An Approximate Computing Stack based on Computation Reuse

Yuuki SATO[*], Takanori TSUMURA[*], 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*—Approximate computing has been studied widely in computing systems ranging from hardware to software. Approximate computing is a paradigm for reducing execution time and power consumption by tolerating some quality loss in computed results. On the other hand, we have proposed a processor called auto-memoization processor which is based on computation reuse. The processor dynamically detects functions as reusable blocks, and automatically stores their inputs and outputs into a lookup table. Then, when the processor detects the same block, the processor compares the current input sequence with past input sequences stored in the table. If the current input sequence matches one of the input sequences in the lookup table, the processor writes back the associated outputs, and skips the execution of the function. Here, by tolerating partial input mismatch in computation reuse, approximate computing can be achieved. In this paper, we propose an approximate computing stack based on computation reuse. The stack includes a programming framework which allows programmers to easily apply approximate computing to various applications, a compiler, and the modified auto-memoization processor. Through an evaluation with cjpeg from MediaBench, by tolerating partial input mismatch in computation reuse, execution cycles are reduced by 22.3% in maximum, and reuse rate is improved by 29.5% in maximum with negligible quality deterioration in outputs.**

I. Introduction

Approximate computing is a paradigm for reducing execution time and power consumption by tolerating some quality loss in computed results, and it has been studied widely ranging from programming languages[1] to hardware implementations[2]. Approximate computing is a promising paradigm, but there has been no portable framework for applying approximate computing to various applications and programs yet.

On the other hand, we have proposed a processor called *auto-memoization processor*[3] which is based on computation reuse. The processor dynamically detects functions as reusable blocks, and automatically stores their inputs and outputs into a lookup table. When executing a function, the processor tests whether the current input sequence matches one of the stored input sequences on the table. If matches, the processor writes back the associated outputs to the registers and the caches, and skips the execution of the function.

Of course, to reuse the previous results, the current inputs must completely match the previous inputs. However, reusing the previous results when the current inputs for a function is not same as but close to the previous inputs can be an implementation of approximate computing. In other words, computation reuse which tolerates partial mismatch of inputs can achieve approximate computing. Thus, approximate computing and computation reuse are well suited to each other. In this paper, we propose an approximate computing stack based on computation reuse. The approximate computing stack consists of a programming framework, a compiler, and a modified auto-memoization processor. The programming framework provides a pragma which allows programmers to easily apply approximate computing to various applications, and the compiler encodes the approximation parameters directed with the pragma and tell them to the auto-memoization processor.

II. Base Architecture

We adopted the auto-memoization processor as a platform of the approximate computing stack. In this section, we describe the auto-memoization processor and its behavior.

A. The auto-memoization processor

Computation Reuse is a well-known speed-up technique in the software field. It is storing the input sequences and the results of some computation blocks, such as functions, for later reuse and avoiding recomputing them when the current input sequence matches one of the past input sequences. It is called *memoization*[4] to apply computation reuse to computation blocks in programs. Memoization is originally a programming technique for speed-up, and brings good results on expensive functions[5]. However, it requires rewrite of target programs, and the traditional load-modules or binaries can not benefit from memoization. Furthermore, the effectiveness of memoization is influenced much by programming styles. Rewriting programs using memoization occasionally makes the programs slower. This is because software implemented memoization costs considerable overheads.

On the other hand, the auto-memoization processor, which we have proposed, can execute traditional load-modules faster with low overheads and without rewriting a binary program. The auto-memoization processor dynamically detects functions as reusable blocks, and memoizes them automatically. A computation block between an instruction with a callee

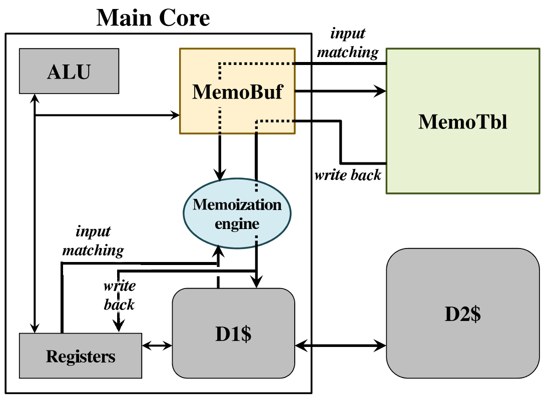


Fig. 1. Structure of Auto-Memoization Processor.

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

Fig. 2. A sample code.

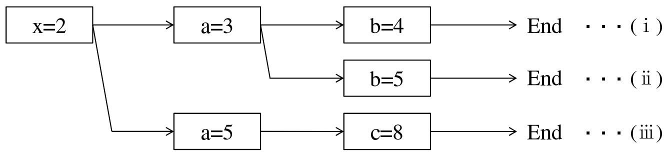


Fig. 3. Tree of input sequences.

label and a return instruction is detected as a memoizable function. 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. 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. If the current input sequence matches one of the stored input sequences on MemoTbl, the memoization engine writes back the stored outputs, which are associated with the matched input sequence, to the registers and the caches. This omits execution of the computation block and reduces total execution time.

If the current input sequence does not match any past input sequences, the processor stores the inputs and the outputs of the computation block into MemoBuf while executing the computation block as usual. The input sequence consists of the register/memory values which are read over the computation block, the output sequence consists of the values which are written, and its return value is also included in the output sequence. Reaching the end of the computation block, the memoization engine stores the content of MemoBuf into MemoTbl for future reuse.

MemoBuf has multiple entries, and each MemoBuf entry corresponds to one input/output sequence. Each MemoBuf entry has a stack pointer (*SP*), a return address (*retOfs*), an input sequence (*Read*), and an output sequence (*Write*). Read field has addr/reg field which holds input addresses or register IDs, and has value field which holds input values. Similarly, Write field has addr/reg field which holds output addresses or register IDs, and has value field which holds output values.

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. The result of the branch instruction varies according to an input value. Therefore, the universal set of the different input sequences for an computation block can be represented as 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 10, 11 and 12 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.

Next, let us see the structure of MemoTbl shown in Fig.4. MemoTbl consists of four tables:

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 RAM. On the other hand, InTbl is implemented with a ternary CAM (Content Addressable Memory), for small overhead of input matching by associative search. In the following, components of each table in MemoTbl are described.

Each FLTbl entry corresponds to a reusable computation block. Each FLTbl entry has the fields which hold 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 each InTbl entry also holds input values (*input*

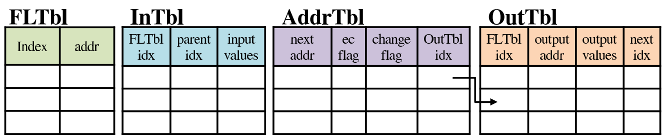


Fig. 4. Structure of MemoTbl.

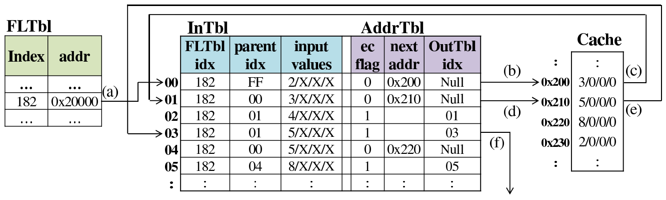

FLTbl
Index | addr
... | ...
182 | 0x20000
... | ...
(a)
InTbl
AddrTbl
FLTbl idx | parent idx | input values | ec flag | next addr | OutTbl idx
00 | 182 | FF | 2/X/X/X | 0 | 0x200 | Null
01 | 182 | 00 | 3/X/X/X | 0 | 0x210 | Null
02 | 182 | 01 | 4/X/X/X | 1 | | 01
03 | 182 | 01 | 5/X/X/X | 1 | | 03
04 | 182 | 00 | 5/X/X/X | 0 | 0x220 | Null
05 | 182 | 04 | 8/X/X/X | 1 | | 05
(b)
(c)
(d)
(e)
(f)
Cache
0x200 3/0/0/0
0x210 5/0/0/0
0x220 8/0/0/0
0x230 2/0/0/0

Fig. 5. Input matching flow on MemoTbl.

values). Since each InTbl entry can hold 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*).

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 valid 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.

B. Execution Mechanism

In this section, we describe the execution mechanism of the auto-memoization processor. Fig.5 shows how the input sequences shown in Fig.3 are stored into InTbl/AddrTbl. Fig.5 also shows an input matching flow on MemoTbl as (a)...(f), and this flow corresponds to the function call at line 13 in Fig.2. Each of `0x200`, `0x210` and `0x220` in *next addr* corresponds to the variable `a`, `b` and `c` in Fig.2 respectively. Here, `X` in *input values* is a don't care nibble in cache line, and will not be tested for computation reuse. The entry of InTbl whose *parent idx* is `FF` indicates that it is the root node of an input sequence tree.

First, the processor reads the values of registers when the start address of the function `calc` is detected. Then, the processor searches the root entry whose *parent idx* is `FF` and whose *input values* match with the inputs on the current registers. Now, the entry whose index is `00` matches and this entry proves to be the root entry (a). Next, the address of `0x200` is read because *next addr* of the entry `00` in AddrTbl indicates `0x200` (b). Then, the processor searches the entry from InTbl whose *parent idx* is `00` and *input values* is 3, which is the value read from caches (c). This process is applied repeatedly until a mismatch of an input value is found (d)(e). In this example, all input matching for the current input values succeeds. Therefore, the processor can get the output values by referring to *OutTbl idx* of the terminal entry (f). Finally, the processor writes back the output values to the registers and the caches. This omits execution of the computation block and reduces total execution time.

III. Approximate Computing Stack Based on Computation Reuse

In this section, we propose an approximate computing stack based on computation reuse. The approximate computing stack consists of a programming framework, a compiler, and a modified auto-memoization processor. The programming framework provides a pragma for approximate computing, and the compiler encodes the approximation parameters directed with the pragma and tell them to the auto-memoization processor.

A. Outline of the Approximate Computing Stack

In this paper, we propose an approximate computing stack based on computation reuse. The stack allows programmers to easily apply approximate computing to various applications.

In this approximate computing stack, we utilize computation reuse as same as [6]. When the current input for a function is close to one of the previous inputs, reusing the previous results can achieve approximate computing. We implement this "approximate matching" between previous and the current inputs by masking some part of inputs. The approximate computing stack includes a programming framework which allows programmers to easily indicate which input should be masked and which bits of the input should be masked. The computing stack also includes a compiler which encode the indicated masking parameters into a binary, and the modified auto-memoization processor which can receive and use the parameters in the binary.

Here, we show an example of input masking. In image processing, a pixel value is sometimes used as an input of a function, and when a pixel value is represented in 32 bits, each of red(R), green(G), blue(B) elements is usually encoded in 8-bit fields as shown in Fig.6. In this figure, the pixel value is 0x006985a4. When a function takes such a pixel value as its input, approximate computation reuse can be applied to the function by masking the input. For example in Fig.6, the input value 0x006985a4 is masked with 0x00f0f0f0, and only the upper 4-bits of each RGB element are required to be matched with one of the previous inputs for computation reuse. In other words, differences in the lower 4-bits of each RGB element are ignored in input matching. Hence, if the function has been executed with a close input value, or 0x**6*8*a*, the execution with the input 0x006985a4 can be skipped by reusing the past result. If 0x00c0c0c0 is used instead of 0x00f0f0f0, more lower bits (6-bits) of each RGB

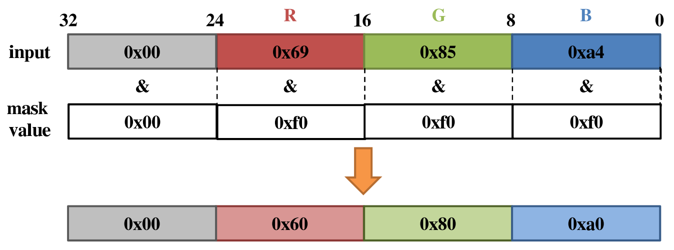


Fig. 6. An example for masking input.

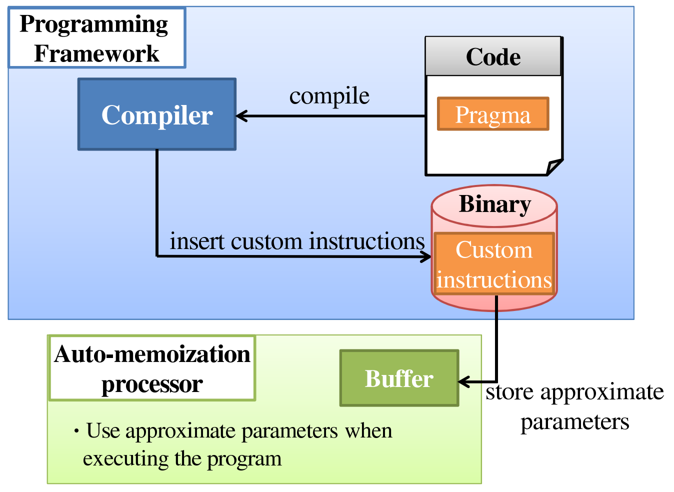

Programming Framework
Code
Pragma
Compiler
compile
Binary
insert custom instructions
Custom instructions
Auto-memoization processor
Buffer
store approximate parameters
• Use approximate parameters when executing the program

Fig. 7. Outline of the approximate computing stack based on computation reuse.

are ignored in input matching, and previous results can be reused more frequently although some output precision might be compromised.

Now, we show the overview of the approximate computing stack in Fig.7. The programming framework, one of the components of the approximate computing stack we propose, provides a directive for easily indicating inputs which should be masked and mask values. A programmer should first indicate which input should be masked and which part of the input should be masked, by using the directive, or the pragma we will describe in III-B. The compiler, another component of the stack, interprets the pragma into a custom instruction, and interprets the mask parameters indicated by the pragma into the operands of the instruction. The other component of the stack is the modified auto-memoization processor as the hardware platform. The processor decodes the custom instructions, and achieves approximate computing by applying approximate computation reuse to the indicated functions. The following paragraph in this section describes how programmers can use the pragma for indicating mask parameters, how the compiler interprets the pragma, and how the processor works when executing a binary which includes the custom instructions.

B. How to Indicate Mask Parameters

The approximate computing stack provides a pragma for indicating an input variable and a mask value for the variable.

```
#pragma approx(img, 0x00f0f0f0)
int col2gray(int img, float y){
  int R, G, B;
  R = 0.2989*img>>16;
  G = 0.5870*(img&0x0000ff00)>>8;
  B = 0.1140*img&0x000000ff;
  return pow(R, y)+pow(G, y)+pow(B, y);
}
```

Fig. 8. The example of the code which includes a pragma.

The reason why we adopt a pragma is that it is highly extensible and parsing mechanism can be simple. The format of the pragma is as follows;

#pragma approx (app_input, app_mask)

where *approx* is the keyword for approximate computation reuse, and *approx* takes two arguments *app_input* and *app_mask*. The first argument *app_input* is for indicating an input variable to which approximate input matching should be allowed, and the second argument *app_mask* is for a mask value. As *app_input*, not only the arguments of a function but also global variables which are read in the function can be designated. The pragma should be inserted just before the definition of a function to which approximate computing should be applied. Fig.8 shows a sample code in which a pragma for approximate computation reuse is inserted. In this code, `img` one of the input variables of the function `col2gray` is designated as *app_input*, and 0x00f0f0f0 is as *app_mask*, just before the definition of `col2gray`. By using this framework, programmers can flexibly indicate to which function approximate computing should be applied, to which input variable approximate computation reuse should be applied, and which parts of the input should be ignored in input matching for the computation reuse.

C. Compiler Specifications

The approximation parameters indicated with the pragma in a code should be transferred to and used by the modified auto-memoization processor. Therefore, we design a new custom instruction for representing the approximation parameters. The compiler interprets a pragma into the custom instruction, and encodes approximation parameters directed with the pragma, as the operands of the instruction. Then, the compiler inserts the custom instruction into the first line of the function to which approximate computing should be applied. When the modified auto-memoization processor executes a binary which is generated by the compiler and a custom instruction is detected, the processor can know that the function now being executed is the target of approximate computation reuse. Fig.9 shows a sample assembly code which is translated from the code shown in Fig.8 by the compiler. In this example, assume that the input variable `img` which should be compared approximately with the past inputs in computation reuse is mapped to the register r0. In Fig.9, because the function `col2gray` which to be reused approximately is defined from

```
1   20000 call   30000 <col2gray>
2                  :
3 00030000 <col2gray>:
4   30000 approx    r0, 0x00f0f0f0
5   30004 sub       sp, sp, #16
6   30008 str       r1, [sp, #8]
7   3000c str       r2, [sp, #12]
8                  :
9   30024 ret
```

Fig. 9.   An example assembly code with the custom instruction.

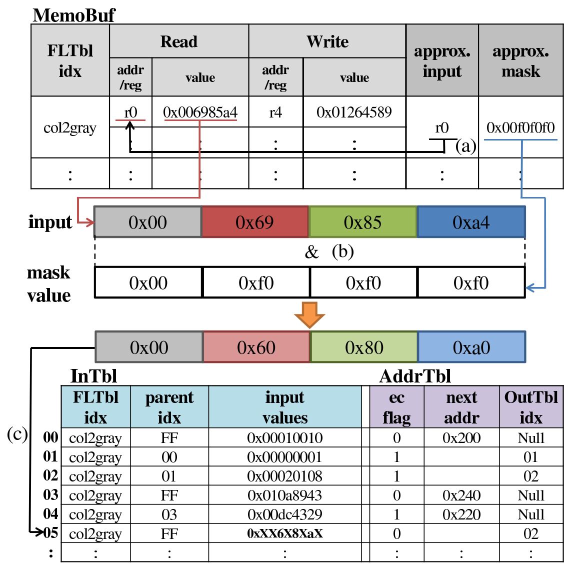
line 5, the compiler inserts the custom instruction `approx r0, 0x00f0f0f0` into line 6.

D. Execution Mechanism of the Modified Auto-Memoization Processor

In this section, we describe how the modified auto-memoization processor stores inputs into the MemoTbl and how it searches MemoTbl for the input.

1) Storing an Input into MemoTbl: In the approximate computing stack, when the modified auto-memoization processor executes the custom instruction, the processor should memorize the input which should be compared approximately with the past inputs, and the processor should also memorize the mask value for the input. To memorize these approximation parameters, we installed two new fields in MemoBuf. The field *approx. input* is for the input which to be compared approximately with past inputs, and *approx. value* is for the mask value. When executing each custom instruction, the auto-memoization processor stores approximation parameters which are indicated by the operand, into the extended fields of MemoBuf.

Fig.10 shows how the modified auto-memoization processor stores inputs and outputs into MemoTbl. In Fig.10, the upper table illustrates extended MemoBuf, and the lower tables illustrate InTbl and AddrTbl. Incidentally, in Fig.10, values stored in input value field are expressed in hexadecimal. When the program shown in Fig.9 is executed and the function is called at line 2, the processor compares the current input sequence with past input sequences in MemoTbl. If no past input sequence matches with the current input sequence, the processor executes the instructions of the function from line 5. When the processor executes instructions of the function, the custom instruction described in section III-C is detected first, and the function now being executed proves to be the target of approximate computation reuse. Then, when the custom instruction is executed, the auto-memoization processor stores approximation parameters into MemoBuf. In this example, the auto-memoization processor stores the register number `r0` into *approx. input* field and stores the mask value `0x00f0f0f0` into *approx. mask* field. After executing the custom instruction, the processor stores inputs and outputs into MemoBuf while executing instructions of the function. When the return instruction is detected in line 11, the processor stores the content of MemoBuf into MemoTbl. In Fig.10, because MemoBuf



FLTbl idx	Read		Write		approx. input	approx. mask
	addr /reg	value	addr /reg	value		
col2gray	r0	0x006985a4	r4	0x01264589	r0	0x00f0f0f0
	:	:	:	:		
:	:	:	:	:	:	:

	FLTbl idx	parent idx	input values	ec flag	next addr	OutTbl idx
00	col2gray	FF	0x00010010	0	0x200	Null
01	col2gray	00	0x00000001	1		01
02	col2gray	01	0x00020108	1		02
03	col2gray	FF	0x010a8943	0	0x240	Null
04	col2gray	03	0x00dc4329	1	0x220	Null
05	col2gray	FF	**0xXX6X8XaX**	0		02
:	:	:	:	:	:	:

Fig. 10.   How to store an input into modified the auto-memoization processor.

holds the register number `r0`, the processor refers to *approx. mask* field and applies the mask value `0x00f0f0f0` to the value in the value field which corresponds to the register number `r0` (Fig.10(a)(b)). As mentioned in section II, each InTbl entry holds single cache line and when a variable is read as an input value, its whole cache line is stored into an InTbl entry. When storing a whole cache line into an InTbl entry, the processor masks other parts than the variable in the cache line with "don't care" values. Similarly, when the approximate computing stack stores an input which to be compared approximately with past inputs into MemoTbl, the stack stores its masked bits of the input variable into MemoTbl as "don't care" values (Fig.10(c)).

2) Searching MemoTbl: The search process of the auto-memoization processor need not to be modified for approximate computation reuse. This is because, as mentioned in section III-D2, the auto-memoization processor stores the masked bits as "don't care" values. When a function is called, the auto-memoization processor compares the current input with past inputs in MemoTbl except "don't care" values of the input. In this way, approximate computation reuse is implemented by the input matching ignoring masked bits.

IV. Evaluation

We have evaluated the approximate computing stack from two viewpoints, its writeability and performance. We also measure precision degradation in outputs. We have evaluated the approximate computing stack by using cjpeg an image compression program from MediaBench benchmark suite. We applied approximate computation reuse to the function `forward_DCT` which is for the quantization.

A. Code Rewriting for Approximate Computing

Fig.11 and Fig.12 show how we rewrite around the function `forward_DCT`. Fig.11 shows the original function, and

```
1  void forward_DCT(JSAMPLE sample...){
2                    :
3    for (i = 0; i < DCTSIZE; i++) {
4      qval = div[i];
5      temp = work[i];
6      if (temp < 0) {
7        temp = -temp;
8        temp += qval>>1;
9        DIVIDE_BY(temp, qval);
10       temp = -temp;
11     } else {
12       temp += qval>>1;
13       DIVIDE_BY(temp, qval);
14     }
15     out[i] = (JCOEF) temp;
16   }
17 }
```

Fig. 11. The original code of forward_DCT in cjpeg.

```
1  void forward_DCT(JSAMPLE sample...){
2                    :
3    for (i = 0; i < DCTSIZE; i++) {
4      out[i] = q_loop(work[i], div[i]);
5    }
6                    :
7  }
8  #pragma approx(temp, 0xfffffff0)
9  JCOEF q_loop(temp, qval){
10   if (temp < 0) {
11     temp = -temp;
12     temp += qval>>1;
13     DIVIDE_BY(temp, qval);
14     temp = -temp;
15   } else {
16     temp += qval>>1;
17     DIVIDE_BY(temp, qval);
18   }
19   return (JCOEF) temp;
20 }
```

Fig. 12. The code of forward_DCT rewritten for approximate computation reuse.

TABLE I
SIMULATION PARAMETERS.

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

Fig.12 shows the corresponding code after rewrite. In the original function shown in Fig.11, the part from line 7 to 17 takes on quantization process. If a programmer thinks that the part will be suitable for approximate computing, the programmer first should cut out the part from the function, and should define the part as another function. In Fig.12, the defined function `q_loop` is at from line 11 to 22, and the function is called at line 5 in `forward_DCT`. In addition, the programmer should insert the pragma statement `#pragma approx(temp, 0xfffffff0)` to mask the lower 4-bits of the variable `temp` at line 10, for applying approximate computation reuse to `q_loop`. With such slight program modification, programmers can apply approximate computing to the appropriate part of the code.

B. Performance Evaluation

We have measured execution cycles and reuse rates to evaluate the performance of approximate computation reuse. We have developed a single-issue SPARC-V8 simulator equipped with the auto-memoization mechanisms. The simulation parameters are shown in TABLE I. The cache structure and the instruction latencies are based on SPARC64-III[7]. The on-chip CAM for InTbl in MemoTbl is modeled on DC18288[8] (32Bytes $\times$ 4K lines). The latencies of the CAM are defined on the assumption that the clock frequency of the processor is about 2 GHz, and is 10-times higher than the CAM.

First, Fig.13 shows the execution cycles of cjpeg with the following five configurations;

- (Q) Original program
- (Q'0) Rewritten program shown in Fig.12
- (Q'2) Rewritten program with the lower 2-bits of `temp` being masked
- (Q'4) Rewritten program with the lower 4-bits of `temp` being masked
- (Q'6) Rewritten program with the lower 6-bits of `temp` being masked

Computation reuse is applied to these all configurations by the auto-memoization processor, and each bar is normalized to the execution cycles of the original program without computation reuse. Incidentally, rewriting the program may bring some function call overhead. Thus, we evaluated (Q'0) to confirm the influence due to the function call overhead.

The legend in Fig.13 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 overhead for writing back the outputs from MemoTbl to the registers and the caches ('*write*'), the first-level and shared second-level data cache miss penalties ('*D$1*', '*D$2*'), and the register window miss penalty ('*window*') respectively. The results of (Q'2), (Q'4), (Q'6) show that widening the masked bits increases *read* and *write* slightly. However, widening the masked bits reduces *exec* considerably and improves the overall performance. We also show the reuse rates of the function `forward_DCT` in TABLE II. The results show that widening the masked bits also considerably

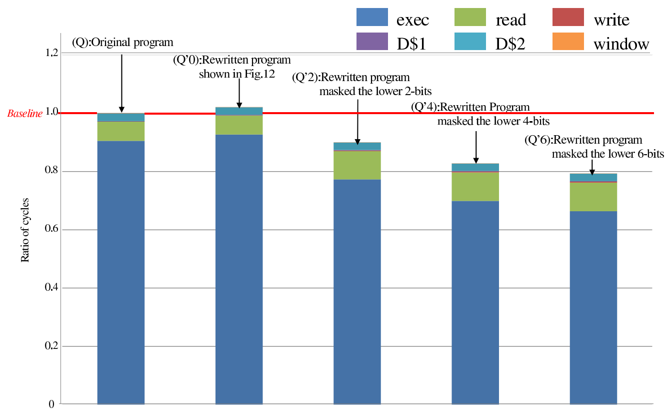


Fig. 13. Execution cycles.

TABLE II
REUSE RATES OF FUNCTION "FORWARD_DCT."

(Q)	(Q'0)	(Q'2)	(Q'4)	(Q'6)
0.019%	0.029%	51.7%	76.2%	91.1%

improves the reuse rates.

C. Output Precision

We also evaluated the deterioration of the output precision. Fig.14 shows the output images of cjpeg with the configurations (Q), (Q'2), (Q'4) and (Q'6). As shown in Fig.14, almost no quality reduction can be confirmed by sight in the outputs with (Q'2) and (Q'4). On the other hand, the output with (Q'6) has slight quality loss, but it will be within the acceptable range of many applications. We numerically evaluated the deteriorations of the output precision by using SNR (Signal-to-Noise Ratio). We measured the SNR between the input and each output. Results are shown in TABLE III. These results give evidence for that the quality loss is very small.

V. CONCLUSIONS

In this paper, we proposed an approximate computing stack based on computation reuse. The approximate computing stack consists of a programming framework, a compiler, and the modified auto-memoization processor. The programming framework allows programmers to easily apply approximate computing to various applications by using pragmas, and the compiler encodes the approximation parameters directed with the pragmas and tell them to the auto-memoization processor. Through an evaluation with cjpeg from MediaBench, execution cycles are reduced by 22.3% in maximum, and reuse rate is improved by 29.5% in maximum, with negligible quality deterioration in outputs and with slight program modification. One of our future works is to investigate a mechanism for dynamically adjusting mask values, aiming at achieving both the best performance and lower error rate than a threshold indicated by a programmer. Considering another platform than the auto-memoization processor is also our future work. In this work, we used the modified auto-memoization processor as the platform for the approximate computing stack. However,

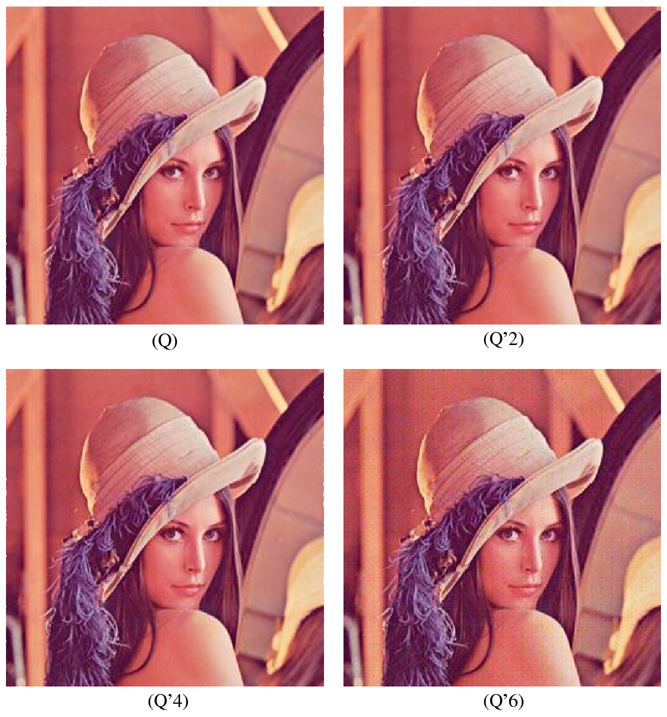


Fig. 14. Output images.

TABLE III
SIGNAL-TO-NOISE RATIO (DECIBEL).

(Q)	(Q'2)	(Q'4)	(Q'6)
27.20	27.18	27.07	25.48

it can be replaced by a commodity processor equipped with an accelerator which includes a simple lookup table, or with a software library for approximate computation reuse. Providing various platform options will enhance the portability of this approximate computing stack.

REFERENCES

[1] E. Hadi *et al.*, "Architecture support for disciplined approximate programming," in *Proc. 17th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, Apr. 2012, pp. 301–312.

[2] G. Vaibhav *et al.*, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, Jan. 2013.

[3] T. Tsumura *et al.*, "Design and evaluation of an auto-memoization processor," in *Proc. Parallel and Distributed Computing and Networks*, Feb. 2007, pp. 245–250.

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

[5] J. Huang *et al.*, "Exploiting basic block value locality with block reuse," in *Proc. 5th Int'l Symp. on High-Performance Computer Architecture (HPCA-5)*, Jan. 1999, pp. 106–114.

[6] T. Tsumura *et al.*, "An evaluation of telerant function reuse on stereo depth extraction," *IPSJ Trans. on Advanced Computing Systems*, vol. 44, no. SIG 11(ACS 3), pp. 246–256, Aug. 2003, (in Japanese).

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

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