2009 International Conference on Parallel and Distributed Computing, Applications and Technologies

A Speculative Technique for Auto-Memoization Processor with Multithreading

Yushi KAMIYA*, 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. This processor automatically and dynamically memoizes both functions and loop iterations, and skips their execution by reusing their results. On the other hand, multi/many-core processors have come into wide use. The number of cores is expected to increase to a hundred or more. However, many programs do not have so much parallelism in them. Therefore it becomes very important to consider how to utilize many cores effectively. This paper describes a speedup technique for auto-memoization processor using speculative multi-threading. Two speculative threads will be forked on reuse test. The one assumes that the reuse test will succeed, and executes the following codes of the reuse target block speculatively. The other assumes that the reuse test will fail, and executes the reuse target block. These two threads conceal the overhead of auto-memoization processor. The result of the experiment with SPEC CPU95 suite benchmarks shows that proposing method improves the maximum speedup from 13.9% to 36.0%.**

***Index Terms*—memoization, multicore, multithread**

I. Introduction

As electric power consumption and calorific power are increasing, and semiconductors are going smaller, it becomes difficult to raise clock rates of microprocessors. Now, microprocessors are facing to the crossroads of speedup techniques.

So far, the latency of microprocessors have been controlled by the gate latencies. Hence, transistor scaling provided higher clock for microprocessors, and it made microprocessors faster. But now, the interconnect delay is going major and the main memory and other storage units are going relatively slower. In near future, high clock rate cannot achieve good microprocessor performance by itself.

Speedup techniques based on ILP (Instruction-Level Parallelism), such as superscalar or SIMD, have been counted on. However, the effect of these techniques has proved to be limited. One reason is that many programs have little distinct parallelism, and it is pretty difficult for compilers to come across latent parallelism. Another reason is the limitations caused by other processor resources, for example, memory throughput. Even if the compilers can extract parallelism, the memory throughput restricts the issue width. Therefore, microprocessors are under the pressure of necessity of novel speedup techniques.

In response to this distress, multi-core processors attract a great deal of attention. They are now in wide use from generic processors for PCs to embedded processors[1]. Many-core processors such as UltraSPARC T2[2] and TILE64[3] also are available now.

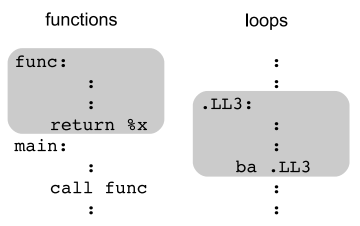
Several speedup techniques for multi-core processors have been proposed. For example, some compilers have the function of auto-parallelization. But it is difficult to find thread level parallelism (TLP) in programs, and parallel programming is also difficult. Consequently, some cores may be idle on many-core processors because of the lack of parallelism. Hence, it becomes much important to utilize all cores on the processor.

Meanwhile, in the software field, memoization[4] is a widely used programming technique for speedup. It is storing the results of functions for later reuse, and avoids recomputing them. As a speedup technique, memoization has no relation to parallelism of programs. It depends upon value locality, especially input values of functions. Therefore, memoization has a potential for breaking through the stone wall the speedup techniques based on ILP have run into.

Memoization brings a good result on expensive functions, but it requires rewrite of target programs. The traditional load-modules or binaries cannot benefit from memoization. Furthermore, the effectiveness of memoization is influenced much by programmers. Rewriting programs using memoization occasionally makes the programs slower. Memoization costs a certain overhead because it is implemented by software.

We propose an auto-memoization processor which makes traditional load-modules faster without any software assist. There is no need to rewrite or recompile target programs. Our processor detects functions and loop iterations dynamically, and memoize them automatically. However, memoization has some overhead for input matching and output write-backing. Our previous research showed that the overhead cancels the speedup with memoization on some programs[5].

This paper shows a technique of how to reduce these overheads by using multi-threading. Supposing to design an



```
func:
     :
     :
  return %x
main:
     :
  call func
     :
```

```
        :
        :
.LL3:
        :
        :
    ba .LL3
        :
        :
```

Fig. 1. Memoizable Instruction Regions

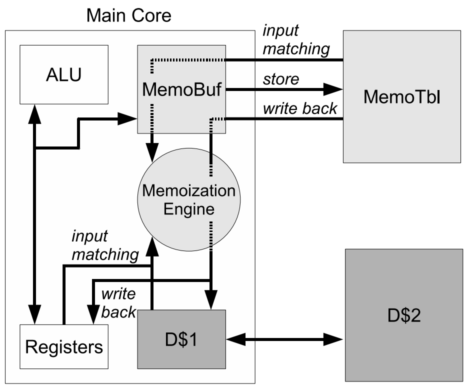


Fig. 2. Structure of Auto-Memoization Processor

```
int main() {
  int g = 4;
  int h = 2;
  int a = A(3);
}
int A(int x) {
  return B(x*2);
}

int B(int x) {
  int b = x;
  if(x > 3)
    b += g;
  else
    b += h;
  return b;
}
```

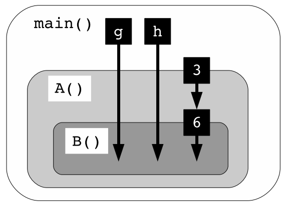


Fig. 3. An example of nested functions

auto-memoization processor based on multi-core processor, surplus cores are utilized effectively and are able to contribute to speedup with this technique.

II. Auto-Memoization Processor

A. Outline and Structure

Our auto-memoization processor memoizes functions and loops dynamically and automatically. Fig. 1 shows the memoizable instruction regions. A region between the instruction with a callee label and `return` instruction is detected as a memoizable function. A region between a backward branch instruction and its branch target is detected as a memoizable loop iteration. This processor detects these memoizable regions automatically and memoizes them.

Fig. 2 shows the structure of auto-memoization processor. Memoization system consists of memoization engine and memo table '**MemoTbl**.' Processor core has also '**MemoBuf**' a small writing buffer for MemoTbl. Entering to the memoizable region, the processor refers to the MemoTbl and compares current inputs with former inputs which are stored in MemoTbl.

If the current input set matches with one of the stored input sets on the MemoTbl, the memoization engine write backs the stored outputs to cache and registers. This omits the execution of the region and reduce whole execution time.

If the current input set does not match with any past input sets, processor stores the inputs and the outputs of the region into MemoBuf while executing the region as usual. At the end of the region, the memoization engine stores the contents of MemoBuf into MemoTbl for future reuse.

MemoBuf: Through the execution of an instruction region, the processor stores the addresses and the values of inputs/outputs to MemoBuf. Note that the memoizable regions in programs are usually nested. Fig. 3 shows a simple example. A function B is called in another function A, and B uses global variables g, h as its inputs. When B is directly called, g and h are inputs only for B. When A is called, these variables g, h are inputs not only for B but also A. That is, the processor should memorize inputs/outputs of nested regions simultaneously. We make the depth of MemoBuf six, and each MemoBuf line can store the one of the nested instruction regions. At the end of a region, the corresponding line is popped from MemoBuf and copied into MemoTbl.

MemoTbl: Now, MemoTbl should keep this input tree in it. MemoTbl consists of CAM/RAM parts. The CAM part is for edges of the input tree, and the RAM part is for nodes of the input tree and outputs. Considering the CAM width and granularity, we define the length of edge value as 16 bytes. An edge of input tree is corresponds to a CAM line. Fig. 4 shows the input tree and the brief structure of MemoTbl. MemoTbl consists of following tables as shown in Fig. 4.

RF: stores the start address of instruction set.

RB: stores the input data set of instruction set.

RA: stores the input address set of instruction set.

W1: stores the output data set of instruction set.

The RF, RA, and W1 are constructed as RAM. On the other hand, the RB is constructed as a CAM (Content Addressable Memory) array, so that input matching can be implemented by fast associative search. Generally, the sequence of input addresses of a reusable block varies depending on their input values. This comes from the variable which holds a memory address or a condition value for a branch. Hence, the all variations of input sequences for a certain instruction block can be represented as a tree structure, and one of the sequences can be represented as a pass from the root to a leaf as shown in Fig.4.

When our auto-memoization processor runs a program and detects a call instruction or a backward branch instruction, it compares current input values with former input values which are stored in the MemoTbl. We call this '**input matching**.'

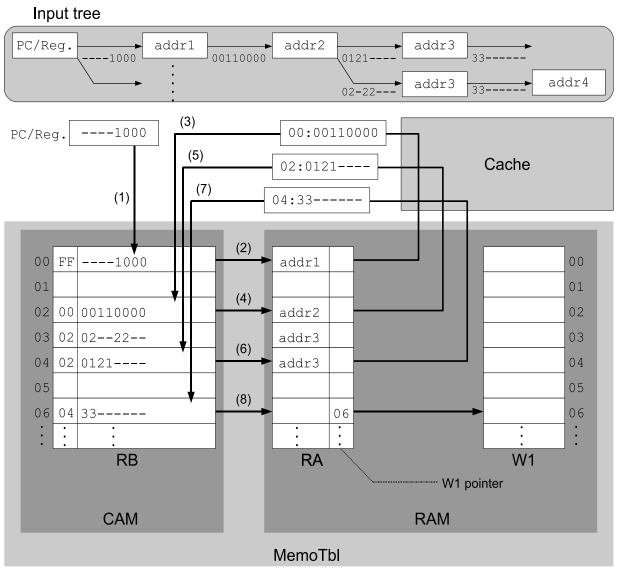


Fig. 4. Structure of MemoTbl

B. Input Matching

Now, we explain the behavior of MemoTbl and input matching using Fig.4.

When the auto-memoization processor detects a function or a loop iteration, it first searches the block address through the RF table for deciding the inputs of the reusable block is stored or not. Next, input matching starts. This process is shown as (1)...(8) in Fig.4. First, the memoization engine reads the value of program counter (PC) and registers. Suppose the value was '`----1000`'. The CAM entry stores whole cache line which includes the input. Hence, the non-input bits should be masked. They are implemented by don't care bits on ternary CAM. The index `00` line of CAM is found (1). Each RB line has next input address in the same line of RA. In Fig.4, the next address is decided as `addr1` (2). The processor reads `addr1`, gets the next input values, and search them through RB again (3). This process is applied repeatedly (5)...(8) until all input values are confirmed. If all inputs of a reusable block have matched with one of the stored input set on the MemoTbl, our processor can get the output set from W1 by using the index for W1 (called 'W1 pointer') stored in terminal RA entry.

Meanwhile, accessing MemoTbl causes overhead inevitably. Through input matching, searching RB, referring RA, and reading caches cost a certain time. When input matching has succeeded, outputs of the reusable block should be written back from W1. This also costs some time. We call these two kind of overheads '**Reuse Overheads**.'

For some reusable blocks, reuse overhead overcomes the eliminated execution cycles by reuse. This goes for some blocks which have many input values to be tested, and all tiny blocks. Hence, the auto-memoization processor estimates the effect of reuse, and avoids memoing/testing unsuitable instruction regions. This can reduce useless input matching

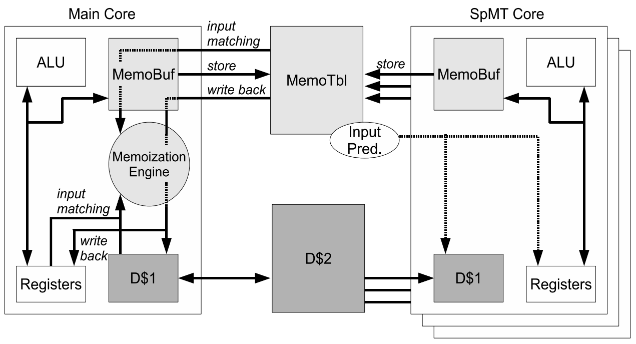


Fig. 5. Auto-Memoization Processor with SpMT cores

and contributes to good performance.

C. Parallel Speculative Execution

As a matter of course, memoization can omit the execution of a instruction region only if the current input values for the region match completely with the input values which are used in former execution. Hence, for example, a loop which uses its iterator variable as its input never benefit from memoization.

Meanwhile, many of microprocessor companies are switching to multi-core designs today. There is a story going around that processors with hundreds of cores may be delivered in another decade[6]. But how we can use these many-core processors effectively is still under review between researchers.

Speculative multi-threading (SpMT) is an answer to this question, but it is not so easy to deal with crossthread dependence violation and thread squash. We append some SpMT cores to our auto-memoization processor. These cores help the unsuitable regions for memoization mentioned above. Fig.5 shows the new structure of the auto-memoization processor with SpMT cores.

Each core has its own MemoBuf and first level data cache. The second level data cache and MemoTbl is shared between all cores. While the main core executes an memoizable instruction region, SpMT cores execute the same region using predicted inputs. The inputs are predicted from the last value being managed by the main core and strides of the values. For example, when the main core detects function calls `f(1,1)` and `f(1,2)`, the coming inputs of `f()` are predicted as `(1,3)`, `(1,4)`, and so on using the input stride. Then, SpMT cores executes `f(1,3)`, `f(1,4)` in advance. With three SpMT cores, the instruction region is executed using three predicted input sets concurrently with the main core. SpMT cores store the results into MemoTbl. If the input prediction was correct, the main core finds the next result in MemoTbl. Unlike as ordinary speculative execution, even if the input speculation proves to be incorrect later, the main core need not to pay a cost for any backout processes. This extension can omit the execution of instruction regions whose inputs show monotonous increase/decrease.

These SpMT cores not only omit some execution, but also works as a cache prefetch technique[7], [8].

III. Reduction of Reuse Overhead

A. Proposal

In this section, we pay attention to reuse overhead. Reuse overhead can be divided into two classes. One is the overhead which costs when input matching succeeds. The other is the overhead which costs when input matching failed. We improve an auto-memoization processor for concealing these two overheads. Now the processor can run additional two threads. The one assumes that the reuse test will succeed, and executes the following codes of the reuse target block speculatively. The other assumes that the reuse test will fail, and executes the reuse target block normally. We will explain these threads in detail.

Main thread (Memoization thread): This thread is the main thread of the auto-memoization processor. This thread executes instructions sequentially, memoizes functions and loops, and tries input matching.

Preceding thread: This thread conceals the overhead in the case of that input matching succeeds. This thread supposes that input matching will succeeds. That is to say, outputs of the testing block are written back before input matching is finished, and preceding thread executes following instructions of the testing block speculatively. Preceding thread runs in parallel with main thread and conceals the reuse overhead.

No-Memoization thread: This thread conceals the overhead in the case of that input matching fails. If the current input value sequence cannot be found on MemoTbl, main core must execute the reusable block normally. In this case, whole searching overhead is accumulated to the execution cycles. No-memoization thread executes the reusable block without searching inputs through MemoTbl assuming that reuse test will be fail. This thread runs in parallel with main thread and conceals the searching overhead.

We improve the auto-memoization processor for being able to run these three threads simultaneously. The processor have three cores at the lowest and the three cores take charge of three thread mentioned above by turns. If the processor has four or more cores, the surplus cores are used for parallel speculative execution. These cores are called as SpMT cores. In this paper, we assume that the processor has five cores. Three cores out of the five are assigned to main thread, preceding thread, and no-memoization thread respectively. Then the other two cores are assigned to parallel speculative execution.

B. Execution Model

Function call: Fig. 6 shows a behavior of the processor on a function call. The processor has cores named (A), (B) and (C). At the beginning of the program, core (A), core (B) and core (C) is assigned to main thread, preceding thread, and no-memoization thread respectively.

When the core (A) which is assigned to main thread detects the call instruction of function `f()` at t_1, (A) starts input matching. Simultaneously, (B) and (C) copy the value of program counter of (A). The core (C) which is assigned to no-memoization thread skips input matching of `f()` and starts to execute function `f()` normally. When (A) finds that the first part of current inputs hits an entry on MemoTbl at t_2, (B) starts to execute following codes of `f()` after speculating suitable W1 index and writing back a past output value sequence from W1 table.

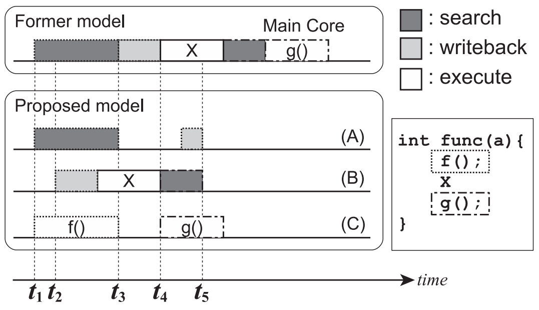


Fig. 6. Execution Model for a function region

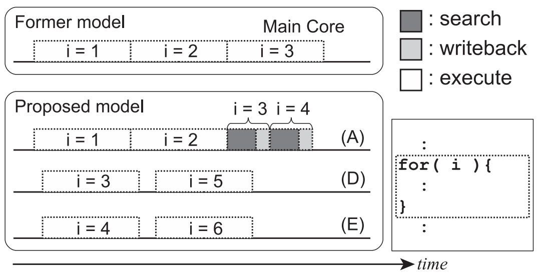


Fig. 7. Execution model for a loop region

At t_3, (A) verifies that current input value sequence completely coincides with a past input sequence on MemoTbl, and (B) examines whether preceding thread on (B) can run forward or must be squashed. This can be examined by comparing the W1 index which was picked up at t_2 by preceding thread with the W1 index which the terminal entry of RA points to. If these W1 indices are same, the preceding thread on (B) turns into the main thread, and the main thread on (A) will be squashed. Hereafter, the core (A) runs preceding thread and the core (B) runs main thread. Simultaneously the no-memoization thread on (C) will be also squashed. These three threads runs independently, therefore preceding thread and no-memoization thread do not prevent main thread from accessing to MemoTbl and main memory.

Although main thread finds that `f()` is reusable at t_3, preceding thread often must be squashed. This is caused by a wrong speculation about W1 pointer at t_2. In this case, the main thread on (A) runs forward and the threads on (B) and (C) are squashed without exchanging threads. The squashes of two threads on (B) and (C) are not observed by (A), and there comes into no overhead. Now, assume that the preceding thread on (B) is proved to be correct at t_3. Then, (B) starts input matching on detecting a new call instruction against function `g()` at t_4. As the behaviour at t_1, the no-memoization thread on (C) executes function `g()` normally. When (B) detects that input matching fails at t_4, (B) and (C) exchange

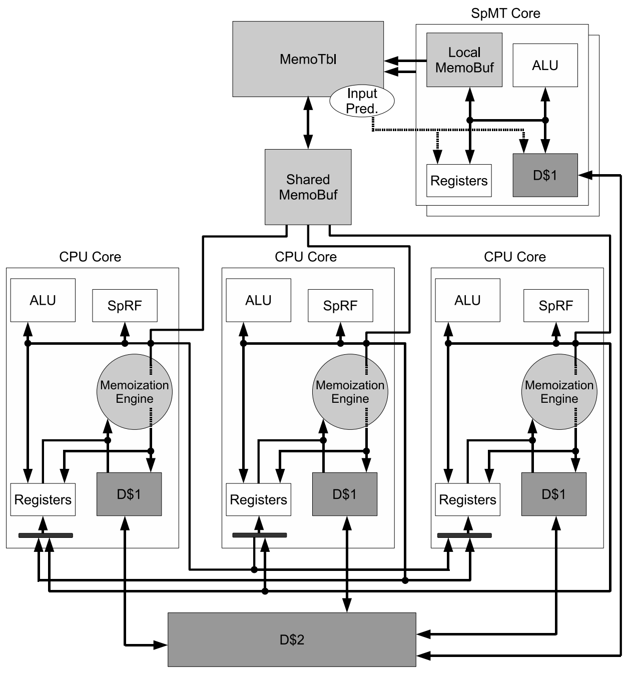


Fig. 8. Structure of Proposed Auto-Memoization Processor

their threads. The core (B) takes no-memoization thread and the core (C) takes main thread after t_4. In this case, the no-memoization thread on (C) conceals the overhead of input matching by the main thread on (B).

This model allows preceding thread and no memoization thread neither to call another function nor to return to the callee. Therefore, we can avoid an explosive increase of MemoBuf size.

On loop iteration: Fig. 7 shows a behavior of the processor on executing a loop iteration. The processor has cores named (D) and (E), and they are assigned to SpMT threads statically. When the processor detects a loop iteration, (D) and (E) start value prediction and parallel speculative execution.

In the case of a loop iteration, no preceding thread and no-memoization thread are generated, and two cores for them are suspended. On the other hand, the SpMT threads are assigned statically to cores, and they work not only on function calls but also loop iterations.

An example of a loop is shown in Fig. 7. The SpMT threads on (D) and (E) speculate input sequences of the loop by stride value prediction. The SpMT threads execute the loop iteration using a speculated input sequence independently.

IV. Implementation

A. Architecture

We have improved an auto-memoization processor to implement the model explained in previous section. Fig. 8 shows the improved auto-memoization processor.

Thread exchange between cores occurs on our model when the preceding thread succeeds its speculation or when the main thread finds that input matching fails. Therefore, three cores have a bus for coping the value of program counter and register data set between the cores. Each core has completely same structure and shares second level data cache, MemoBuf and MemoTbl. The ALU output bus on each core connects not only the register file on the core and its first level data cache but also the register files on the other cores. Hence, the output value on a core can be also written to the other cores.

These three cores are illustrated at the middle of the Fig. 8. Each of the three cores takes one of the three threads, main thread, preceding thread and no-memoization thread respectively, and exchanges their roles dynamically. So, it is necessary to make sure of the coherence of registers and caches. The three cores communicate register values each other, and get the required PC/register values from the main core. This copying overhead may be expensive, if the cores copy register values from the main core whenever new threads are generated. To reduce this copying overhead, the processor has two additional units. The one is **SpRF (Speculative Register File)** and the other is **RegMask (Register Mask)**. The SpRFs have a similar structure to the register file. Both preceding thread and no-memoization thread stores values on SpRFs instead of register files.

Next, let's see the two cores for parallel speculative execution. They are illustrated at the top of the Fig. 8. These cores connect to MemoTbl and D$2 and have MemoBuf locally so as to store input/output sequences. They run parallel speculative execution independently.

B. Register/Memory Coherency

We will show a mechanism to guarantee the coherence of register/memory values between cores. On accessing register file, the main core refers the RegMask to see which register values are overwritten by other cores. The latest value is stored in register file or SpRF. If the register the main core is going to access is masked by RegMask, the register value is dirty and the main core should get the value from SpRF. In this case, the main core stalls and transfer the latest value from the SpRF on one of the other two cores.

Next, we show the memory coherence. If preceding thread or no-memoization thread fails, output values on its MemoBuf become unusable. Hence, preceding thread and no-memoization thread writes output values not onto memory but onto their own MemoBufs. This prevents memory values from being violated by these threads.

C. Prediction Pointer

Preceding thread should pick up an output sequence of current reusable block for executing the following codes after the block. We have added a **prediction pointer** field to RA table in MemoTbl. Halfway through input matching, multiple W1 entries are potential output sequences. They will be narrowed down to only one sequence at the end of input matching, but it is unclear which sequence will be selected finally. The prediction pointer field holds one of the candidate among potential output sequences. Preceding thread considers this prediction pointer value as the pointer to correct W1 entry, and writes back the output sequence to the registers and MemoBuf. When storing new input/output sequence onto MemoTbl, prediction pointer fields of all RA entries which construct the storing input sequence are filled with the pointer to the W1 entry on which the output sequence is going to be stored. When input matching succeeds and one entry is picked up from W1, the value of prediction pointer is updated with a pointer to the entry.

V. Performance Evaluation

A. Simulation Parameters

We have developed a single-issue simple SPARC-V8 simulator with auto-memoization structures. This section discusses the performance of the processor. The simulation parameters are shown in Table I. The cache structure and the instruction latency are based on SPARC64[9].

As for memo tables, we defined the size of shared MemoBuf as 64 KB (32 Bytes×256 lines×8 sets), the size of local MemoBuf provided for speculative core as 48 KB (32 Bytes×256 lines×6 sets), and the size of MemoTbl CAM as 128 KB (32 Bytes×4K lines). The size of MemoBuf is near to D1 cache, and the size of MemoTbl is very smaller than D2 cache. And the size of SpRF appended in this proposal as 416 Bytes (4 Bytes×104 registers). The on-chip CAM is modeled on MOSAID DC18288[10]. We assume the latency for register synchronization as 32 bit/cycle.

B. Results with SPEC CPU95

We evaluated our auto-memoization processor with multi-threading which has three cores. Workloads are all benchmarks in SPEC CPU95 suits and are executed with 'train' dataset. All benchmarks are compiled by gcc-3.0.2 with -msupersparc -O2 options.

Fig. 9 shows the normalized execution cycles of the SPEC CPU95 benchmark programs. Each bar is normalized to the number of executed cycles without memoization(N). Each benchmark is represented by five bars in this chart. The leftmost bar (N) plots the baseline that is the execution cycles original benchmark costs. The second bar (M) plots the cycles using auto-memoization with no speculative cores. The middle bar (P) plots the cycles using parallel speculative execution with two SpMT cores in addition to a main core. This model has three cores in total. The fourth bar (S) plots the cycles by overhead concealing model proposed in this paper. This model also has three cores for three threads; main thread, preceding thread and no-memoization thread. The fifth bar (A) plots the cycles by the hybrid processor of the parallel speculative execution and the proposal model with five cores. One core is for the main thread, two cores are for parallel speculative execution, and the other two cores are for overhead concealing threads. This hybrid model is illustrated in Fig. 8.

The legend shows the itemized statements of cycles. They represent the executed instruction cycles ('**exec**'), the reuse overhead ('**reuse_ovh**'), the overhead of register value transfer between cores ('**regcopy**'), D1 cache miss penalty ('**D$1**'),

TABLE I
Processor Parameters

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
Shared MemoBuf	64 kBytes
Local MemoBuf (in SpMT cores)	48 kBytes (×2)
MemoTbl CAM	128 kBytes
Comparison (register and CAM)	9 cycles/32bytes
Comparison (Cache and CAM)	10 cycles/32bytes
Write back (MemoTbl to Register or Cache)	1 cycle/32bytes
SpRF	416 Bytes
Register copy	1 cycle/32bits

shared D2 cache miss penalty ('**D$2**'), and register window miss penalty ('**window**') respectively.

As shown in Fig. 9, the execution cycles of some benchmark programs such as 125.turb3d, 146.wave5 and 147.vortex are reduced by the auto-memoization processor (M). On the other hand, there are several benchmark programs whose cycles increase. This is quite noticeable with 129.compress, 130.li, and 134.perl. These programs will tend to fail input matching in many cases and there should be a little reusable blocks in them. Consequently, the speedup by (M) is canceled out by reuse overhead on average. Hence, it is very important to reduce reuse overheads.

The parallel speculative execution (P) works very well with CFP benchmarks. Many CFP benchmark programs have large and heavy loop iterations which calculates such as product-sum of matrices. This would be the reason why CFP programs are suitable for (P). On the other hand, (P) can contribute little toward CINT programs' speedup. The parallel speculative execution is a good method for utilizing idle cores indeed, but the effect is limited to certain programs.

The multi-threading model (S) proposed in this paper achieves a big reduction of reuse overhead. It is quite remarkable with some CINT programs such as 129.compress, 130.li, 134.perl and 147.vortex. However, the execution cycles of almost all the programs of CFP are not reduced as compared with (M). The reusable functions account for rather small ratio in CFP benchmark programs, and the function-reuse overhead is essentially small as seen in the (M) results.

As shown above, the model (P) is effective for the programs which have monotonous and heavy loop iterations such as CFP benchmark programs. On the other hand, the model (S) is effective for the programs which have many and large reusable functions such as CINT benchmark programs.

Then, now let's see how (A): the hybrid model of (P) and (S) works. Each of the CFP results with (A) is equal to or slightly better than its result with (P). On the other hand, each of the CINT results with (A) is equal to or pretty better than its result with (S). Consequently, the hybrid method (A) can

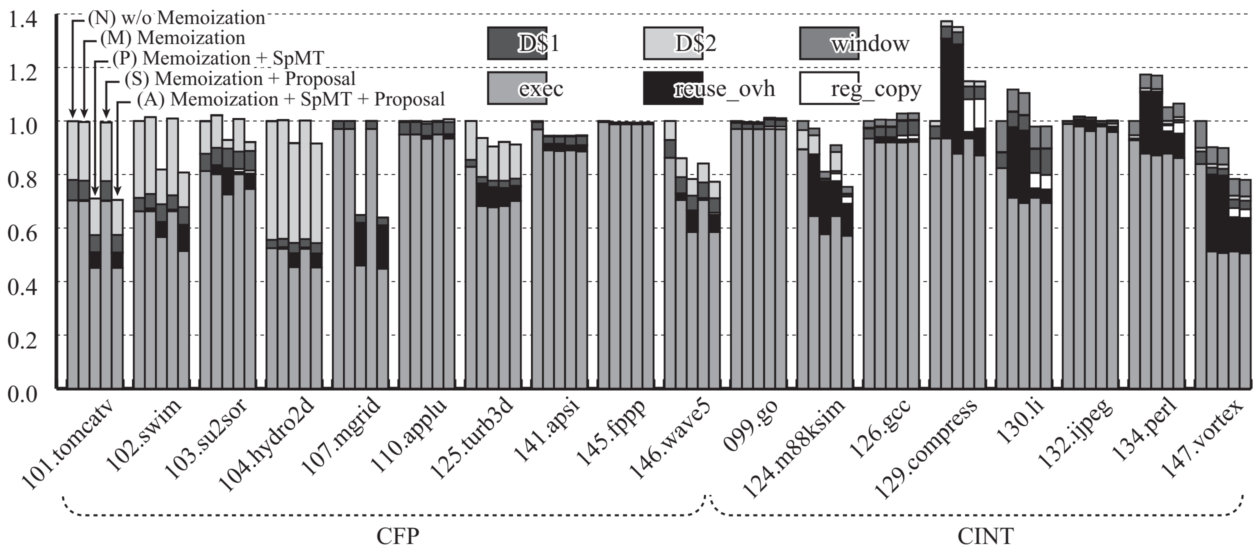


Fig. 9. Normalized execution time (SPEC CPU95)

TABLE II
REDUCED EXECUTION CYCLES (SPEC CPU95)

	Average	Maximum
(M) Memoization	-0.1%	13.9% (146.wave5)
(P) Memoization + SpMT	5.6%	35.2% (107.mgrid)
(S) Memoization + Proposal	2.1%	21.7% (147.vortex)
(A) Memoization + SpMT + Proposal	9.0%	36.0% (107.mgrid)

achieve equal performance to the better one of (P) and (S). Especially, the benefit of the hybrid model is very distinct on 124.m88ksim benchmark program which has both monotonous loops and reusable functions. Table II summarizes these results of all models.

VI. CONCLUSION

Generally, it is difficult to accelerate the programs which have little parallelism in them. An auto-memoization processor is an answer to this problem, but memoization costs a certain overhead for computation reuse. Running programs which have little parallelism, multiple cores will be idle on trendy multi/many-core processors. Hence, merging auto-memoization processor and multi-threading can conceal reuse overhead and would be able to accelerate the programs with no parallelism.

In this paper, we have proposed an auto-memoization processor with multi-threading which can reduce the reuse overhead. The hybrid model of parallel speculative execution and multi-threading for concealing reuse overhead can enjoy mutual advantage of them.

The performance evaluation through SPEC CPU95 benchmarks shows that many programs can gain speedup and that the proposed model pushes up the maximum eliminated cycles from 13.9% to 36.0% and the average eliminated cycles from -0.1% to 9.0%.

Our future work is to change the assignment of cores to the threads dynamically. On the current implementation, the cores for parallel speculative execution and the three cores for concealing overheads does not exchange their threads each other. This makes some cores idle. For example, preceding thread and no-memoization thread are not generated on loop iterations, and two cores for them will be idle. We plan to let these two cores run parallel speculative execution threads when they are idle.

There is an another problem that the processor does not conceal the overheads of reusing loop iterations. Hence, a further improvement of the processor model will be required.

ACKNOWLEDGMENT

This research was partially supported by the Kayamori Foundation of Informational Science Advancement.

REFERENCES

[1] ARM Ltd, *The ARM Cortex-A9 Processors*, Sep 2007.
[2] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn, "UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC," A-SSCC 2007, Tech. Rep., 2007.
[3] Tilera Corporation, *Product Brief: TILE64 Processor*, 2007.
[4] P. Norvig, *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1992.
[5] 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.
[6] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," Intel Corp., White Paper, 2005.
[7] J. A. Brown, H. Wang1, G. Chrysos, P. H. Wang1, and J. P. Shen, "Speculative precomputation on chip multiprocessors," *The 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2002.
[8] I. Ganusov and M. Burtscher, "Future execution: A hardware prefetching technique for chip multiprocessors," *Parallel Architectures and Compilation Techniques*, pp. 350–360, Sep 2005.
[9] *SPARC64-III User's Guide*, HAL Computer Systems/Fujitsu, May 1998.
[10] MOSAID Technologies Inc., *Feature Sheet: MOSAID Class-IC DC18288*, 1st ed., Feb. 2003.