This paper is author's private version of the paper published as follows:
Proc. 2nd Int'l. Conf. on Networking and Computing (ICNC'11), pp.179–185
Copyright (C) 2011 IEEE
DOI: 10.1109/ICNC.2011.34

Input Entry Integration for an Auto-Memoization Processor

Ryosuke ODA*, Tatsuhiro YAMADA*, Tomoki IKEGAYA*,
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 table for registering inputs/outputs is implemented by a ternary CAM, and the input sequences are stored onto the table, being folded into tree forms. This paper proposes a new registration model for merging multiple input entries into a single entry. The new model can efficiently store input values and can reduce the search cost. The result of the experiment with SPEC CPU95 suite benchmarks shows that the new model improves the maximum speedup ratio from 40.5% to 50.0%, and the average speedup ratio from 10.5% to 16.4%.**

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.

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[1][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 has a potential for breaking through the stone wall against which the speedup techniques based on ILP have been up.

In the auto-memoization processor, the table for registering inputs/outputs is implemented by a general-purpose ternary CAM (Content Addressable Memory). The width of CAM is limited, and the processor can't register any input sequence in a single entry. Hence, the input sequences are stored onto the table by using multiple entries, being folded into tree forms.

In this paper, we propose a new registration model which integrates multiple entries into a single entry by sorting them. This can make reuse overhead lower and the usage of input entries smaller. We also propose a model for keeping tree structures correctly when input entry integration is applied.

II. Research Background

In this section, we describe about an auto-memoization processor and it's behavior.

A. Auto-Memoization Processor

Computation reuse is a speed-up technique. It is storing the input sequences and the results of some computation regions, 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[3] to apply computation reuse to computation regions in programs.

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 regions, and memoizes them automatically. However, a loop, 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 loops. 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 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

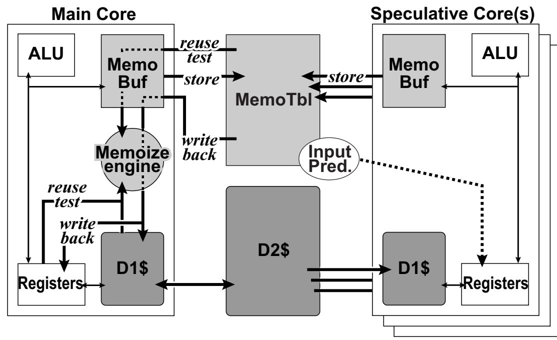


Fig. 1. Structure of Auto-Memoization Processor.

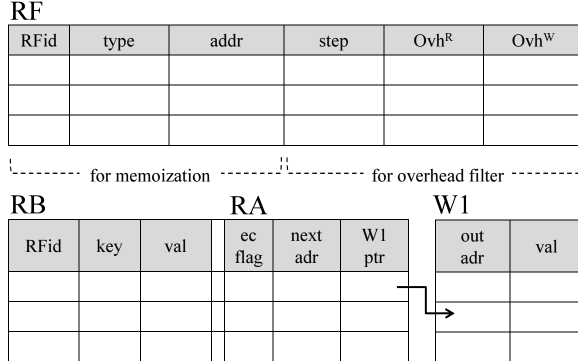


Fig. 2. Structure of MemoTbl.

former input sequences which are stored in MemoTbl. If the current input sequence matches with one of the stored input sequences on MemoTbl, the memoization engine writes back the stored outputs, associated with the input sequence, to the cache and the registers. This omits the execution of the region and reduces the total execution time.

If the current input sequence does not match with 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.

The MemoTbl consists of four tables. They are **RF** for the start addresses of the instruction regions, **RB** for the input data sets of the instruction regions, **RA** for the input address sets of the instruction regions and **W1** for the input address sets of the instruction regions. The structure of MemoTbl is shown in Fig.2.

Each RF line corresponds to a reusable computation region. One RF line 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 II-C. The fields for computation reuse store whether the region is a function or a loop (*type*) and the start address of the region (*addr*). The fields for overhead filter store the execution cycles of the region (*step*) and it's past reuse overhead (*Ovh*).

RB is implemented with a ternary CAM, so that input matching can be completed fast by associative search. The CAM has limited width, so the processor can't register any input sequence in a single entry. Each RB entry can hold single cache line, and an input sequence over multiple cache lines is registered onto RB using several entries. An RB entry holds an index key for parent entry (*key*) and input values (*val*). Each bit in the *val* fields can represent 0, 1 or *don't care*. When a variable is read as an input, its whole cache line is stored in an RB entry, masking unassociated bits as *don't care*.

An RA line has an input address which should be tested next (*next adr*). RA has the same number of entries as RB entries, and each RA entry is correspond to an RB entry which has the same index. An RA entry has a flag, which shows whether it is a terminal entry of an input sequence (*ec flag*), and if the entry is a terminal, it has a pointer (*W1 ptr*), which refers a W1 entry for associated outputs.

A W1 entry has addresses (*out adr*) and values (*value*) of an output sequence. If all inputs of a reusable region have matched with one of the stored input sequence on MemoTbl, the processor can get the output sequence from W1.

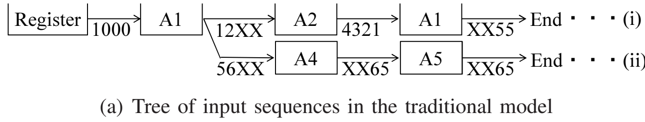
As mentioned above, the auto-memoization processor also has some speculative cores. While the main core executes a memoizable computation region, speculative cores execute the same region using predicted inputs, and stores the results into MemoTbl. The inputs are predicted by stride prediction using the last two input sequences stored in RF. 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 occurs little conflicts between the access to the CAM.

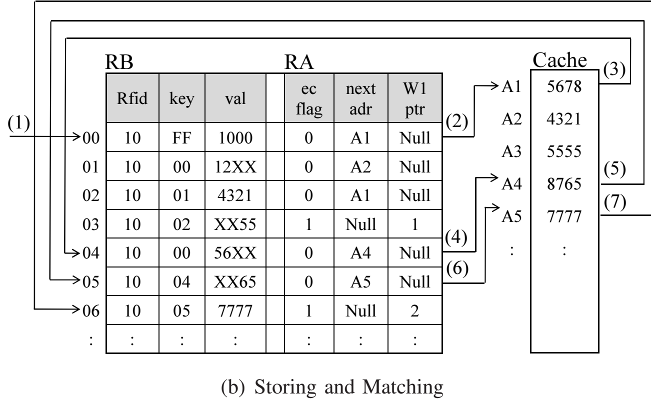
B. Input Entry and Input Matching

A series of inputs, or 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. A series of inputs of a memoizable region is represented as a way from the root to a leaf on this tree. Hence, the auto-memoization processor should hold input sequences as a tree structure.

Fig.3(a) shows an example of a tree structure of input sequences in the traditional model. Each node of the tree represents memory address which is read in the instruction region, and each edge represents the stored value in it. Here `End` represents the terminal of a sequence. In the input values, `X` in input values represents a *don't care* nibble in the cache



(a) Tree of input sequences in the traditional model



(b) Storing and Matching

Fig. 3. Input sequences and how they are registered.

line, and will not be tested. First, the values of registers are read as inputs, and now assume that values are `1000`. Next, the address `A1` should be accessed. In the input sequence (i), the input value in the first half of `A1` cache line is `12`, and `A2` will be read next. On the other hand, in the input sequence (ii), the value of `A1` is `56XX`, and `A4` will be read next.

Now, there are two nodes which represent same `A1` cache line in the input sequence (i). One of these nodes is associated to the input value which is read from the address of the first half of `A1`, and the other is of the latter half of `A1`. This is because that the input values construct a path in the input tree in the order of that they are read from registers or memories. On the other hand, the processor can merge multiple values which are continuously read from a common cache line into one edge. For example, the node `A2` in the input sequence (i) is associated with multiple values which are continuously read.

Fig.3(b) shows how the input sequences shown in Fig.3(a) are registered onto RB/RA. In this example, the index of the associated RF entry is assumed as 10, and it is stored in RFid fields. Fig.3(b) also shows an input matching flow on MemoTbl as (1)...(7). This flow is for searching the input sequences (ii) shown in 3(a). First, the auto-memoization processor reads the values of registers when the start address of the instruction region is detected. Then, the processor searches the root entry whose `key` is `FF`, and whose `val` matches to the values of the current registers. Now, the line `00` matches (1). Next, the value of `A1` is read because `next adr` of the line `00` in RA indicates `A1` (2). Then, the processor searches the entry whose `key` is `00`, and whose `val` matches to the value of `A1` (3). This process is applied repeatedly (4)...(7) until all input values have been compared. Each node in the input sequence tree is mapped onto a line in RB/RA as shown. Hence, search overhead is proportional to the number of nodes which construct the input sequence.

```
1 int func(int *a, int *b){
2   int i, c[3], d[2];
3   for(i=0; i<2; i++) {
4     c[i] = a[i];
5     d[i] = b[i];
6   }
7   c[i] = a[i];
8   return 0;
9 }
```

Fig. 4. A sample code.

C. Overhead Filter

For some reusable regions, these 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, the auto-memoization processor has a structure which estimates the effect of reuse, and avoids memoizing unsuitable instruction regions. With the execution cycles $step$ of the region, the processor calculates the performance gain in terms of omitted cycle as

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

where Ovh^R and Ovh^W represent search/write-back overheads for the region respectively. If the value is negative, applying memoization will decrease the performance, and the processor stops reusing the region.

III. Input Entry Integration

In this section, we will propose a new execution model which integrates multiple input entries into a single entry.

A. Entry Integration

As mentioned in the previous section, several input values which are continuously read from the same cache line can be registered in one entry. However, the values are registered separately into multiple entries when they are not read continuously even if they are on same cache line and the number of entries and the search cost for the region might increase.

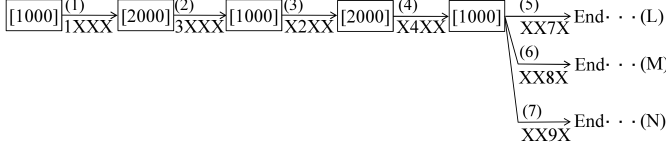
Therefore, we propose a new model which integrates input entries by sorting the nodes in input trees. This model can reduce the usage of MemoTbl entries and the search overhead.

Now, let us see how the new model can reduce the number of MemoTbl entries and the search cost with the sample code shown in Fig.4. This program is executed with three input sequences shown in TABLE I. Here, the array `a` is sequentially allocated from the address `1000`, and the array `b` is also sequentially allocated from `2000`. There are both a function and a loop in the sample code, but we will consider the inputs only for the function.

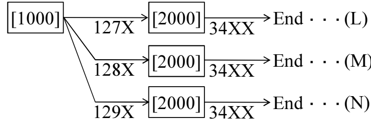
In the traditional model, input values are serialized in accessed order, and the input tree for the input sequences shown in TABLE I will be constructed as shown in Fig.5(a). When the processor executes the sample code with the input sequence `L`, it reads `a[0]` in the 4th line and `b[0]` in the 5th line. The processor registers these values as (1) and (2). Also

TABLE I
EXAMPLE INPUT SEQUENCES

	a[0]	a[1]	a[2]	b[0]	b[1]
Address	1000	1004	1008	2000	2004
Input Sequence L	1	2	7	3	4
Input Sequence M	1	2	8	3	4
Input Sequence N	1	2	9	3	4



(a) the traditional model.



(b) the new integration model.

Fig. 5. Example of input entry structure

in the next iteration, the processor reads a[1] and b[1], and it registers these values in the same way(3)(4). Then, the processor reads a[2] in the 7th line and registers this value(5). Consequently, each entry has only a single value in it. The values of the three input sequences are different each other only by the value of a[2]. Hence, the entries other than a[2] are merged and shared between the three input sequences in the input tree. On the other hand, the terminal entry for a[2] is not shared and each input sequence has its own leaf entry (5), (6) and (7).

In the new model proposed in this paper, the processor integrates multiple entries by sorting them by memory address. The resulting tree with the new model is shown in Fig.5(b). When the processor executes the sample program with the input sequence L, the values of a[0] and b[0] are registered in the same way as the traditional model. Next, the processor reads a[1]. Its cache line address is same as a[0]. So, the processor registers the value of a[1] into the entry which has the value of a[0], and the value of the address 1000 is changed from 1XXX to 12XX. The value of b[1] is also integrated into the 2000 entry which already has the value of b[0]. Then, the processor reads a[2] in the 7th line. Its cache line address is same as a[0] and a[1], and the value of a[2] is registered into the entry which already has the value of a[0] and a[1].

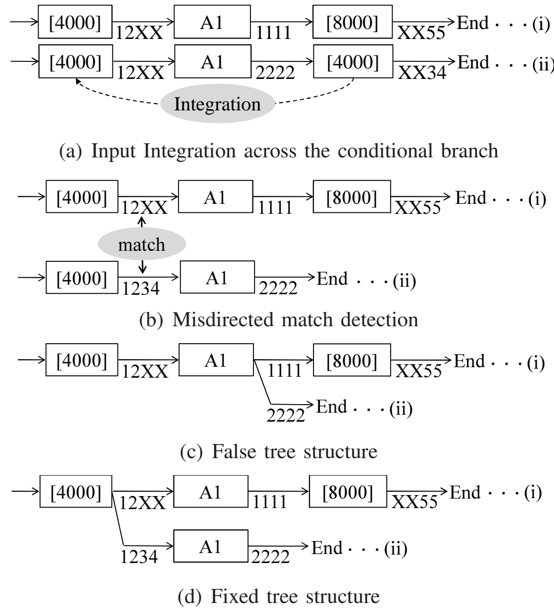
In this example, seven entries are used for storing the all input sequences in the traditional model. On the other hand, only six entries are needed for storing them in the new model, and the less MemoTbl entries are required. However, it might be more difficult to construct the input tree in the proposal model. This is because that an input which is accessed earlier has high possibility of being shared between multiple input sequences, and should be placed near the root of the input tree. In some cases, more entries may be required for storing all input sequences, because there will be less opportunities to share nodes between them than the traditional model. Assume that there are two input sequences other than the three shown in TABLE I, and the two also differs from the three only by the value of a[2]. In this case, nine entries are required with the traditional model, but ten entries with the new model. However, how many entries should be searched per input sequence has no relation to the entry sharing. For example, the processor should trace five entries for single call of the function shown in Fig.4 in the traditional model, but only two entries in the proposal model. Hence, the search overhead can be reduced by integrating the input entries under any circumstances.

B. Keeping Correct Structure of the Input Tree.

As mentioned above, the values are registered in the order that they are read in the traditional model. When registering new entry, the processor searches the entry for deciding whether the entry is already registered and can be shared. We call this *search-before-register*. However, if the processor integrates multiple entries without considering the accessed order of the input values in the proposal model, there will be several entries which have same values with different *don't care* masks. Such entries lead to misdirected detection through registering input values onto MemoTbl. The misdirected detection which may occur through the *search-before-register* might make false tree structure because the entries with different *don't care* masks are registered as a common entry. This false tree structure can cause false computation reuse, and false results of executions. Hence, it should be confirmed that all matches in *search-before-register* are not misdirected. This can be implemented by comparing the don't care masks.

Fig.6 shows how an entry integration causes misdirected detections. In this example, the values of A1 and their next input addresses are different between the input sequence (i) and (ii) in Fig.6(a). The address A1 might be of a variable which is used in a condition statement. In this case, the next address of A1 is same as the previous address of A1 in the input sequence (ii), but not in (i). The two entries in (ii) can be integrated in the new model, and now the tree is as shown in Fig.6(b). In these two input sequences, the values associated with 4000 cache line are 1234 in (i) and 12XX in (ii). Now, the value 1234 is implied in 12XX, and the traditional auto-memoization processor considers these two entries as same. After the integration, these two entries are merged and shared incorrectly in MemoTbl as shown in Fig.6(c). This false tree structure can cause false computation reuse because latter half values of A1 entry will be not tested.

For avoiding the incorrect integration, the new model compares the don't care masks of two entries, and integrates them only if the masks are same. If these masks are not same, the processor makes new route (ii) without sharing the entry as shown in Fig.6(d). Hence, the both entries with the values 12XX and 1234 will be registered in MemoTbl. Now, a search for 1234 will match both of these entries. Therefore, the multi-match will occur when the processor searches the value



(a) Input Integration across the conditional branch

(b) Misdirected match detection

(c) False tree structure

(d) Fixed tree structure

Fig. 6. Fixing of false tree structure.

1234. Now, we assume that the priority encoder of the ternary CAM for MemoTbl is designed to output the first match-line.

When a misdirected match occurs through input search for computation reuse, the processor can continue searching further input values. After then, the variable used in a condition statement will be compared before long, and its input matching will fail. Now, as mentioned above, when multi-match occurs, the first match-line will be selected. The computation reuse will be succeeded if the selection is appropriate and fail if it is not. Therefore, false computation reuse will not be applied even if multi-match is occurred through the input search for computation reuse. There are no additional overhead with the new model, because the search cost will be reduced by the input integration. However, the performance can be slightly lower, because the hit rate of MemoTbl decreases under the situation where an inappropriate entry is selected in multi-matches.

IV. Implementation

In this section, an implementation for the input integration model will be described. In the previous section, we have assumed that the all input/output values are on memory. However, register values also can be inputs/outputs of the memoizable regions. Hence, the auto-memoization processor registers not only the memory values but also the value of various registers as inputs/outputs.

As mentioned in II-A, MemoBuf is used as a write buffer for MemoTbl. The detailed structure of MemoBuf is shown in Fig.7. MemoBuf has a number of entries, and one entry corresponds to one input/output sequence. Each entry has a field for storing an index for RF (*RFid*), the stack pointer (*SP*) of when the memoizable region begins, the return value of the function or the terminal address of the loop (*FOfs*), input values (*Read*) and output values (*Write*). In the *Read* and

RFid	SP	FOfs	Read					Write				
			#1		…	#n		#1		…	#n	
			addr	val		addr	val	addr	val		addr	val

Fig. 7. Structure of Memobuf.

Write fields, entry has the kind of the registers or the cache line addresses (*addr*) and input/output values (*val*). Here, *val* can also hold *don't care* masks as well as in MemoTbl.

There is no need to register the value which is already registered in MemoBuf as an input or output. Now, assume that the processor detects a load instruction while executing a memoizable region. First, *Read* and *Write* fields are searched for finding whether the registering value has already been registered as an input or an output. The processor searches for an entry which has the same *addr* with the registering value. When such an entry is found, the processor compares the mask of its *val* and the mask of registering value. If they are same, it is found that the input value has already been registered, and the processor completes the search. Otherwise, the processor continues searching. Consequently, if such an entry is not found on MemoBuf, the processor makes new entry for the registering value.

The processor needs a mechanism for detecting some multiple values which can be registered in an entry, or be integrated. The traditional model has a similar mechanism for integrating multiple continuous read into single entry, as mentioned in II-B.

In the traditional model, whether the registering value can be merged into one of the entries on MemoBuf is decided by searching *Read* fields. The detail of this process is described below using Fig.8. In the *Read* fields, the values of argument ARG, cache line address 8000, global register GR, and cache line address 4000 are sequentially registered. If the processor reads the value X0XX from the address 8000 in this situation, the *Read* fields are searched from #1 shown as (A). Now, the entry for the same cache line is discovered in #2. The value is 1XXX and is not same with the registering input value X0XX, but they are on same cache line and the difference is represented by *don't care* masks. The addresses of these value are not same because the masks of these entries do not match. However, the entry #2 is not the terminal entry and the reads of the values X0XX and 1XXX are not continuous. So, the processor doesn't integrate these entries. Consequently, the input is registered in the empty entry #5.

On the other hand, in the new model, multiple entries for the same cache block can be integrated even if they are read separately. With the example shown in Fig.8, if the processor reads a value X0XX, the *Read* fields are sequentially searched in turn from #1, shown as (B). Then, the entry for the same cache block is discovered in #2, and the processor integrates value X0XX to #2. Consequently, the value 10XX is stored to

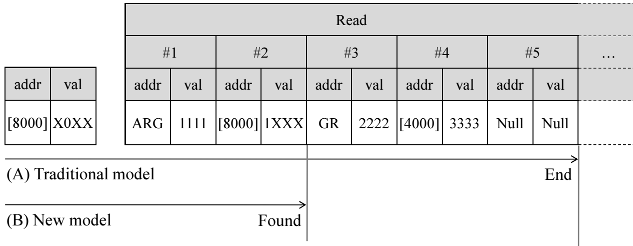


Fig. 8. Searching Read Field

TABLE II
SIMULATION PARAMETERS

MemoBuf	64 kBytes
MemoTbl CAM	128 kBytes
Comparison (register and CAM)	9 cycles/32Bytes
Comparison (Cache and CAM)	10 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

the entry #2.

As mentioned in III-B there is a possibility to apply false computation reuse by misdirected match. Then, in the proposal model, a logic for comparing the *don't care* masks is installed. This comparison is done as a part of the process for registering inputs onto MemoBuf, and this process can be overlapped by the program execution.

V. PERFORMANCE EVALUATION

A. Simulation Environment

We have developed a single-issue SPARC-V8 simulator with auto-memoization structures and three speculative cores. This section discusses the performance of the new integration model. The simulation parameters are shown in TABLE II. The cache structure and the instruction latencies are based on SPARC64-III[4]. The on-chip CAM for RB in MemoTbl is modeled on R8A20410BG[5]. The latencies of the CAM are defined on the assumption that the clock of the processor is 10-times faster than the CAM.

B. Average Number/Length of Entries

First, we have evaluated two statistics of input sequences. The one is *average entries*; the average of how many entries are used per input sequences. The other is *average length*; the average of how long each entry is.

The *average entries* is calculated by dividing the number of total entries by the number of registered input sequences. The *average length* is calculated by dividing the sum of length of total entries by the number of input sequences. These averages will differ from each other, because some entries are shared between input sequences.

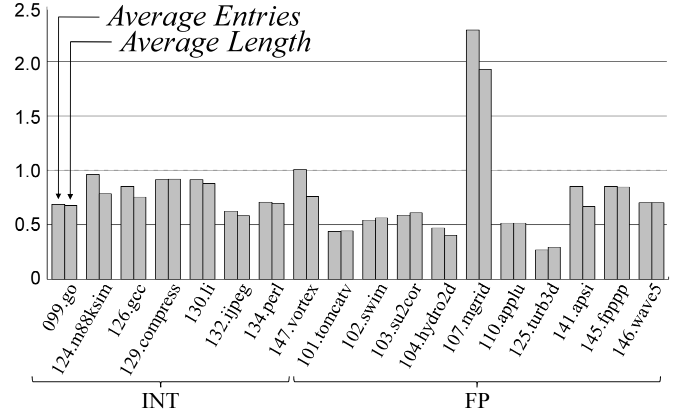


Fig. 9. *average entries* and *average length*.

Fig.9 shows the two averages of the new integration model with SPEC CPU 95 benchmark suite. Each workload is represented by two bars in this chart. The left bar plots *average entries* and the right bar plots *average length*. Each bar is normalized to the number of which in the traditional model.

Looking at the Fig.9, it is found that both averages are reduced drastically. Especially with 125.turb3d, *average entries* is reduced 73.7%, and *average length* is reduced 69.5%. However, these values with 107.mgrid are drastically increased. Through a detailed examination with 107.mgrid, it is found that the input sets for a certain instruction region which has numerous inputs are not registered in the traditional model, but registered in the new model. The reason is that the length of entries for the input sets was too long for storing onto MemoBuf in the traditional model, and the instruction region could not be memoized. However in the new model, the length is reduced by the input integration, and the region can be memoized. On the other hand, *average entries* with 147.vortex is slightly increased than the traditional model while the *average length* is reduced. This reason is that it becomes difficult to sharing the entries by applying the new integration model as described in III. The decrease of *average length* means that the search overhead for computation reuse will be reduced, and the increase of *average entries* means that less input sets can be stored on the MemoTbl and the hit rate of computation reuse may be lower.

C. Execution Cycles with SPEC CPU95

Next, we have evaluated the execution cycles. The evaluation results are shown in Fig.10. We have evaluated following three models, no-memoization model (N), traditional model (M) and input entry integration model (I). Fig.10 shows the normalized execution cycles of these models. Each bar is normalized to the number of executed cycles of (N).

The legend in Fig.10 shows the breakdown items of total cycles. They represent the executed instruction cycles ('exec'),

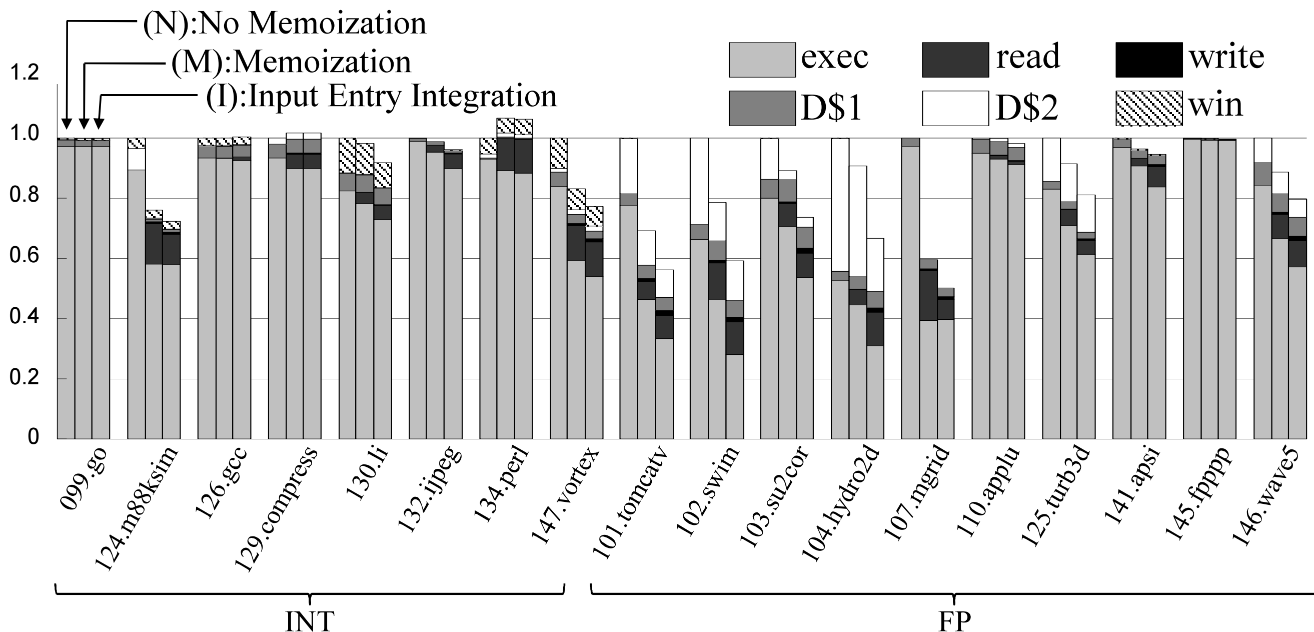


Fig. 10. Ratio of cycles (SPEC CPU95).

TABLE III
INFORMATION OF MEM_GETWORD.

	traditional model(M)	new model(I)
average entries	6.2	8.0
average length	10.0	8.0
number of stop reusing	1317472	961174
number of reusing	1992350	2062569

the comparison overhead between CAM and the registers or the caches ('read'), the write-back overhead ('write'), the D1 and shared D2 cache miss penalties ('D$1', 'D$2'), and the register window miss penalty ('win') respectively.

First, note that the new model (I) reduces the 'read' of 107.mgrid whose *average entries* is increased as shown in Fig.9. This is because the search overhead in some memoizable regions, which have high hit rate of computation reuse, decreases. However *average entries* is increased by registering an instruction region which has numerous inputs. On the other hand, 'exec' of 107.mgrid is increased with the new model (I). This reason is that, more instruction regions are registered onto MemoTbl, and instruction regions which are reused in the traditional model may be purged from MemoTbl away.

The new model (I) reduces both of 'read' and 'exec' of 147.vortex whose *average entries* is increased but *average length* is decreased. Detailed investigation proves that the function 'Mem_GetWord' is deeply involved with the result. The result is shown in TABLE.III. The overhead filter stops applying computation reuse to the function less frequently, and the reuse hit rate for the function raises with the new model.

Incidentally, the hardware cost and energy consumption of the auto-memoization processor is fairly larger than a multi-core processor, because a ternary CAM is required for its implementation. However, the auto-memoization processor can accelerates the programs which have little parallelism.

In conclusion, with most workloads, the reuse hit rate raises and the execution cycles are reduced with the new model. The new model (I) improves the maximum speedup ratio to 50.0%, and the average speedup ratio to 16.4%.

VI. CONCLUSIONS

In this paper, we have proposed input entry integration as a speed-up method for the traditional auto-memoization processor. A mechanism for keeping input tree structure correctly has also been proposed.

Through an evaluation with SPEC CPU95 suite benchmark programs, it is found that the new model brings the reduction of the number of reuse table entries and search overhead. The new model improves the maximum speedup ratio from 40.5% to 50.0%, and the average ratio from 10.5% to 16.4%.

One of our future works is the hybrid method which uses input entry integration and software assist. This method aims at the efficient registration of inputs/outputs by combining input entry integration with an existing speed-up technique such as loop unwinding.

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] P. Norvig, *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1992.
[4] *SPARC64-III User's Guide*, HAL Computer Systems/Fujitsu, May 1998.
[5] Reneses Electronics Corporation., *Product Overview: R8A20410BG*.