This is the accepted manuscript of a paper published in
Proc. 4th Int'l Symp. on Computing and Networking (CANDAR'16)
Copyright (C) 2016 IEEE

Exploiting Bloom Filters for Saving Power Consumption of Auto-Memoization Processor

Masayoshi FUJII*, Yuuki SATO*, Tomoaki TSUMURA* and Yasuhiko NAKASHIMA†
*Nagoya Institute of Technology, Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp
†Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara, Japan
Email: nakashim@is.naist.jp

***Abstract*—We have proposed a processor which can exploit value locality in programs by automatically applying computation reuse. The processor which we call auto-memoization processor dynamically detects functions and loop iterations as reusable blocks, and stores their input sequences and results into a lookup table. When the current input sequence matches one of the stored input sequences on the table, the stored result associated with the matched input sequence is written back to the registers and caches. In the previous implementation, a part of the table is implemented with a CAM for achieving associative search for input matching with small overhead. However, CAMs consume considerably large energy, area and manufacturing cost. Therefore, CAM size should be as small as possible for improving practicality of the auto-memoization processor. In this paper, we propose a low-power implementation of the auto-memoization processor by utilizing a RAM and a Bloom filter. The result of the simulation experiment shows that power consumption of the table is reduced by 67.5% at a maximum and by 50.4% on average.**

I. Introduction

So far, various speed-up techniques exploiting some parallelisms in some granularities for microprocessors have been proposed. These techniques aim to improve the performance of microprocessors by executing some several sized tasks simultaneously. However, many programs have a few distinct parallelisms, and it is very difficult to extract latent parallelisms. Therefore, effectiveness of these techniques has proved to be limited.

Meanwhile, we have proposed a processor called the *auto-memoization processor*[1] which exploits value locality by utilizing computation reuse. In contrast to techniques exploiting some parallelisms, the auto-memoization processor aims to improve the performance by avoiding recomputation. The auto-memoization processor automatically and dynamically detects functions and loop iterations as reusable blocks, and stores their input sequences and results into a lookup table. When the current input sequence matches one of the past input sequences, the auto-memoization processor reuses the result stored on the table to avoid recomputation. For reducing overheads for searching the table, fast associative search is required. Therefore, the table is implemented with a *CAM (Content Addressable Memory)* for achieving associative search for input matching with small overhead. However, CAMs consume considerably large energy, area and manufacturing cost. Therefore, CAM size should be as small as possible for improving practicality of the auto-memoization processor. In this paper, we propose a low-power implementation of the auto-memoization processor by utilizing a RAM and a *Bloom filter*[2] without performance degradation.

II. 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*[3] to apply computation reuse to computation blocks in programs. We have proposed an auto-memoization processor which dynamically detects functions and loop iterations as reusable blocks, and memoizes them automatically on the processor.

The auto-memoization processor is equipped with the memoization engine, *MemoTbl* and *MemoBuf* in addition to components of a common processor. MemoTbl is a set of tables for storing input/output sequences of past executed computation blocks. MemoBuf works as a write buffer for MemoTbl. Entering to a memoizable block, the processor refers to MemoTbl and compares the current input sequence with the past input sequences which are stored in MemoTbl. In the following, we call this comparison '*reuse test*.' If the current input sequence matches one of the stored input sequences in MemoTbl, the memoization engine writes back the stored outputs, which are associated with the matched input sequence, to the registers and caches. This omits the execution of the block and reduces total execution time. On the other hand, if the current input sequence does not match any past input sequences, the processor stores the current inputs and outputs of the block into MemoBuf while executing the block as usual. Reaching the end of the block, the memoization engine stores the content of MemoBuf into MemoTbl for future reuse.

The structure of MemoTbl is shown in Fig. 1. MemoTbl consists of four tables:

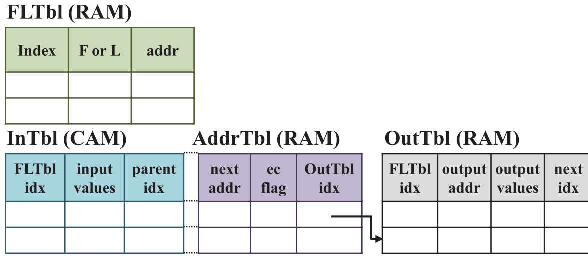

Fig. 1. Structure of MemoTbl.

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

FLTbl, AddrTbl, and OutTbl are implemented with RAMs. On the other hand, InTbl is implemented with a ternary CAM, so that input data can be found fast by associative search. In the following, the structure of each table in MemoTbl are described.

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

Each InTbl entry has an index for FLTbl (*FLTbl idx*), which represents the associated computation block of the input stored in the entry, and also holds input values (*input values*). Since each InTbl entry can hold a single cache line, an input sequence over multiple cache lines is stored onto InTbl by using several entries. When a variable is read as an input value, its whole cache line is stored into an InTbl entry, masking the other parts than the variable in the cache line with "don't care" values of ternary CAM. Each InTbl entry also holds an index key for its parent entry (*parent idx*) because input sequences are managed as a tree structure as mentioned later.

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*), and a flag (*ec flag*) which shows whether it is the terminal entry of an input sequence or not. Each AddrTbl entry also has a pointer (*OutTbl idx*), which refers to an OutTbl entry for associated outputs, when the input matching succeeds.

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

Now, an input sequence for a certain computation block can be represented as a sequence of tuples, each of which contains an address and a value. In a certain computation block, 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. Accordingly, the universal set of the different input sequences for an computation block can be represented as

```
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 = 5; calc(2);   /* x = 2, a = 5, c = 8 */
15   a = 3; calc(2);   /* x = 2, a = 3, b = 5 */
16   return(0);
17 }
```

Fig. 2. A sample code.

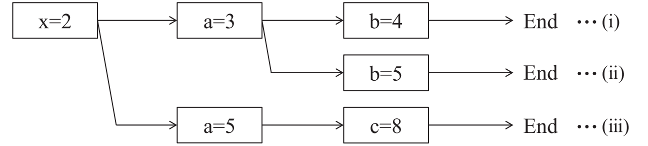

Fig. 3. Tree of input sequences.

a multiway input tree, and the auto-memoization processor holds input sequences as a tree structure. 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 an input value, and each edge represents the address which should be referred next, and `End` represents the terminal of a sequence. Each input sequence (i), (ii) and (iii) corresponds to the function call at line 12, 13 and 14, respectively, in Fig. 2. In the input sequences (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 the value of variable `a` which is read as the second input differs from the past input value, and the result of the branch instruction at line 5 changes.

III. A Low-Power Implementation

In this section, we describe the problem of the auto-memoization processor and the overview of our proposal to address this problem.

A. Problems of the Auto-Memoization Processor

As mentioned in section II, because fast associative search is required for input matching, InTbl of the auto-memoization processor is implemented with a CAM. Although CAMs provide fast associative search, they consume considerably large energy, area and manufacturing cost. Therefore, on general-purpose processors, CAMs are used for only certain units, for example, TLB and LSQ, and besides, their size is about 16KBytes at most.

In contrast, the previous auto-memoization processor is supposed to be implemented with a 128KBytes CAM, and it is not practical to be implemented on a chip. Therefore,

for improving practicality of the auto-memoization processor, CAM size should be reduced.

B. Low-Power Implementation of MemoTbl

To reduce required CAM size, we propose a low-power implementation of the auto-memoization processor with negligibly small performance degradation.

1) Reducing CAM Size: At first, we consider that MemoTbl which is searched associatively should be implemented with a RAM instead of a CAM. However, each entry should be sequentially accessed for associative search on RAMs. If InTbl is implemented with only a RAM instead of a CAM, the lookup overhead will considerably increase. Accordingly, increasing the lookup overhead may cancel the benefit of computation reuse, or will even degrade the total performance.

Several existing techniques utilize a filter for reducing the energy consumption or CAM size [4], [5], [6]. Thus, in order to reduce reuse test overheads, we utilize not only a RAM but also a filter which can identify whether an entry is stored or not on a table without searching the table. In addition, because the auto-memoization processor holds input sequences as a tree structure, we design an entry management suitable for the tree structure, in order to reduce reuse test overheads.

2) Reducing Overheads with a Filter: In reuse test, the previous auto-memoization processor can compare an input data set with all stored input data sets on InTbl simultaneously, because InTbl is implemented with a CAM. Therefore, the lookup overhead which costs per input data set in reuse test is constant, and the overhead is independent of the number of registered entries or whether the current input data set matches any of the registered entries or not. On the other hand, if InTbl is implemented with a RAM, each entry needs to be accessed and checked sequentially. Therefore, lookup overhead raises proportionately to the number of entries which are accessed until the matching entry is found. Additionally, when failing in reuse test, namely, no matching entry exists on InTbl, lookup overhead becomes very large due to input matching for all registered entries on InTbl.

We propose a solution for these problems by utilizing a *Bloom filter*[2]. Bloom filters can test whether an element is registered in a database or not without searching the database. When using a Bloom filter for our proposal, whether an input data set is registered on InTbl or not can be tested before searching InTbl. If the result of the test is negative, the input data set proves to be not registered on InTbl. In consequence, the auto-memoization processor can judge that reuse test has failed without searching InTbl. Therefore, only the calculation overhead for checking the Bloom filter costs when there is no matching entry on InTbl.

3) Reducing Overheads Considering Tree Structure: As mentioned in the section above, lookup overhead when failing in reuse test can be reduced by utilizing a Bloom filter. On the other hand, if an entry which matches the current input data set is registered on InTbl, the test by the Bloom filter results in positive, and entries on InTbl must be accessed sequentially after all, for the matched entry being found.

To address this problem, we focus on that the auto-memoization processor holds input sequences as a tree structure. Assume that, an entry which matches the current input data set is detected during a reuse test for an reusable block. An entry which matches the next input data set of the reusable block is one of the child nodes of the detected entry in an input tree structure. In other words, lookup overhead on RAM-based InTbl can be minimized by considering parent-child relations of an input tree structure. Therefore, the lookup overhead will be reduced.

In addition, we adopt a set-associative strategy for the RAM-based InTbl for testing multiple child nodes at once. Thereby, the number of accesses to the RAM can be reduced.

IV. Reducing Overheads by Utilizing Filters

In this section, we describe how to reduce the lookup overhead with filters when reuse test fails. First, we describe the outline of a Bloom filter. After that, we describe the outline of Parallel Counting Bloom filter and how to utilize the filter in our proposal.

A. Bloom Filter

A *Bloom filter* is a data structure which can be used to test whether an element is registered in a database or not. A Bloom filter consists of a bit array and some hash functions. For a Bloom filter, following two operations are prepared;

addition: to add a new element to a Bloom filter.

query: to test whether an element is stored in a Bloom filter.

In order to add an element, hash values for the element are calculated with hash functions, and all bits correspond to the hash values are set. On the other hand, in order to query for an element, its hash values are calculated with hash functions, and all bits correspond to the hash values are checked. If all of the bits have not been set, it is judged that the element is not registered. On the other hand, when all the bits correspond to the hash values have been set, it is judged that the element is registered in the set.

However, a Bloom filter can give false-positive matches because of hash collisions. Therefore, after a *query* for an element is judged as positive, it must be confirmed whether the element really exists in the database or not. On the other hand, when a *query* for an element is judged as negative, it is guaranteed that the element does not exist in the database, because Bloom filter does not give false-negatives.

When utilizing a Bloom filter for reuse test in the auto-memoization processor, there are two points at issue. The first point is that a Bloom filter does not support deletion of registered elements. The size of MemoTbl is finite, and deletion of elements must be supported.

The second point is the overhead which is caused by operations for a Bloom filter. A bit array as a Bloom filter needs to be accessed at every operation *addition* and *query*. In case when the array is implemented with a one-port RAM, access overhead rises proportionately to the number of hash functions, and it can considerably degrade the total performance of the auto-memoization processor. On the other hand, in case when

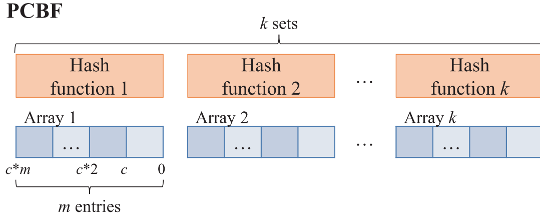

Fig. 4. Structure of a PCBF.

the array is implemented with a multi-port RAM, the area cost will be a concern. Thus, for a low-power implementation of the auto-memoization processor, a filter which does not have such problems is required.

B. Parallel Counting Bloom Filter

We adopt a *Parallel Counting Bloom Filter (PCBF)* which is an expanded variant of Bloom filters, for the low-power implementation. The structure of a PCBF is shown in Fig. 4. A PCBF is different from an original Bloom filter in two points. The first point is that a PCBF consists of as many arrays as hash functions, while a Bloom filter consists of only one array. The second point is that each array element is a counter, while each element of the array for Bloom filter is a 1-bit cell. Therefore, when the number of hash functions is k, *addition* on a PCBF is achieved by incrementing a corresponding element to the calculated hash value for each of k arrays, instead of setting k bits in an array. Meanwhile, *query* on a PCBF is achieved by checking the value of the element corresponding to the calculated hash value is non-zero for each of k arrays. Because each array element can hold the number of stored elements which have the hash value corresponding to the element, *deletion* can be achieved on a PCBF by decrementing an array element for each of k arrays, unless and until array elements do not overflow.

In addition, k arrays of a PCBF can be independently accessed in parallel because each array corresponds to one of the k hash functions, and no operations on PCBF cause access conflicts on the arrays. Therefore, operating overhead on a PCBF is smaller than that on a Bloom filter. We utilize a PCBF for a low-power implementation of the auto-memoization processor. As explained in section III-B1, when InTbl is implemented with a RAM, reuse test overhead will increase. Thus, by querying a PCBF before searching InTbl, the overhead of when reuse test fails can be reduced.

We utilize a PCBF with six hash functions based on the calculation defined in the paper[6]. Each array corresponding to each hash function has 1366 elements, and each element is an 8-bit counter. The codomain of each hash function ranges from 0 to 1365.

V. Optimized Search for Input Tree Structure

In this section, we describe search optimization for an input tree structure with considering parent-child relation as mentioned in section III-B3, for reducing the lookup overhead on RAM-based InTbl when reuse test succeeds.

```
1  int a = 3, b = 0;
2  int calc(x){
3    int tmp = x + 1;
4    tmp = tmp + a;
5    tmp = tmp + b;
6    return(tmp);
7  }
8  int main(void){
9    for(i=0;i<5;i++) {
10     b = b + i;
11     calc(2);
12   }
13   return(0);
14 }
```

Fig. 5. A loop that includes a function call.

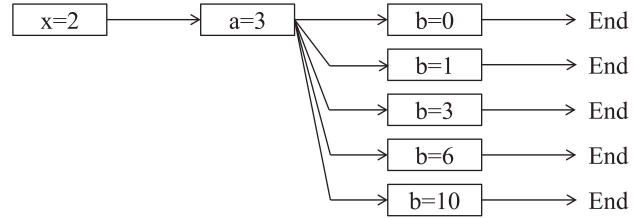

Fig. 6. Input tree of the function calc.

A. Characteristics of Input Trees

Some reusable blocks have a certain characteristic regarding their input sequences. The characteristic is that many input sequences branch off at an input, or an input node has many children nodes in the input tree. For example, when the auto-memoization processor executes the sample program shown in Fig. 5, the tree structure of input sequences for the function `calc` will be formed as shown in Fig. 6. In this program, only the value of `b` changes every time while the loop is executed. Therefore, entries which correspond to the variable `b` are created. The number of created entries are five, which is equal to the number of loop iterations. On the previous CAM-based InTbl, these five entries can be searched for simultaneously. However, on the RAM-based InTbl, these five entries have to be tested sequentially, and lookup overhead for the input variable `b` will be a concern.

B. Structure of MemoTbl

For utilizing parent-child relation in the input trees, we propose a RAM-based InTbl which consists of three units; a small CAM, a direct map RAM (1-way RAM), and an n-way set associative RAM (n-way RAM). Here, nodes in an input tree are classified into three types; root nodes, nodes which have sibling nodes, and nodes which have no sibling node. When a new input data set is stored on RAM-based InTbl, one of these units is selected considering the characteristics of these three types of nodes. Firstly, a root node should be associatively searched for, because it has no parent node and parent-child relation can not be utilized when it is searched for. Therefore, a small CAM is used for root nodes, and the nodes can be searched for rapidly. Secondly, we consider non-root nodes which have sibling nodes. When a node matches the

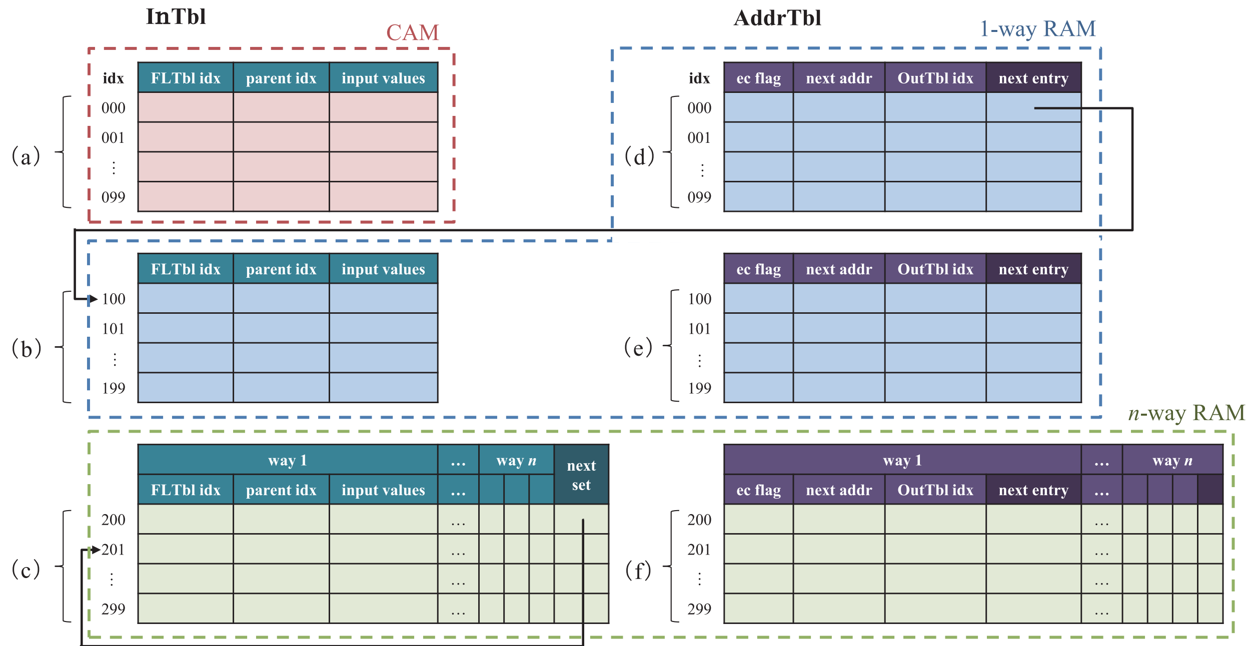

Fig. 7. Structure of InTbl and AddrTbl.

current input data set, one of the child nodes of the matched node can match the next input data set. Hence, the matched node has several children, or its children have siblings, the number of required accesses for the next input matching can be the same as the number of the sibling nodes at a maximum. Thus, for reducing the number of required accesses, an n-way RAM which can read n entries at once is used and the sibling nodes are stored in entries in a line of the n-way RAM. Finally, we consider non-root nodes which have no sibling node. If a matched node has only one child, only the child node can match the next input data set, and the number of required accesses for the next input matching is only one. For this reason, a 1-way RAM is suitable for storing such a node.

Accordingly, we modified the structure of InTbl and AddrTbl. The modified structure is shown in Fig. 7. InTbl consists of (a) a small CAM, (b) a 1-way RAM and (c) an n-way RAM, and the fields of each entry are the same as the fields of the previous implementation of InTbl mentioned in section II. Besides, each line of (c) the n-way RAM has n entries. In addition, each line of the n-way RAM has the index (*next set*) for chaining the lines when the number of sibling nodes is larger then n.

Each entry on AddrTbl should correspond to each entry on InTbl, so that the structure of AddrTbl also should be modified as InTbl. Therefore, AddrTbl consists of (d) a 1-way RAM corresponding to (a), (e) a 1-way RAM corresponding to (b), and (f) an n-way RAM corresponding to (c). The fields of AddrTbl in each entry are same as the fields of the previous implementation of AddrTbl mentioned in section II. In addition, we installed the index (*next entry*) in AddrTbl for indicating the line which should be tested next.

C. *How to Store Input Data into RAM-based InTbl*

We explain how to store input sequences shown in Fig. 6. At first, the variable x is stored into the small CAM because this input is the root node on the input tree shown in Fig. 6. Root nodes have no parent node, and parent-child relation can not be used for searching them. Therefore, for avoiding performance degradation, root nodes should be stored on the CAM.

On the other hand, either the 1-way RAM or the n-way RAM is used for storing nodes except for roots. Which RAM to be used is decided by considering whether the node has a sibling or not. For example, the node "x = 2" has only one child "a = 3", and the node "a = 3" has no sibling. Such nodes which have no sibling node are stored on the 1-way RAM. Therefore, when a matched node has only one child node, the next entry is stored on the 1-way RAM or is not stored on any table. Hence, if the child of the matched entry exists in the 1-way RAM, only the entry needs to be tested, and the reuse test overhead costs for the only one entry.

When the entry corresponding to the node "a = 3" is found on the 1-way RAM, five entries for variable b should be tested next. Such nodes which have sibling nodes are stored on the same line of the n-way RAM, and can be tested simultaneously. Accordingly, the number of accesses to RAM can be reduced even if many entries should be searched next.

VI. Evaluation

We have evaluated the performance and energy consumption of newly designed MemoTbl, and compared with the previous implementation.

A. *Simulation Environment*

We have implemented the new structure of MemoTbl consists of a CAM, a 1-way RAM, an n-way RAM, and a PCBF

TABLE I
SIMULATION PARAMETERS

Parameter	Value
Processor	
architecture	SPARC-V8
#cores	4 cores
issue width	single
issue order	in-order
non-memory IPC	1
MemoBuf	
size	64 KBytes
MemoTbl	
FLTbl size	8 KBytes
InTbl/AddrTbl/OutTbl size	128 KBytes each
Comparison latency	
reg. ⇔ CAM (128/8/4 KBytes)	9/3/3 cycles / 32Bytes
Cache ⇔ CAM (128/8/4 KBytes)	10/4/4 cycles / 32Bytes
reg. ⇔ RAM	2 cycles / 32Bytes
Cache ⇔ RAM	3 cycles / 32Bytes
Write back latency	
MemoTbl ⇒ reg.	1 cycle / 32Bytes
MemoTbl ⇒ Cache	2 cycles / 32Bytes
PCBF	
hash function	6 pieces
array size	1366 Bytes
counter	8 bit
false positive rate	1.00 %
latency	2 cycles
D1 cache	
size	32 KBytes (4 ways)
access latency	2 cycles
miss penalty	10 cycles
D2 cache	
size	2 MBytes (4 ways)
access latency	10 cycles
miss penalty	100 cycles
Register windows	4 sets
miss penalty	20 cycles / set

on an in-house simulator of auto-memoization processor. The configuration of the simulator is shown in TABLE I.

We have measured the energy consumption of MemoTbl by using Wattch[7] which is an architecture-level power analysis framework. The platform architecture shown in TABLE I is based on SPARC64-Ⅲ[8]. The large CAM for the previous implementation is modeled on DC18288[9] (32Bytes × 4K lines). On the other hand, the small on-chip CAM for a low energy implementation is modeled on eFlexCAM[10] (32Bytes × 256 lines). The latencies of the large CAM are defined on the assumption that the clock frequency of the processor is 10-times higher than the CAM, and the latencies of the small CAM are defined on the assumption that the clock frequency of the processor is 4-times higher than the small CAM.

We define the latency of *query* with the PCBF as two cycles, and we have estimated the overhead of the PCBF under the assumption that the false-positive rate of the PCBF is constant. The PCBF consists of six hash functions and six arrays which have 1366 elements. Hence, we define the false-positive rate of PCBF as 1.00% based on the calculation defined in the paper[6].

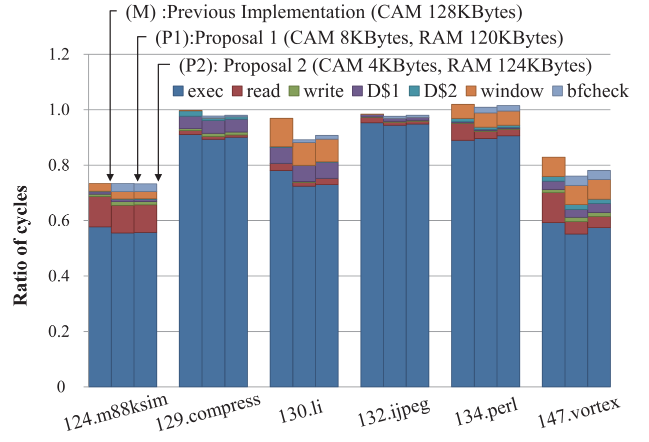

Fig. 8. Execution Cycles.

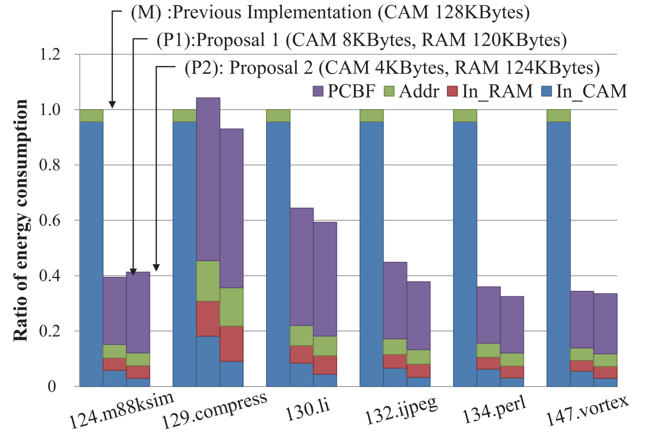

Fig. 9. Energy Consumption of MemoTbl.

B. Execution Cycles and Energy Consumption

We have evaluated a low-power implementation of the auto-memoization processor by executing benchmark programs in SPEC CPU95 suites with 'train' dataset. We have evaluated following three configurations;

- (M) Previous implementation (CAM 128KBytes)
- (P1) Proposal 1: A low-power implementation (CAM 8KBytes, RAM 120KBytes)
- (P2) Proposal 2: A low-power implementation (CAM 4KBytes, RAM 124KBytes)

Figure 8 shows the execution cycles of these processors, and each bar in Fig. 8 is normalized to the number of executed cycles by the processor without memoization mechanisms. The legend in Fig. 8 shows the breakdown items of total cycles. They represent the executed instruction cycles ('**exec**'), the comparison overhead between the 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**'), the register window miss penalty ('**window**'), and the overhead for the PCBF ('**bfcheck**'), respectively.

Figure 9 shows the energy consumption of InTbl, AddrTbl, and the PCBF with above three configurations, and each bar in Fig. 9 is normalized to the energy consumption of (M). The

legend in Fig. 9 shows the breakdown items of total energy consumption. They represent the energy consumption of the PCBF ('**PCBF**'), the energy consumption of the RAM which is for AddrTbl ('**Addr**'), the energy consumption of the RAM which is for InTbl ('**In_RAM**'), and the energy consumption of the CAM which is for InTbl ('**In_CAM**'), respectively.

As shown in Fig. 8, compared with (M), the total execution cycles of (P1) is reduced by 2.7% on average and by 8.3% at a maximum, and the total execution cycles of (P2) is reduced by 2.5% on average and by 6.4% at a maximum. Although we expected that the performance of the auto-memoization processor will decrease, the result shows that performance is rather improved. In addition, as shown in Fig. 9, compared with (M), the energy consumption is reduced by 46.1% on average and by 65.6% at a maximum with (P1), and is reduced by 50.4% on average and by 67.5% at a maximum with (P2).

C. Examination

Execution Cycles: First, with 129.compress, 130.li, 132.ijpeg and 147.vortex, the performance of a low power implementation of the auto-memoization processor is higher than (M) the previous auto-memoization processor, as shown in Fig. 8. This is because that 'read' the overhead for reuse test is reduced on the new implementation, and much more reusable blocks can gain performance than on the previous implementation. There are two possible reasons for why 'read' is reduced. One is that access latencies of a CAM depend on the size of the CAM. On the implementation proposed in this paper, CAM size can be smaller than on the previous implementation, and the latency per access can be smaller than the large CAM which the previous implementation uses. This reduces the total lookup overhead for CAM. The other is that the latencies of RAMs are lower than the latencies of CAMs. On the previous implementation, because InTbl is implemented with only a CAM, all entries can be tested at once. Although, latency per access is rather high. In contrast, on the proposed implementation with RAMs, latency per access is very small although multiple entries should be accessed sequentially for reuse test. Therefore, when the number of required accesses until a matching entry being found is small, the total overhead can be smaller than using only a CAM. Consequently, the performance is improved.

However, with 124.m88ksim, the performance gain of the new implementation is canceled out by 'bfcheck.' For a program as 124.m88ksim which originally has large performance gain with computation reuse, the effect on the reduction of 'read' by PCBF is limited, and the 'bfcheck' overhead appears to be relatively large. However, contrary to expectations, total performance is not degraded with all programs by using RAM-based InTbl, and is rather improved with some programs.

Energy Consumption: As shown in Fig. 9, with all benchmark programs except 129.compress, the energy consumption of the new implementation is considerably lower than that of previous implementation. The energy consumption of the small CAM per access is smaller than that of the large CAM which is used in the previous implementation. In addition, the number of accesses to the CAM is reduced because the CAM is accessed only when root nodes are searched for. Besides, the energy consumption of 128KBytes RAM per an access is about 20 times less than that of 128KBytes CAM per an access. Thus, the reduction of the energy consumption on the CAM more than made up for the additional energy consumption on the RAMs and the PCBF. Accordingly, the total energy consumption is reduced.

However, with 129.compress, the energy consumption of (P1) is slightly higher than that of (M). In this program, the number of total execution cycles is small, and the benefit of computation reuse appears to be little. With a program such as 129.compress, the number of accesses to MemoTbl during reuse test is small, and the energy consumption for accessing MemoTbl is also small. As a result, the increase of energy consumption by PCBF appears relatively high.

VII. Conclusion

In this paper, for improving practicality of the auto-memoization processor, we proposed a low power implementation of the auto-memoization processor by utilizing RAMs and a Bloom filter. By utilizing a Bloom filter, the size of CAM can be smaller than the previous implementation without performance degradation. Through an evaluation with SPEC CPU95 benchmark suite programs, although we expected that performance of the auto-memoization processor decreases, the total execution cycles are reduced by 2.5% on average and by 6.4% at a maximum. Additionally, the energy consumption of the lookup table for computation reuse is reduced by 50.4% on average and by 67.5% at a maximum. One of our future work is to design a structure of a Bloom filter which is suitable for the auto-memoization processor.

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] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

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

[4] F. Castro, D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang, "Load-store queue management: An energy-efficient design based on a state-filtering mechanism," in *2005 International Conference on Computer Design*. IEEE, 2005, pp. 617–624.

[5] F. Castro, L. Pinuel, D. Chaver, M. Prieto, M. Huang, and F. Tirado, "Dmdc: delayed memory dependence checking through age-based filtering," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 297–308.

[6] N. Kurata, R. Shioya, M. Goshima, and S. Sakai, "Address order violation detection with parallel counting bloom filters," *IEICE Transactions on Electronics*, vol. 98, no. 7, pp. 580–593, 2015.

[7] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. 27th Annual Intl. Symp. on Computer Architecture*, Jun. 2000, pp. 83–94.

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

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

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