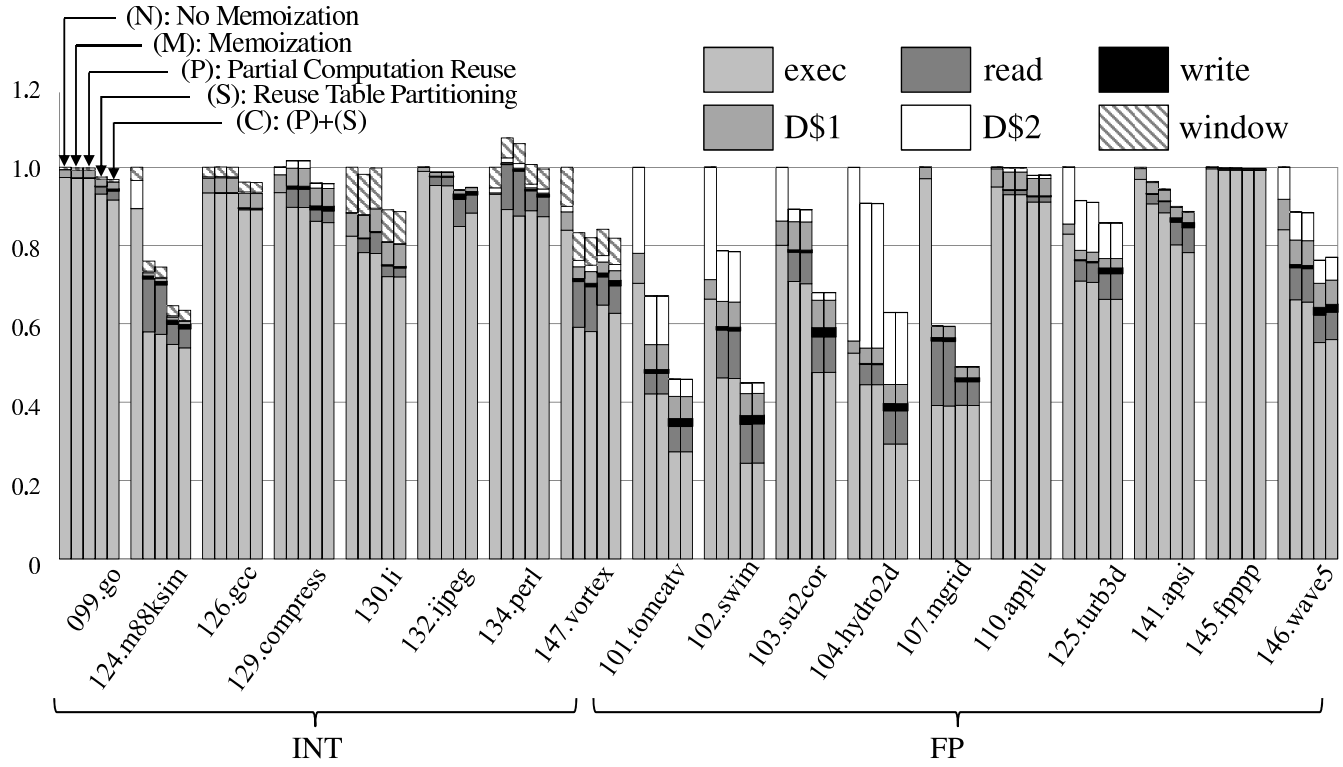

Fig. 9. Ratio of cycles (SPEC CPU95).

valid entries works well for usage efficiency of InTbl.

Finally with the hybrid model (C), 'exec' cycles of some benchmark programs such as 099.go, 124.m88ksim, 134.perl, 147.vortex and 141.apsi are reduced in compared with both (P) and (S). The execution cycles of these programs other than 099.go are reduced also with the model (P). This means that two speedup techniques which we proposed have produced good synergistic effect. Through the execution of 099.go, the overhead filter stops applying computation reuse to instruction regions less frequently than with the traditional model (M), because the search overhead is reduced by partitioning reuse table. Hence, with the new hybrid model (C), more instruction regions are tried to be reused and more functions are partially reused than the reuse model (P). In fact, 12.0% of functions are partially reused with the new hybrid model (C) whereas few functions are partially reused with the model (P). This means that reducing the overhead makes the partial computation reuse more effective.

Incidentally, we have proposed the unwinding model[7] previously. This model integrates multiple continuous iterations into one reusable block and detects how many iterations should be integrated into one reusable block automatically. The unwinding model has reduced execution cycles by a maximum of 57.6%, and an average of 26.0% with SPEC CPU95 FP benchmark suite. On the other hand, the hybrid model (C) proposed in this paper can reduce execution cycles by a maximum of 55.1%, and an average of 30.9%. Therefore, this hybrid model (C) is better than the unwinding model totally. However, these two models may be merged together for achieving further speed-up.

Moreover, compared with parallel execution on a multi-core processor assisted by some automatic parallelization compilers, the hardware cost and the energy consumption of the auto-memoization processor is fairly larger, because ternary CAMs are required for its implementation. Furthermore, some parallelization-friendly workloads, such as 101.tomcatv, 102.swim, 104.hydro2d or 107.mgrid from SPEC CPU95 FP benchmark suite, will show a greater performance gain with parallel execution than with the auto-memoization processor. However, the auto-memoization processor has some advantages against parallelization. First of all, the auto-memoization processor is completely binary compatible, and there needs no software assist such as recompilation. Secondly, programs which have no potential parallelism can gain performance with memoization. Additionally, memoization and parallelization are not mutually exclusive, and they can complement each other.

In conclusion, the performance of the new hybrid model (C) is better than the traditional model (M) as a whole. The model (C) improves the maximum speedup from 40.6% to 55.1%, and the average from 10.6% to 22.8%.

VII. Conclusions

In this paper, we proposed a new partial reuse model, which can apply computation reuse to the early part of a reusable block as long as the early part of the current input sequence matches one of the past input sequences. In addition, in order

to acquire sufficient benefit from partial reuse model, we also proposed a technique that reduces the search overhead for the memoization table by partitioning it. Through an evaluation with SPEC CPU95 suite benchmark programs, it is found that the new model improves the maximum speedup ratio from 40.6% to 55.1%, and the average ratio from 10.6% to 22.8%.

One of our future works is merging this model with other low-overhead models such as [25] that we had proposed. In this paper, we have implemented this partial reuse model on a simple single-issue processor architecture. Implementing this model on more recent architecture such as superscaler, and trying to merge other ILP-based methods and the memoization mechanism are also our future works.

REFERENCES

[1] ARM Ltd, *The ARM Cortex-A9 Processors*, Sep 2007.

[2] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc T4: A Dynamically Threaded Server-on-a-Chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, Apr. 2012.

[3] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture," *IEEE Micro*, vol. 27, no. 2, pp. 10–21, Aug. 2007.

[4] Tilera Corporation, *Product Brief: TILE64 Processor*, 2007.

[5] ——, *TILE-Gx Processor Family Product Brief*, 2009.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] N. Tuck and D. M. Tullsen, "Multithreaded Value Prediction," in *Proc. 11st Int'l Symp. on High-Performance Computer Architecture (HPCA-11)*, Feb. 2005.

[12] 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.

[13] 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.

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

[15] A. Bhowmik and M. Franklin, "A General Compiler Framework for Speculative Multithreaded Processors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 8, Aug. 2004.

[16] Y. Li, Y. Zhao, Y. Wei, and Y. Du, "A Cost Estimation Model for Speculative Thread Partitioning," in *Proc. 8th IEEE Int'l Symp. on Parallel and Distributed Processing with Applications (ISPA-10)*, Sep. 2010.

[17] L. Gao, L. Li, J. Xue, and T.-F. Ngai, "Loop Recreation for Thread-Level Speculation," in *Proc. 13th Int'l Conf. on Parallel and Distributed Systems (ICPADS'07)*, Dec. 2007.

[18] L. Gao, J. Xue, and T.-F. Ngai, "Loop Recreation for Thread-Level Speculation on Multicore Processors," *Softw. Pract. Exper.*, vol. 40, no. 1, Jan. 2010.

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

[20] 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.

[21] B. Agrawal and T. Sherwood, "Ternary CAM Power and Delay Model:Extensions and Uses," *IEEE VLSI*, vol. 16, no. 5, pp. 554–564, May 2008.

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

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

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

[25] R. Oda, T. Yamada, T. Ikegaya, T. Tsumura, H. Matsuo, and Y. Nakashima, "Input entry integration for an auto-memoization processor," in *Proc. 2nd Int'l Conf. on Networking and Computing (ICNC'11)*, Nov. 2011, pp. 179–185.