This paper is author's private version of the paper published as follows:
Proc. of Int'l. Symp. on System-on-Chip 2011 (SoC2011), pp.22–28
Copyright (C) 2011 IEEE
DOI: 10.1109/ISSOC.2011.6089225

A Hybrid Model of Speculative Execution and Scout Threading for Auto-Memoization Processor

Tomoki IKEGAYA*, Ryosuke ODA*, Tatsuhiro YAMADA*,
Tomoaki TSUMURA*, Hiroshi MATSUO* and Yasuhiko NAKASHIMA†

*Nagoya Institute of Technology, Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

†Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara, Japan
Email: nakashim@is.naist.jp

***Abstract*—We have proposed an auto-memoization processor based on computation reuse, and merged it with speculative multi-threading based on value prediction into a parallel speculative execution. In the parallel speculative execution model, speculative cores do not work when the target instruction region is not suitable for computation reuse. This paper proposes a new parallel speculative execution model where the idle speculative cores execute scout threads for reducing cache miss penalties. The scout thread is based on value prediction, and can handle an instruction region which accesses the addresses with several strides. It also can reduce execution cycles by raising computation reuse ratio. The result of the experiment with SPEC CPU95 FP suite benchmarks shows that the new hybrid model of parallel speculative execution and scout threading improves the maximum speedup from 40.6% to 41.3%, and the average speedup from 15.0% to 19.1%.**

I. Introduction

So far, various speed-up techniques for microprocessors have been proposed. The performance of microprocessors had been controlled by the gate latencies, and it had been relatively easy to speed-up microprocessors by transistor scaling. However, the interconnect delay has been going major, and it has become difficult to achieve speed-up only by higher clock frequency. Therefore, speed-up techniques based on ILP (Instruction-Level Parallelism), such as super-scalar or SIMD instruction sets, have been counted on.

Recently, multi-core processors equipped with two or more cores attract a great deal of attention. They are now in wide use from generic processors for PCs to embedded processors [1]. The UltraSPARC T2 [2] with eight cores, and the TILE64 [3] with 64 cores is available now, and many core processors such as the TILE-Gx processor [4] with 100 cores is planned to be shipped.

A program generally forms a poset, or a lattice. It has a length along time axis, and has a width (i.e. parallelism) orthogonal to time axis. Traditional speed-up techniques mentioned above are all based on some parallelism in different granularity. In other words, their approaches aim to increase performance by shrinking the width of the program lattice.

On the other hand, we have proposed an auto-memoization processor based on computation reuse [5], [6], [7]. In contrast to traditional speed-up techniques for microprocessors, memoization or computation reuse tries to shrink the vertical length of the program lattice. As a speedup technique, memoization has no relation to parallelism of programs. It depends upon value locality, especially input values of functions or loops. Therefore, memoization has a potential for breaking through the stone wall, against which the speedup techniques based on ILP have come up.

We also have proposed a model called parallel speculative execution. It predicts the inputs for a reusable loop iteration, and additional shadow cores execute the iteration speculatively. The shadow cores register the results of the speculative executions onto the reuse table. If the value prediction for inputs was correct, the registered outputs can be reused by the main core and execution time will be reduced.

However, only the loops whose input values change monotonously can benefit from parallel speculative execution, and the speculative cores are idle without such loops. In this paper, we propose a new hybrid model of traditional parallel speculative execution and scout threading [8] for the auto-memoization processor. While being idle, the speculative cores execute scout threads and try to conceal memory access latencies without disturbing computation reuse.

II. Related Works

Studies for extracting ILPs with speculative executions based on value prediction have been proposed by Lipasti et. al. [9] or Wang et. al. [10] Many speculative multi-threading (SpMT) models also have been proposed. They have multiple processors or cores, and run threads speculatively using predicted value sets. In an SpMT model, a speculative thread will generally squashed when its input values are overwritten by main thread.

Roth et. al. [11] has proposed *register integration*. It is a mechanism for reusing the results of squashed instructions by writing back the past register mapping. It is shown that the model can provide performance improvements of up to 11.5%.

Some hybrid methods of computation reuse and value prediction have been studied. Wu et. al. [12] have proposed a speculative multi-threading supported by computation reuse.

In the model, the compiler identifies computation region for reuse or value prediction. At runtime, if a region cannot be reused, the processor predicts the outputs of the region, and speculatively execute its following instructions using the predicted values. Hence, if the value prediction fails, the speculative executions should be squashed, and it costs additional hardware and overhead for the squash.

Molina et. al. [13] have proposed a combination model of speculative thread and non-speculative thread. The execution results of speculative thread are stored into the FIFO called a *look ahead buffer*, and non-speculative thread picks up instructions from the FIFO. If current source operands and the stored operands are same, the non-speculative thread reuses the execution results and skips execution.

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

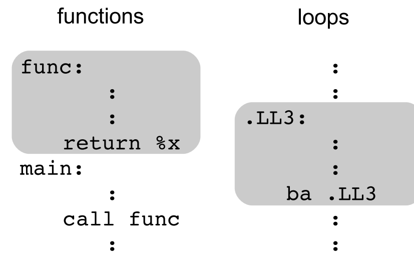

Fig. 1. Memoizable instruction regions.

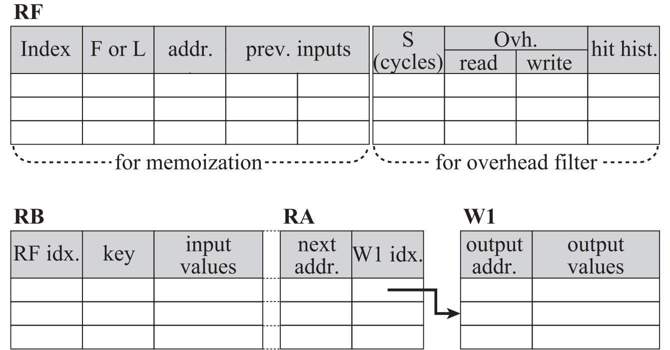

Fig. 2. Structure of MemoTbl

III. Research Background

In this section, we describe about an auto-memoization processor and a parallel speculative execution model as the background of our study.

A. Auto-Memoization Processor

Computation reuse is a well-known speed-up technique in the software field. It is storing the input sequences and the results of some computation regions, such as functions, for later reuse and avoiding recomputing them when the current input sequence matches one of the past input sequences. It is called **memoization** [14] to apply computation reuse to computation regions in programs.

Memoization is originally a programming technique for speed-up, and brings good results on expensive functions. However, it requires rewrite of target programs, and the traditional load-modules or binaries cannot benefit from memoization. Furthermore, the effectiveness of memoization is influenced much by programming styles. Rewriting programs using memoization occasionally makes the programs slower. Memoization costs a certain overhead because it is implemented by software.

On the other hand, the auto-memoization processor, which we have proposed, makes traditional load-modules faster without any software assist. There is no need to rewrite or recompile target programs. The processor detects functions and loop iterations as reusable regions dynamically, and memoizes them automatically.

Fig. 1 shows the two types of memoizable instruction regions. A region between a callee label and its associated `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.

The auto-memoization processor consists of the memoization engine, **MemoTbl** and **MemoBuf**. The MemoTbl is a set of tables for storing input/output sequences of past executed computation regions. The MemoBuf works as a write buffer for MemoTbl. The brief structure of MemoTbl is shown in Fig.2.

Entering to a memoizable region, the processor refers to MemoTbl and compares current input set with former input sets which are stored in MemoTbl. If the current input set matches with one of the stored input sets on MemoTbl, the memoization engine writes back the stored outputs, which are associated with the input set, to cache and registers. This omits the execution of the region and reduce the total execution cycles.

If the current input set does not match with any past input sets on MemoTbl, the processor stores the inputs and the outputs of the region into MemoBuf while executing the region. The input set consists of the register/memory values which are read in the region; the output set consists of the values which are written in the region and return value of function. Reaching the end of the region, the memoization engine stores the content of MemoBuf into MemoTbl for future reuse.

The MemoTbl consists of four tables. They are

RF: for start addresses of instruction regions.
RB: for input data sets of instruction regions.
RA: for input address sets of instruction regions.
W1: for output data sets of instruction regions.

The RF, RA, and W1 are implemented with RAM. On the other hand, the RB is implemented with CAM (Content Addressable Memory) array, so that input matching can be done fast by associative search.

Each RF line corresponds to a reusable computation region. One RF line has two groups of fields, the one is for computation reuse and the other is for the overhead filter which will be explained later in III-C. The fields for computation reuse store whether the region is a function or a loop (*F or L*), the start address of the region (*addr.*), and previous two input sets for predicting next input set for parallel speculative execution (*prev. inputs*). The fields for overhead filter store the execution cycles of the region (*S*), its previous reuse overhead (*Ovh*), and its hit/miss pattern (*hit hist.*).

The brief execution mechanism of the auto-memoization processor is as follows. When the auto-memoization processor detects a function or a loop iteration, it first searches its start address through the RF table for deciding the inputs of the reusable region are stored or not. Then, the input matching for computation reuse starts.

The processor reads the value of program counter and registers, and searches their values from the RB. If one of the RB lines matches, the processor gets its index and reads RA using the index. The RA line has the address for the input of the region which should be tested next. Next, the processor gets input value from the cache or main memory using the address, searches the input value through the RB again, and repeats it. If all inputs of a reusable block have matched with one of the stored input set on the MemoTbl, the processor can get the output set from W1 by using the index for W1 (called 'W1 pointer') stored in the terminal RA entry. The detail of this execution mechanism is shown in [5], [6].

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 cycles. We call these two kinds of overheads '**reuse overheads**.'

B. Parallel Speculative Execution

As a matter of course, memoization can omit the execution of an instruction region only if the current input values for the region match completely with the input values which are used in former execution. Hence, a loop iteration whose inputs include its iterator variable never benefits 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 [15]. 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 cross-thread dependence violation and thread squash. We installed some SpMT cores called **SpC** (speculative cores) to our auto-memoization processor. These cores help the unsuitable regions for memoization mentioned above. Fig.3 shows the structure of the auto-memoization processor with three SpCs.

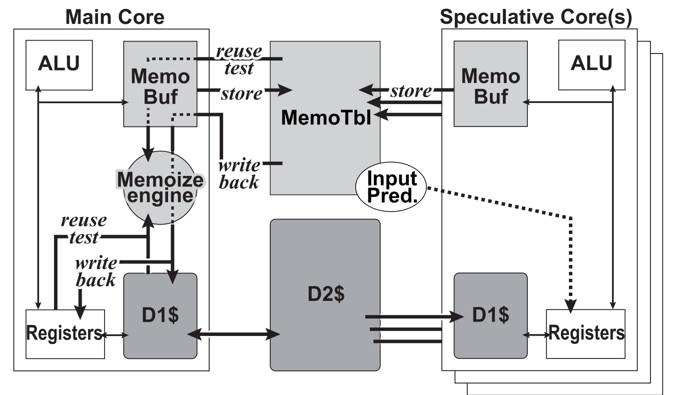

Fig. 3. Structure of a parallel speculative execution.

Each SpC has its own MemoBuf and a first-level data cache. The second-level data cache and MemoTbl is shared between all cores. While the main core executes a memoizable computation region, SpCs execute the same region using predicted inputs, and stores the results into MemoTbl. The inputs are predicted by stride prediction using the last two input sets stored in RF *prev. inputs* field. If the input prediction was correct, the main core can omit intended execution by reusing the result of SpC. Unlike as other SpMT methods, even if the input speculation proves to be incorrect later, the main core need not to pay a cost for any back-out management, it only fails reuse test and executes the region as usual. This extension can omit the execution of instruction regions whose inputs show monotonous increase/decrease. These SpCs not only omit some executions, but also work as a cache prefetch technique [16], [17].

C. Overhead Filter

For some reusable regions, reuse overhead may outweigh the eliminated execution cycles by reuse. This will go for some regions which have many input values to be tested, and all tiny regions. Hence, the auto-memoization processor should estimate the effect of reuse, and avoid memoizing unsuitable instruction regions. This can reduce useless input matching and contribute to good performance.

For the reusability estimation, we installed a small logic onto MemoTbl. This logic estimates how much cycles will be reduced by memoizing a block, and how much overhead will cost for its reuse. It is important how to decide which instruction region should be suitable for parallel speculative execution. When the results of speculative executions for an instruction region are frequently reused, the instruction region is supposed to be suitable. Shift registers, shown as *hit hist.* fields in Fig.2, are used for recording these reuse frequency. The reuse overhead of an instruction region can be calculated from these frequency values.

Assume that M represents the number of successful reuses about a certain region for recent T times tries ($0 \leq M \leq T$). The value of M can be gotten from *hit hist.* field in RF. With

the execution cycles S of the region, which can be also got from RF, the reduced cycle can be represented as

$$M \cdot (S - Ovh^R - Ovh^W) \quad (1)$$

where Ovh^R and Ovh^W represent search/writeback overheads for the region respectively.

Ovh^R also costs when input matching fails and reuse cannot be applied. This overhead can be calculated as follows.

$$(T - M) \cdot Ovh^R \quad (2)$$

Here, if the *loss* (2) is larger than the *gain* (1), the computation region cannot be suitable for reuse. Now, we define the difference between (1) and (2) as $Gain$ (3). An additional small logic calculates whether (3) goes positive or negative, and decides the suitability of computation regions.

$$Gain = M \cdot (S - Ovh^W) - T \cdot Ovh^R \quad (3)$$

IV. Scout Threading by Speculative Cores

In this section, we will propose a new parallel speculative execution model, where the SpCs execute scout threads.

A. Outline

On the traditional auto-memoization processor, the overhead filter described in III-C stops registering input/output values of an instruction region, if applying computation reuse to the region will deteriorate the whole performance. The previous input values for value prediction, which are shown as *prev. inputs* in Fig. 2, will also not be updated for such worthless regions.

The *prev. inputs* for value prediction are initialized before the associated region is executed, and updated when the input values of the region are registered from MemoBuf to MemoTbl at the end of the region. The value predictor, which is shown as *Input Pred.* in Fig. 3, reads these previous input values, predicts next input values from them by stride prediction, and passes the predicted values to SpCs. Now, if the overhead filter prohibits registering input/output values onto MemoTbl, *prev. inputs* are not updated and the value predictor cannot generate correct input values. When value prediction fails, SpCs cannot issue speculative execution. We call this that SpCs are *idle*.

In this paper, we propose a new parallel speculative execution model, in which SpCs execute some scout threads for cache prefetching when they are idle. This hybrid model of parallel speculative execution and scout threading can conceal some memory access latencies of the main core and reduce execution cycles. At input-predictable instruction regions, SpCs issue parallel speculative execution. On the other hand, the input values are not predictable, the instruction region will not benefit from memoization, and SpCs execute scout threads.

Scout threading not only reduces some cache miss penalties but also can raise the hit-rate of computation reuse. The parallel speculative execution typically brings cache misses because it precedes the main core execution. The cache miss delays registering the result of parallel speculative executions

```
186 C
187       DO 600 I3=2,N-1
188       DO 600 I2=2,N-1
189       DO 600 I1=2,N-1
190 600 R(I1,I2,I3)=V(I1,I2,I3)
191  >   -A(0)*(U(I1,   I2,   I3  ))
192  >   -A(1)*(U(I1-1,I2,   I3  ) + U(I1+1,I2,   I3  )
193  >         + U(I1,   I2-1,I3  ) + U(I1,   I2+1,I3  )
194  >         + U(I1,   I2,   I3-1) + U(I1,   I2,   I3+1))
195  >   -A(2)*(U(I1-1,I2-1,I3  ) + U(I1+1,I2-1,I3  )
196  >         + U(I1-1,I2+1,I3  ) + U(I1+1,I2+1,I3  )
197  >         + U(I1,   I2-1,I3-1) + U(I1,   I2+1,I3-1)
198  >         + U(I1,   I2-1,I3+1) + U(I1,   I2+1,I3+1)
199  >         + U(I1-1,I2,   I3-1) + U(I1-1,I2,   I3+1)
200  >         + U(I1+1,I2,   I3-1) + U(I1+1,I2,   I3+1))
201  >   -A(3)*(U(I1-1,I2-1,I3-1) + U(I1+1,I2-1,I3-1)
202  >         + U(I1-1,I2+1,I3-1) + U(I1+1,I2+1,I3-1)
203  >         + U(I1-1,I2-1,I3+1) + U(I1+1,I2-1,I3+1)
204  >         + U(I1-1,I2+1,I3+1) + U(I1+1,I2+1,I3+1))
205 C
```

Fig. 4. A part of 107.mgrid program code.

onto MemoTbl, and this leads to the low reuse hit-rate of *input-predictable* regions because the result registration by SpCs may be too late for reuse test by the main core. Now, consider that there is a loop *A*, which is not suitable for computation reuse, and *A* contains another loop *B* in it and the loop *B* is suitable for reuse. SpCs in the new hybrid model will execute scout threads for the outer loop *A*, and will issue parallel speculative execution for the inner loop *B*. This can reduce cache misses at the inner loop *B* and may raise the reuse hit-rate for the region *B*.

Fig. 4 shows a part of the program code of 107.mgrid from SPEC CPU95 FP benchmark suite. The code calculates $R = V - AU$, where R, V, U are 3-dimensional arrays. On the traditional parallel speculative execution model, the innermost loop at line 189 benefits from parallel speculative execution and computation reuse, but the outer two loops at line 187 and 188 do not, because they have too many input values. In this case, SpCs will execute scout threads for the outer loops, and it will lead to higher reuse hit-rate for the innermost loop, because the cache misses with first some iterations of the innermost loop will reduced by the scout threading.

B. Execution Model

For scout threading, SpCs should be able to execute the instruction region speculatively. Input values for the speculative execution are available if value prediction succeeds. Consequently, higher the hit-rate of value prediction is, more times SpCs can execute speculatively. As mentioned in III, value prediction depends on *prev inputs* fields in RF. However, on the traditional parallel speculative execution model, when the overhead filter detects that the *Gain* shown in (3) is negative, the processor stops not only applying computation reuse to the region, but also updating *prev inputs* fields.

On the new hybrid model, the processor does not stop updating *prev inputs* fields even if *Gain* is negative. This will increase the opportunities of speculative execution. However, the result of speculative execution should not to be stored onto MemoTbl, because *Gain* is negative and the result of the instruction region will not be suitable for reuse. Hence, the processor does not issue parallel speculative execution when *Gain* is negative, but executes a kind of scout thread. The scout thread is composed of *load* instructions in the instruction region. This can avoid storing useless entries onto MemoTbl, and leads to high cache hit-rate without disturbing computation reuse.

Scout thread includes only *load* instructions, and it will consume not so much energy. The targets of scout threading are the instruction regions, which are memoizable and will not benefit from computation reuse. Hence, running time of scout threads will be short and it will cause little additional energy consumption.

V. Implementation

This section describes an implementation of the new hybrid model.

A. Switching between Speculative Execution and Scout Threading

As mentioned in the previous section, *prev inputs* field in RF keeps being updated, even if *Gain* is negative and the instruction region is not suitable for computation reuse. Now, the processor needs to select whether SpCs should issue parallel speculative execution or execute scout threads for the region. This decision depends on whether the value of *Gain* is positive or negative. A 1-bit-width field is installed onto each RF line, and this field holds the previous result of $Gain <> 0$ for the associated instruction region. SpC dynamically select which should be issued, parallel speculative execution or scout threading, by checking the field.

Through parallel speculative execution, the processor needs to store input/output values onto MemoBuf and MemoTbl, and all units in SpCs should work. On the other hand, through scout threading, only units for prefetching should work. Consequently, SpCs' behavior should vary depending on whether parallel speculative execution or scout threading is now active. Therefore, 1-bit flag is installed to each SpC, and it manages the current execution mode of associated SpC.

B. How to Generate Scout Threads

For scout threading, each SpC should pick up *load* instructions from the instruction region, and execute only them. The auto-memoization processor is based on SPARC-V8, and it has fixed 32-bit width SPARC ISA, and the opcode has fixed 8-bit width. The SPARC ISA has many types of *load* instructions; five *load floating-point* instructions, three *load floating-point from alternate space* instructions, eight *load integer* instructions and eight *load integer from alternate space* instructions [18]. However, the opcodes of them are all in `11xxx0xx` format, and they can be easily distinguished from other instructions.

```
                :
1c1d4:  sethi  %hi(0x1d000), %o1
1c1d8:  sll  %l1, 2, %l0
1c1dc:  inc  %l1
1c1e0:  ld  [ %o1 ], %f3
1c1e4:  fdivs  %f3, %f4, %f2
1c1e8:  fstod  %f2, %f2
1c1ec:  fadds  %f4, %f4, %f4
1c1f0:  std  %f2, [ %fp + -8 ]
1c1f4:  ldd  [ %fp + -8 ], %o2
1c1f8:  mov  %o3, %o1
1c1fc:  st  %f4, [ %fp + -116 ]
1c200:  call  1c05c
1c204:  mov  %o2, %o0
1c208:  cmp  %l1, 0x19
1c20c:  sethi  %hi(0x1d000), %o3
1c210:  fadds  %f0, %f0, %f0
1c214:  ld  [ %o3 + 8 ], %f2
1c218:  fdivs  %f2, %f0, %f0
1c21c:  st  %f0, [ %l2 + %l0 ]
1c220:  ble  1c1d4
                :
```

Fig. 5. A part of FFT assembly code.

The processor recognizes all types of *load* instructions and *save/restore* instructions by instruction decoder, and issues them. In SPARC binaries, there is a *save* instruction at the beginning of a function, and a *restore* instruction at the end of a function. On the SPARC architectures, register windows are managed with *save/restore* instructions. To guarantee correct function call behavior, the new hybrid model is designed to issue not only *load*s but also *save* and *restore*.

On the traditional model, nest structures of loops are kept in MemoBuf and parallel speculative executions are issued for each nested loop. Fig. 5 shows a part of FFT benchmark program from Stanford benchmark suite. With this program, the loop from `1c1d4` to `1c220` will be applied parallel speculative execution. The function at `1c05c` is also applied parallel speculative execution because it is called at `1c200`.

On the other hand, through scout threading on the new hybrid model, function calls are ignored. For example in Fig. 5, the input values for the function at `1c05c` cannot be defined correctly without executing from `1c1d4` to `1c200`. Another case is that a branch instruction controls whether a function is called or not. Consequently, almost all instructions in an instruction region should be executed for correctly handling function calls inside the region. This will increase latencies of scout threads and power consumption by scout threading.

VI. Performance Evaluation

A. Simulation Environments

We have developed a single-issue SPARC-V8 processor simulator with auto-memoization structures and SpCs, which

TABLE I
SIMULATION PARAMETERS

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

can issue both speculative execution and scout threading. This section discusses the performance of the new model proposed in this paper. The simulation parameters are shown in TABLE I. The cache structure and the instruction latencies are based on SPARC64 processors [19]. The on-chip CAM for RB in MemoTbl is modeled on MOSAID DC18288 [20]. The latencies of the CAM are defined on the assumption that the clock of the processor is 10-times faster than the CAM.

B. Results with SPEC CPU95 FP

We evaluated the hybrid model by parallel speculative execution and scout threading. Workloads are all benchmark programs in SPEC CPU95 FP suites and are executed with 'train' dataset. All benchmark programs are compiled by gcc version 3.0.2 with '-msupersparc -O2' options, and linked statically.

The evaluation results are shown in Fig. 6. We have evaluated following three models,

(M) No-memoization model
(P) Traditional model of parallel speculative execution
(S) Hybrid model parallel speculative execution and scout threading

and Fig. 6 shows the execution cycles of these models. Each bar is normalized to the number of executed cycles of (M) the model without memoization.

The legend in Fig. 6 shows the breakdown items of total cycles. They represent the executed instruction cycles ('**exec**'), the comparison overhead between CAM and the registers ('**test(r)**'), the comparison overhead between CAM and the caches ('**test(m)**'), the writeback overhead ('**write**'), the first-level and shared second-level data cache miss penalties ('**D$1**', '**D$2**'), and the register window miss penalty ('**window**') respectively.

As we can see in Fig. 6, (S) the new hybrid model is rewarded with good results. For 102.swim, 104.hydro2d, 125.turb3d and 146.wave5, *D$1* increases and *D$2* decreases. This means that the hit-rate of shared second-level data cache for the main core increases due to the scout threading by SpCs. The reduction of cache miss penalty cycles of the second-level data cache for these four workloads are shown in TABLE II. On the other hand, *exec*, *test(r)*, *test(m)* and *write* for these four workloads do not differ from them of (P). This means that scout threading does not disturb computation reuse. The average of reduced cache miss penalty cycles for all SPEC95 FP workloads increases from 15.0% to 19.1%.

TABLE II
RATIO OF REDUCED D2 CACHE MISS PENALTIES

	(M) Traditional	(S) Proposed
102.swim	55.1%	96.9%
104.hydro2d	16.7%	56.4%
125.turb3d	12.4%	67.1%
146.wave5	13.0%	42.6%
Ave. of 4 programs	24.3%	65.8%

For 107.mgrid, 141.apsi and 145.fpppp, we cannot see the benefit of new hybrid model, because there occurred little cache misses. However, notice that *exec* is slightly reduced for 107.mgrid. This means that the hit-rate of computation reuse increased. As mentioned in IV-A, this should occur in some inner instruction regions of nested structures. We verified that the hit-rate of computation reuse for 107.mgrid increases 1.9% on the new hybrid model.

Now, let us roughly discuss the energy consumption of the new hybrid model. On the traditional model (P), the running time of SpCs is about 23% of whole execution cycles. On the new hybrid model (S), the running time for scout threads is about 26% besides it. However, for executing scout threads, not all units in SpC, but only clock and prefetching unit should work, and this will not lead to much increase of energy consumption. We estimated the energy consumption of the new hybrid model, and it is found that the increase is about 13% over the traditional model (P).

In conclusion, the performance of the new hybrid model (S) is better than the traditional model (P) as a whole. The model (S) improves the maximum speedup from 40.6% to 41.3%, and the average from 15.0% to 19.1%.

VII. CONCLUSIONS

In this paper, we have proposed a hybrid model of traditional parallel speculative execution and scout threading for auto-memoization processor. In the model, idle SpCs execute scout threads and conceal some of memory access latencies without disturbing computation reuse. This prefetching sometimes also raises the hit-rate of computation reuse.

Through an evaluation with SPEC CPU95 FP suite benchmark programs, it is found that the new hybrid model improves the maximum speedup ratio from 40.6% to 41.3%, and the average speedup ratio from 15.0% to 19.1%. The hit-rate of computation reuse also rises a little with 107.mgrid.

One of the our future works is merging this model with other low-overhead models such as [6] we had proposed.

ACKNOWLEDGMENT

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

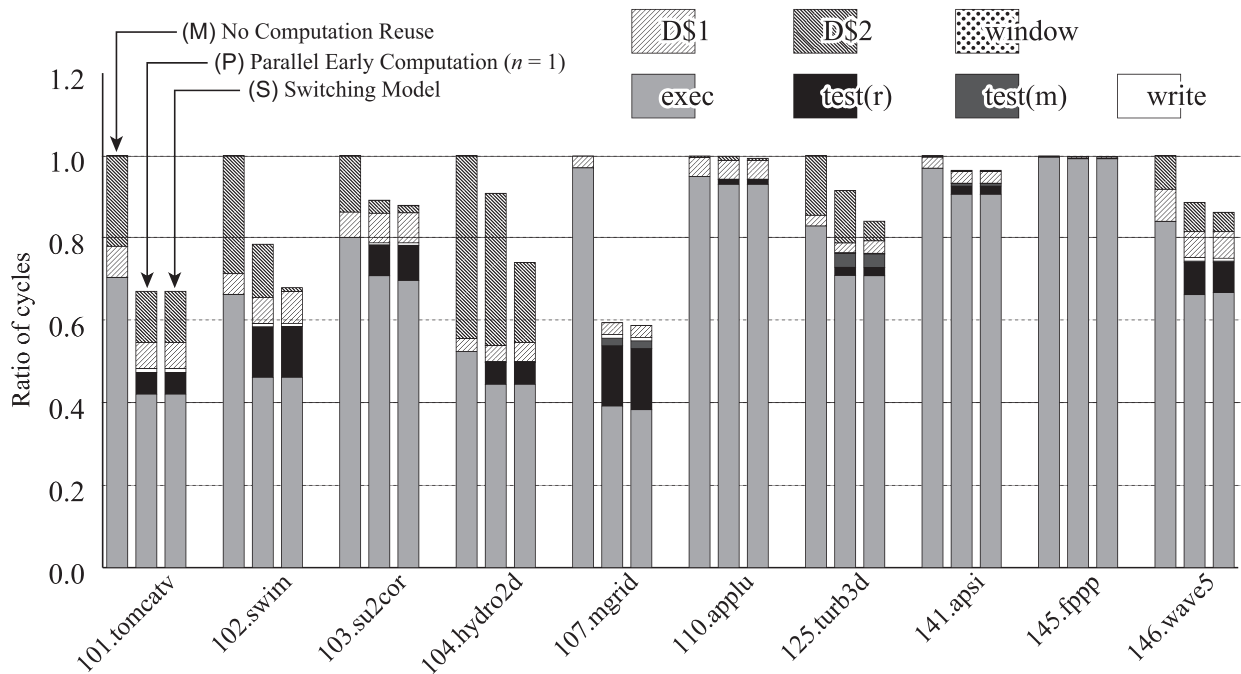

Fig. 6. Ratio of execution cycles (SPEC CPU95 FP).

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

[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] Y. Kamiya, T. Tsumura, H. Matsuo, and Y. Nakashima, "A Speculative Technique for Auto-Memoization Processor with Multithreading," in *Proc. 10th Int'l. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, Dec. 2009, pp. 160–166.

[7] T. Ikegaya, T. Tsumura, H. Matsuo, and Y. Nakashima, "A Speed-up Technique for an Auto-Memoization Processor by Collectively Reusing Continuous Iterations," in *Proc. 1st Int'l. Conf. on Networking and Computing (ICNC'10)*, Nov. 2010, pp. 63–70.

[8] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-Performance Throughput Computing," *IEEE Micro*, vol. 25, pp. 32–45, May. 2005.

[9] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *29th MICRO*, Dec. 1996, pp. 226–237.

[10] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *30th MICRO*, Dec. 1997, pp. 281–290.

[11] A. Roth and G. S. Sohi, "Register integration: A simple and efficient implementation of squash reuse," in *33rd MICRO*, Dec. 2000.

[12] Y. Wu, D. Chen, and J. Fang, "Better exploration of region-level value locality with integrated computation reuse and value prediction," in *28th ISCA*, 2001, pp. 98–108.

[13] C. Molina, A. González, and J. Tubella, "Trace-level speculative multithreaded architecture," in *ICCD*, 2002.

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

[15] 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.

[16] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen, "Speculative precomputation on chip multiprocessors," in *Proc. of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation (METAC-6)*, 2002.

[17] I. Ganusov and M. Burtscher, "Future execution: A hardware prefetching technique for chip multiprocessors," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, 2005, pp. 350–360.

[18] D. L. Weaver and T. Germond, Eds., *The SPARC Architecture Manual Version 9*. Prentice-Hall, Inc., 1994.

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

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