A Speed-up Technique for an Auto-Memoization Processor by Collectively Reusing Continuous Iterations

Tomoki IKEGAYA*, Tomoaki TSUMURA*, Hiroshi MATSUO* and Yasuhiko NAKASHIMA†

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

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

***Abstract*—We have proposed an auto-memoization processor based on computation reuse, and merged it with speculative multithreading based on value prediction into a parallel early computation. In the past model, the parallel early computation detects each iteration of loops as a reusable block. This paper proposes a new parallel early computation model, which integrates multiple continuous iterations into a reusable block automatically and dynamically without modifing executable binaries. We also propose a model for automatically detecting how many iterations should be integrated into one reusable block. Our model reduces the overhead of computation reuse, and further exploits reuse tables. The result of the experiment with SPEC CPU95 FP suite benchmarks shows that the new model improves the maximum speedup from 40.5% to 57.6%, and the average speedup from 15.0% to 26.0%.**

I. Introduction

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

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 are available now, and many core processors such as the TILE-Gx processor[4] with 100 cores are 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 parallelisms in different granularities. In other words, their approaches aim to increase performance by shrinking the width of the program lattice.

On the other hand, we have proposed an auto-memoization processor based on computation reuse.[5], [6] In contrast to traditional speed-up techniques for microprocessors, memoization or computation reuse tries to shrink the length of the program lattice. As a speedup technique, memoization has no relation to parallelism of programs. It depends upon value locality, especially input values of functions. Therefore, memoization has a potential for breaking through the stone wall, against which the speedup techniques based on ILP have been 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 is correct, the registered outputs can be reused by the main core and execution time will be reduced.

In this paper, we propose a new parallel speculative execution model which manages some continuous loop iterations as one reusable computation region. Merging continuous iterations into one reusable region can make reuse overhead lower and reuse hit-rate higher without any binary modification. How many continuous iterations should be merged depends on the characteristics of programs and loops. Hence, we also propose an algorithm for deciding the number of iterations.

II. Related Works

Studies for extracting ILPs with speculative executions based on value prediction have been proposed by Lipasti et. al.[7] or Wang et. al.[8] 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.[9] 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.[10] have proposed a speculative multithreading 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 outpus of the region, and speculatively execute its following region 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.[11] 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 against [10]. 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 [11] and our model are of the same sort. However, our model can reuse some computation regions which require memory read as inputs.

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**[12] 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 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.

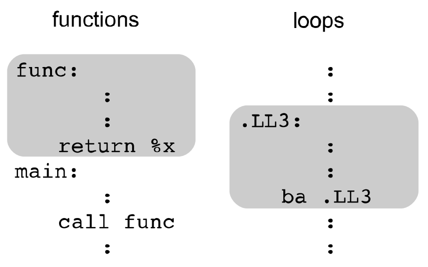

Fig. 1. Memoizable instruction regions.

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.

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 the memoizable region, the processor refers to the 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 the MemoTbl, the memoization engine writes back the stored outputs associated with the input set to cache and registers. This omits the execution of the region and reduce the total execution time.

If the current input set does not match with any past input sets, the processor stores the inputs and the outputs of the region into MemoBuf while executing the region as usual. The input set consists of the register/memory values which are read in the region, and the output set consists of the values which are written 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

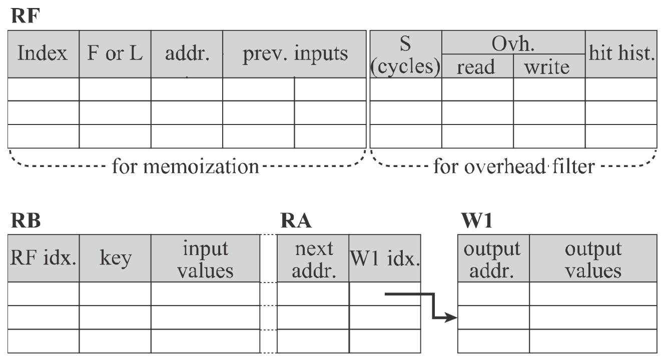

Fig. 2. Structure of MemoTbl

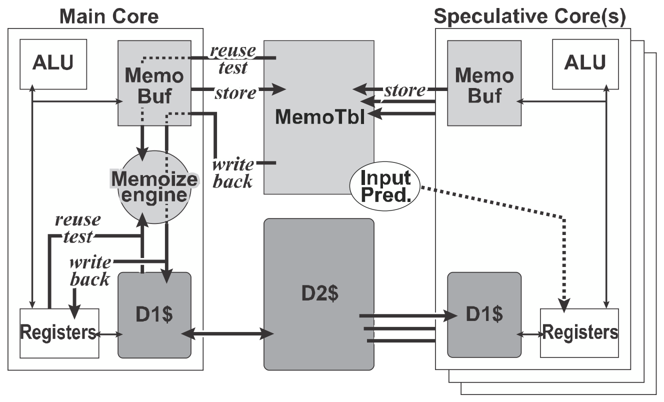

Fig. 3. Structure of a parallel speculative execution.

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 past reuse overhead (*Ovh*), and its past hit rate (*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 computatin 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 the line 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 so on. 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**.' For some reusable blocks, the reuse overhead may outweigh the eliminated execution cycles by reuse. This will go for some blocks which have many input values to be tested, and all tiny blocks.

B. 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, 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[13]. 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 SpCs.

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. If the input prediction was correct, the main core can omit intended execution by reusing the result of SpC. Unlike as traditional speculative executions, even if the input speculation proves to be incorrect later, the main core need not to pay a cost for any backout management. 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[14], [15].

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 memoing unsuitable instruction regions. This can reduce useless input matching and contribute to good performance.

For the reusability estimation, we installed a small logic to MemoTbl. This logic estimates how much cycles will be reduced by memoing 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. 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 actual cycle *gain* can be represented as

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

where Ovh^R and Ovh^W represent search/write-back 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.

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

Here, if the *loss* (2) is larger than the actual *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.

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

IV. Reusing Continuous Iterations Collectively

In this section, we will propose a new parallel speculative execution model for managing loop iterations more effectively.

A. Outline

As mentioned in the previous section, the parallel speculative execution works effectively for loops whose inputs change monotonously. However, the effectiveness will be limited when the loop body is not so heavy or the loop has numerous iterations.

When the loop body is not so heavy, little speed-up can be gained by reuse, because almost all omitted execution cycles will be offset by its reuse overhead. When the number of iterations is large, many parallel speculative executions will be committed by SpCs and the many results of speculative executions will be stored into MemoTbl. They will use many lines in MemoTbl, and useful entries which would be reused later may be overwritten or flushed. It will lead to low hit rates of computation reuse.

Therefore, we propose a new model of parallel speculative execution for solving these two problems. It is managing multiple continuous loop iterations as one reusable region. Why the new model can solve the problems will be described below with an example code partly from 103.su2cor in SPEC95 CPU suite. The code is shown in Fig. 4.

For the `DO` loop in Fig. 4 as a reusable region, the inputs which should be stored in MemoTbl are the iterator `I`, and the variables `NDIM`, `LSIZE(I)`, `MOD(LSIZE(I),2)`, `NPTS` and `LVEC`. On the other hand, the outputs are the iterator

```
         :
       NPTS=1
       LVEC=1
C
       DO 10 I=1,NDIM
       IF(MOD(LSIZE(I),2) .NE. 0) STOP
       NVOL=NPTS
       NPTS=NPTS*LSIZE(I)
       LHALF(I)=LSIZE(I)/2
       LVEC=LVEC*LHALF(I)
10     CONTINUE
         :
```

Fig. 4. A part of 103.su2cor program code.

`I`, and the variables `NVOL`, `NPTS`, `LHALF(I)` and `LVEC`. The traditional auto-memoization processor tests whether one register input (the iterator `I`) and the five memory variables match to one of the past input sets or not. Then, if the computation reuse succeeds, the iterator `I` and four memory variables will be written back to the registers and the caches.

Now, assume that every two continuous iterations of the `DO` loop are taken as one reusable computation region. This leads to a similar effect of merging two iterations into one iteration by loop unwinding. After unwinding, one iteration has eight inputs which are the iterator `I`, `NDIM`, `LSIZE(I)`, `MOD(LSIZE(I),2)`, `NPTS`, `LVEC`, `LSIZE(I+1)` and `MOD(LSIZE(I+1),2)`. Only two more variables should be added than before. Likewise, one iteration has now six outputs which are `I`, `NVOL`, `NPTS`, `LHALF(I)`, `LHALF(I+1)` and `LVEC`. Hence, only one more output should be added.

That is, `NDIM` $\times$ 6 variables must have be tested in total through the whole `DO` loop execution with traditional model, where `NDIM` is the number of all iterations. However, when managing two continuous iterations as one reusable computation region, the number of variables which should be searched is `NDIM` $\div$ 2 $\times$ 8. As a result, the total search overhead for computation reuse will be reduced to $2/3$. The total write back overhead for computation reuse will also be reduced to $3/5$.

Now, the hit rate of the stride prediction for inputs of monotonous loops cannot be influenced by this loop unwinding, although the stride value will be a multiple of original. Hence, the overheads of computation reuse can be reduced without degrading performance. The MemoTbl usage will be also reduced by this loop unwinding, and the total hit rate of reuse for all reusable computation regions will rise.

The code shown in Fig. 4 is an ideal example whose inputs and outputs can be reduced much. However, even in the worst case, the search overhead for at least one input, the iterator variable, can be reduced. Hence, any loop or any program can benefit from this mechanism.

B. Unwinding Degree and Performance

To implement the unwinding-like model mentioned above, it is needed to install simple counters for the parallel speculative

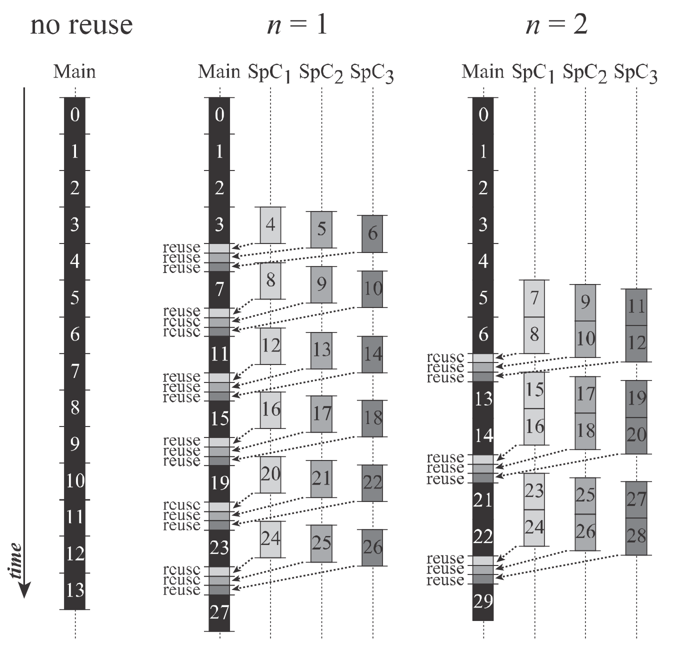

Fig. 5. Execution models with three SpCs.

execution processor. Each RF line has one counter, and it counts the number of iterations for the associated reusable loop dynamically. In the case of managing n iterations as one reusable computation region, when the value of a counter reaches n, the processor registers inputs/outputs set from MemoBuf into MemoTbl and tries to apply computation reuse to the associated loop.

Now, it is very important how many iterations n should be merged into one reusable computation region. Hereafter, we call this n an **unwinding degree**. In the example code shown in Fig. 4, the number of total input variables which need to be tested through the execution of the DO loop can be calculated as $(N \div n)(4 + 2n) = N(2 + 4/n)$, where N is the total number of iterations of the loop. Hence, it seems that the reuse overhead will be reduced much as n goes larger. However, there are some drawbacks in increasing the unwinding degree n.

One of the drawbacks is that more iterations should be executed for stride prediction of input values than the traditional model. For example, assume a loop whose iterator variable i is initiated with 0 and incremented by stride 1. The execution models of this loop with three SpC are shown in Fig. 5. The parallel speculative execution system finds a backward branch at the end of the loop at $i = 0$, and recognizes the loop as a reusable computation region. After executing the iterations of $i = 1$ and $i = 2$, the parallel speculative execution calculates the strides for input values prediction using the past input values of $i = 1, 2$. Given the strides, SpCs execute the iterations of $i = 4, 5, 6$ using the input values predicted with the strides. When the input prediction is correct, the main core can omit the execution of the iteration $i = 4$, which is shown in the middle of Fig. 5.

On the other hand, the execution model of merging two continuous iterations into one reusable computation region is shown in the right part of Fig. 5. The parallel speculative execution system can only calculate the input value strides after the main core executes the iterations of $i = \{0, 1\}, \{2, 3\}$. Therefore, three SpCs execute the iterations of $i = \{7, 8\}, \{9, 10\}, \{11, 12\}$ speculatively while the main core executes the iterations of $i = \{5, 6\}$, and the first iteration which the main core can skip is of $i = 7$. The iterations of $i = 4, \cdots, 6$ which could be omitted with the traditional model should be executed normally. In general, when merging n continuous iterations into one reusable computation region, the execution of the iterations of $i = 4, \cdots, 3n$ cannot be omitted by computation reuse, and the number of these unfortunate iterations grows in proportion to n.

Another drawback is that the parallel speculative execution by SpCs would not complete before the reuse tests by the main core in more situations than the traditional model. Merging n iterations into one reusable region increases n-fold the execution time of one reusable region. On the other hand, the search/writeback overhead for reusing one region will not amount to n times, as described with the sample code in Fig. 4. The parallel speculative execution by a SpC sometimes delays owing to cache misses, and so on. Hence, when the input value predictions are correct, the main core will catch up with SpCs earlier than in the traditional model. This may lead to low hit rate of the computation reuse because the main core cannot reuse the results of the parallel speculative execution by SpCs.

In conclusion, defining the unwinding degree n for a loop as large a number as possible is not good, and it is very important to find an appropriate value for n. Since the appropriate value for unwinding degree n depends on the amount of processing and the number of input variables of its associated loop, an n for a loop should differ from one for another loop.

V. Implementation

In this section, an implementation for the new parallel speculative execution model shown above will be described.

A. Execution Model

As mentioned in IV-B, how many iterations should be merged into one reusable computation region depends on the characteristics of the associated loop, such as how many inputs it has. Hence, we have added a new field for storing unwinding degree to RF table. Since each entry has the field and is associated with a loop, an appropriate value of unwinding degree for each loop will be stored in the fields. Now, let us see the processes of registering inputs/outputs to MemoTbl and searching them through MemoTbl.

1) Registering to MemoTbl:

Each entry of MemoBuf now has a counter which is initialized by zero. After detecting a loop, the processor counts how many times it reaches the backward branch instruction at the end of the loop, using the counter in the MemoBuf entry which is associated with the loop. The processor keeps storing input/output values into the MemoBuf until the value of the counter comes to the unwinding degree n which is stored in the RF entry associated with the loop.

When the counter value comes to the unwinding degree of the loop, n iterations have been executed, and the input/output values stored in MemoBuf are registered into MemoTbl collectively as a set for one reusable computation region. Then, the counter is reset to zero. Therefore, the processor can manage continuous multiple iterations as one reusable computation region with no hinting or recompiling at all.

As same as in the traditional model[6], the SpCs in this new unwinding model get stride values from RF table, predict input values, and execute speculatively the reusable regions. However, the SpCs in the new model issue the parallel speculative execution only when the associated counter to the target loop is reset.

A backward branch instruction at the end of a loop may be untaken before the associated counter reaches its unwinding degree. This means that the total iteration number of the loop is not divisible by its unwinding degree. In this case, the input/output set currently stored in MemoBuf should be registered into MemoTbl as one entry although the counter does not comes to n.

2) Searching through MemoTbl:

There need no special attention about searching for computation reuse in the new unwinding model. The main core needs not to know about what is the value of the unwinding degree of the loop which the main core is trying to apply computation reuse. In other words, the main core needs not to know how many iterations of the target loop have been managed as one reusable computation region on the MemoTbl. The main core only needs to test the all input values stored in each line of MemoTbl. When the all input values on a MemoTbl line match with current input values, it guarantees that the associated computation region can be reused regardless of how many iterations have been merged into the region.

The output values will be written back to the registers and the caches when a computation reuse succeeds. However, the processor need not to know the unwinding degree n as well. One of the outputs should be the iterator variable, and its value ought to be n larger than the current iterator value. Hence, the execution of n iterations will surely be skipped by only writing back the iterator value in the MemoTbl entry.

B. Dynamic Decision Algorithm for Unwinding Degrees

In general, the appropriate values for unwinding degrees should be different between reusable loops. However, it is almost impossible to decide the appropriate values statically. Hence, a dynamic algorithm for deciding the unwinding degree for each reusable loop is required.

As mentioned in IV-B, increasing an unwinding degree n for a loop has both some advantages and some drawbacks. The major advantages are

- Total number of input values for a loop and the reuse overhead per iteration will be reduced.
- MemoTbl entries will be efficiently used and it will lead to higher hit rate of reuse.

and the major drawbacks are

- The number of iterations which should be executed for stride prediction of input values will increase.
- The execution time of SpC will increase and it will lead to lower hit rate of reuse.

Therefore, the processor needs to find an appropriate value of the unwinding degree for each loop considering these advantages and drawbacks.

Now, each line of RF table stores the execution cycles, the reuse overhead, and the recent hit/miss history of its associated computation region, as described in III-C. Hence, these data can be used for deciding appropriate values for unwinding degrees. The processor computes the effect of computation reuse, and keeps track of the effect as changes the unwinding degree, and selects the appropriate degree which produces good result.

A concrete method is as follows. First, each entry of RF keeps the unwinding degree n of the associated loop. The degree should be represented by 2^k considering the hardware cost for implementation. When a program starts running, each unwinding degree n is initialized by 1 ($k = 0$), and every iteration of all loops will be managed as one reusable computation region as in the traditional model.

Once, a loop iteration is executed and computation reuse is applied, its reduced cycles S by the reuse, the overhead Ovh^R of the reuse test, the overhead Ovh^W of the writeback, and its hit/miss history will be stored in RF. Then, the overhead filter described in III-C calculates the $Gain$ which is defined by (3).

Now in the new unwinding model, the processor calculates the normalized $Gain$ per iteration, which is called $\boldsymbol{UnitGain}$. $UnitGain$ can be calculated as follows

$$UnitGain_k = Gain_k / 2^k \tag{4}$$

where $UnitGain_k$ and $Gain_k$ denote $UnitGain$ and $Gain$ with unwinding degree 2^k respectively, and $UnitGain_0 = Gain_0$. As $UnitGain_k$ can be calculated by simple k-bit arithmetic right shift, only a very small additional hardware should be required for this calculation.

Then, the processor tries to increase the unwinding degree. The unwinding degree n changes to 2 ($k = 1$), and two iterations will be managed as one reusable region after this. After trying computation reuse T times, which is described in III-C, $UnitGain_1$ will be calculated by (3) and (4). Now, if $UnitGain_1$ is larger than $UnitGain_0$, increasing the unwinding degree from $n = 1(k = 0)$ to $n = 2(k = 1)$ must bring good result, and 2 is assumed as more appropriate value for the unwinding degree.

The processor keeps incrementing the value of k and comparing $UnitGain_{k-1}$ and $UnitGain_k$, until the inequality $UnitGain_{k-1} > UnitGain_k$ comes true. When $UnitGain_{k-1} > UnitGain_k$ comes true, the appropiate unwinding degree is decided as 2^{k-1}, and k will be never incremented again.

VI. Performance Evaluation

A. Simulation Environments

We have developed a single-issue SPARC-V8 processor simulator with auto-memoization structures and SpCs based on the new unwinding parallel speculative execution model. This section discusses the performance of this processor. The simulation parameters are shown in TABLE I. The cache structure and the instruction latencies are based on SPARC64 proccessors[16]. The on-chip CAM for RB in MemoTbl is modeled on MOSAID DC18288[17]. The latencies of the CAM are defined on the assumption that the clock of the processor is 10-times faster than the CAM. The T parameter described in III-C is set to 64.

B. Results with SPEC CPU95 FP

We evaluated the new unwinding parallel speculative execution model. 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.

The evaluation results are shown in TABLE II and Fig. 6. We have evaluated following five models,

(N) No-memoization model
(P) Traditional model of parallel speculative execution
(P_2) Reference model with the fixed unwinding degree: 2
(P_4) Reference model with the fixed unwinding degree: 4
(D) Dynamic unwinding model (proposed in this paper)

and Fig. 6 shows the normalized execution cycles of these models. Each bar is normalized to the number of executed cycles of (N) the model without memoizaiton. (P) refers the traditional parallel speculative execution model, and (D) the new unwinding model which is proposed in this paper. (P_2) and (P_4) are reference model which have fixed unwinding degrees 2 or 4. Every model has one main core and three SpCs.

The legend in Fig. 6 shows the breakdown items of total cycles. They represent the executed instuction 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 D1 and shared D2 cache miss penalty ('**D$1**', '**D$2**'), and the register window miss penalty ('**window**') respectively.

As we can see from the result of (P) ($n = 1, k = 0$), (P_2) ($n = 2, k = 1$) and (P_4) ($n = 4, k = 2$) in Fig. 6, the appropriate unwinding degrees should vary from program to program. While (P_4) is the best model for 103.su2cor and 141.apsi, (P_2) is the best for 110.applu, and (P) is the best for 107.mgrid. Now, notice that the performance of 107.mgrid is extremely poor with (P_4) model. This shows that recklessly increasing the unwinding degree may lead to serious performance deterioration.

Meanwhile, (D) the new dynamic unwinding model is rewarded with good results. For 102.swim, 107.mgrid and 145.fppp, (D) shows as good result as the best model among (P), (P_2) and (P_4). For 101.tomcatv, 104.hydro2d, 125.turb3d and 146.wave5, (D) shows the best result among the all five models. This shows that it is very important to define the unwinding degree for each program and loop, and to change the degree dynamically.

Looking at the breakdowns, it is found that the dynamic unwinding model (D) can reduce 'exec' cycles drastically for 101.tomcatv, 102.swim, 103.su2cor and 146.wave5 in comparison with the traditional model (P). This should be the fruit of the increase of the reuse hit rate by efficient use of MemoTbl. Note that (D) also reduces reuse test overheads 'test(r)' and 'test(m)' on some programs without increasing 'exec' cycles. This shows that the reuse overheads can be reduced effectively by merging continuous multiple iterations into one reusable computation region.

Every after changing the unwinding degree dynamically, the dynamic unwinding model (D) should execute or reuse the associated computation region at least T times. If the new degree is inappropriate, the execution hinders its performance from progress. Two benchmark programs 107.mgrid and 110.applu are influenced by this problem, and their performances on the model (D) are slightly lower than on the traditional model (P). However, the perfoamence of the model (D) is better than the traditional model (P) as a whole. The model (D) improves the maximum speedup from 40.5% to 57.6%, and the average from 15.0% to 26.0%.

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

TABLE II
Reduced Execution Cycles (SPEC CPU95 FP).

		Mean	Max
(P)	traditional model	**15.0%**	**40.5%** (107.mgrid)
(P_2)	ref: fixed degree 2	19.3%	41.5% (101.tomcatv)
(P_4)	ref: fixed degree 4	15.9%	42.7% (102.swim)
(D)	dynamic unwinding model	**26.0%**	**57.6%** (101.tomcatv)

VII. Conclusions

In this paper, we have proposed a new parallel speculative execution model which can manage continuous multiple iterations as one reusable computation region without hinting or recompilation at all. An mechanism for deciding the appropriate unwinding degree for each loop and changing the degrees

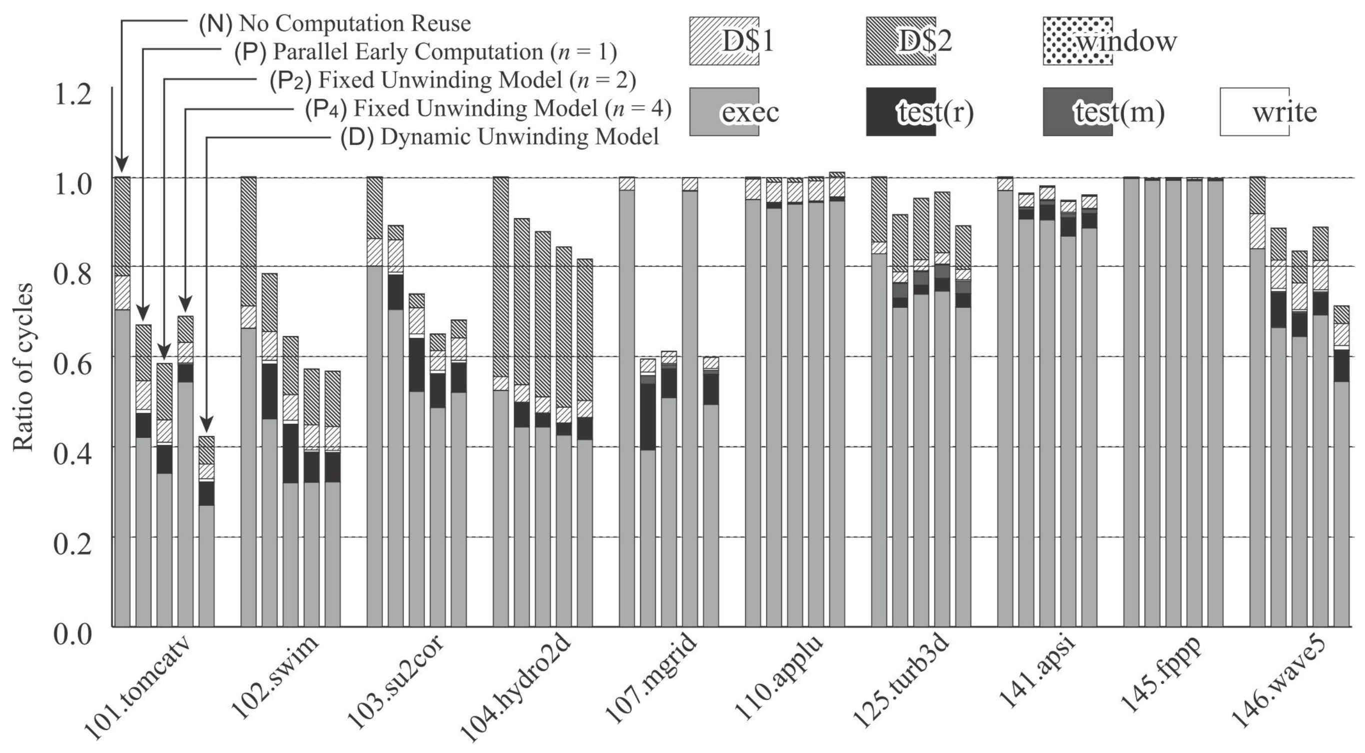

Fig. 6. Ratio of cycles (SPEC CPU95 FP).

dynamically has also been proposed.

Through an evaluation with SPEC CPU95 FP suite benchmark programs, it is found that the proposal model brings the reduction of computation reuse overhead and the high hit rate of computation reuse. The new model improves the maximum speedup ratio from 40.5% to 57.6%, and the average speedup ratio from 15.0% to 26.0%.

One of the our future works are to investigate the effect of the new dynamic unwinding model, such as the coverage of this model, the reuse hit rate of other functions/loops, and so on. Merging this model with other low-overhead models such as [5] we had proposed, or reducing the shared D2 cache misses by an improvement of prefetching effect of SpCs are also left for future works.

In this paper, we have implemented this dynamic unwinding model on a simple single-issue processor architecture. Implementing this model on more recent architecture such as superscaler, and trying to merge ILP-based methods and memoization are also our future works.

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] Tilera Corporation, *TILE-Gx Processor Family Product Brief*, 2009.
[5] 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.
[6] 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.
[7] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *29th MICRO*, Dec. 1996, pp. 226–237.
[8] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *30th MICRO*, Dec. 1997, pp. 281–290.
[9] A. Roth and G. S. Sohi, "Register integration: A simple and efficient implementation of squash reuse," in *33rd MICRO*, Dec. 2000.
[10] 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.
[11] C. Molina, A. González, and J. Tubella, "Trace-level speculative multithreaded architecture," in *ICCD*, 2002.
[12] P. Norvig, *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1992.
[13] 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.
[14] 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.
[15] 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.
[16] *SPARC64-III User's Guide*, HAL Computer Systems/Fujitsu, May 1998.
[17] MOSAID Technologies Inc., *Feature Sheet: MOSAID Class-IC DC18288*, 1st ed., Feb. 2003.