Proceedings of the 25th IASTED International Multi-Conference
Parallel and Distributed Computing and Networks
February 13–15, 2007, Innsbruck, Austria
ISBN Hardcopy: 978-0-88986-637-9 / CD: 978-0-88986-639-3

DESIGN AND EVALUATION OF AN AUTO-MEMOIZATION PROCESSOR

Tomoaki TSUMURA
Nagoya Inst. of Tech.
Gokiso, Showa, Nagoya, Japan
tsumura@nitech.ac.jp

Ikuma SUZUKI
Toyohashi Univ. of Tech.
1-1 Hibarigaoka, Tempaku,
Toyohashi, Aichi, Japan
ikuma@para.tutics.tut.ac.jp

Yasuki IKEUCHI
Toyohashi Univ. of Tech.
1-1 Hibarigaoka, Tempaku,
Toyohashi, Aichi, Japan
yasuki@para.tutics.tut.ac.jp

Hiroshi MATSUO
Nagoya Inst. of Tech.
Gokiso, Showa, Nagoya, Japan
matsuo@nitech.ac.jp

Hiroshi NAKASHIMA
Academic Center for
Computing and Media Studies
Kyoto Univ.
Yoshida, Sakyo, Kyoto, Japan
h.nakashima@media.kyoto-u.ac.jp

Yasuhiko NAKASHIMA
Grad. School of Info. Sci.
Nara Inst. of Sci. and Tech.
8916-5 Takayama
Ikoma, Nara, Japan
nakashim@is.naist.jp

ABSTRACT

This paper describes the design and evaluation of an auto-memoization processor. The major point of this proposal is to detect the multilevel functions and loops with no additional instructions controlled by the compiler. This general purpose processor detects the functions and loops, and memoizes them automatically and dynamically. Hence, any load modules and binary programs can gain speedup without recompilation or rewriting.

We also propose a parallel execution by multiple speculative cores and one main memoing core. While main core executes a memoizable region, speculative cores execute the same region simultaneously. The speculative execution uses predicted inputs. This can omit the execution of instruction regions whose inputs show monotonous increase or decrease, and may effectively use surplus cores in coming many-core era.

The result of the experiment with GENEsYs: genetic algorithm programs shows that our auto-memoization processor gains significantly large speedup, up to 7.1-fold and 1.6-fold on average. Another result with SPEC CPU95 suite benchmarks shows that the auto-memoization with three speculative cores achieves up to 2.9-fold speedup for 102.swim and 1.4-fold on average. It also shows that the parallel execution by speculative cores reduces cache misses just like pre-fetching.

KEY WORDS
memoization, computational-reuse, speculative multi-threading, CAM

1 Introduction

As semiconductors goes smaller, microprocessors are coming 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.

Meanwhile, in the software field, *memoization*[1] 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 being rewritten of target programs. The traditional load-modules or binaries cannot enjoy memoization. Furthermore, the effectiveness of memoization is influenced much by programmers. Rewriting programs using memoization occasionally makes the programs slower. Memoization costs a certain measure of 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. On-chip

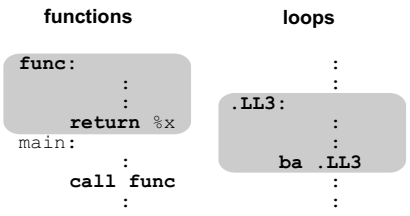

Figure 1. Memoizable Instruction Regions.

ternary CAM (Contents Addressable Memory) stores the inputs and outputs.

The rest of this paper is organized as follows. The next section describes the design and behavior of auto-memoization processor, and Section 3 presents parallel execution model with speculative cores. Section 4 shows the result of experiments to evaluate the processor. After a brief discussion of related work in Section 5, we conclude this paper in Section 6 showing our future work.

2 Auto-Memoization Processor

2.1 Outline

Our auto-memoization processor memoize all functions and loops. Figure 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 back branch instruction and its branch target is detected as a memoizable loop iteration. This processor detects these memoizable regions automatically and memoizes them.

Figure 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 (*reuse test*) 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 back*s 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 *store*s the contents of MemoBuf into MemoTbl for future reuse.

2.2 Inputs / Outputs of Memoizable Region

For memoing instruction regions, processor need to recognize what are the inputs and outputs of the regions dynamically. The inputs of a function are not only its arguments but also the variables which are referred to in the function.

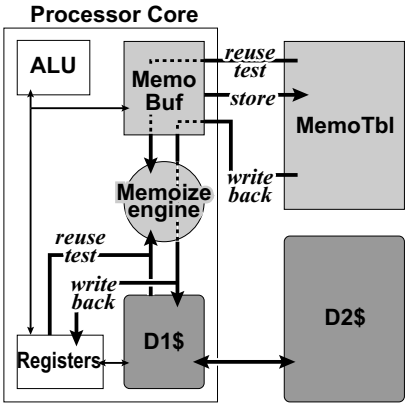

Figure 2. Structure of Auto-Memoization Processor.

Strictly speaking, the variable which is read and has not been written in the function is one of the inputs. If there is a write access to a variable prior to read, the variable is not an input. Referring a variable can be observed as a read access to a register or main memory. Hence, the processor can hook the read accesses and detect variable references.

Now, there is an exception. There is no need to treat function-*local* variables as inputs, even if they are referred in the function. That is to say, the processor need to distinguish the accesses to local variables from the accesses to non-local variables. Generally, the operating system specifies the upper limit of the data/stack size statically. Our processor tells global variables from local variables by this boundary, and tells parent-local variables from local variables by the value of the stack pointer just before current function is called. On the other hand, there is no way to know local variables from non-local variables for loops. Therefore in loops, all referred variables are treated as inputs.

Outputs can be also detected in a similar way. The variables assigned in the instruction region are the outputs of the region, but the local variables are excluded.

2.3 Memo Tables

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. Figure 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.

```
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;
}
```

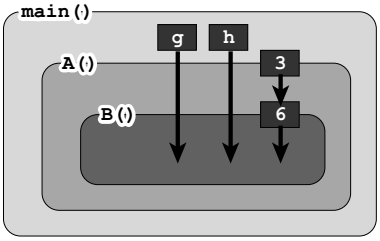

Figure 3. Nested functions.: The global variables *g*, *h* should be treated as inputs not only for *B* but also *A*.

Input tree: A series of inputs for a certain instruction region can be represented as a sequence of the tuples each of which contains an address and a value. In one instruction region, the series of input addresses sometimes branch off from each other. For example, after a branch instruction, what address will be referred next relies on whether the branch was taken or not-taken. Let's see the program shown in Figure 3. The branch taken/not-taken of the if-else statement in *B* depends on an input value *x*, and the value of *x* effects the next input variable (or address). In other words, the value of an input variable can alter the subsequent input addresses.

Therefore, the universal set of the different series of inputs for an instruction region can be represented as a multiway *input tree*. Each node of the tree represents register number or memory address which is referred in instruction regions, and each edge represents the stored value in it. Treating the start address of a memoizable region as one of the inputs of the region, the whole input sequences of all memoizable regions are represented by one big input tree. A series of inputs of a memoizable region is represented as a way from the root to a leaf on this tree.

MemoTbl: Now, MemoTbl should keep this input tree in it. Figure 4 shows the input tree and the brief structure of MemoTbl. 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.[1] An edge of input tree is corresponds to a CAM line. Hence the address of CAM line works as the index for the edge. Each CAM line has a value field and an index field which points to its parental line. There is also a time stamp field (not shown in Figure 4). It remembers the last access time of each lines. Running out of the MemoTbl, the memoization engine sweeps old entries all together by 'Search and Delete' instruction of the CAM.

Figure 4 shows how input matching goes. First, the memoization engine reads the value of program counter

[1] For simplification, it is represented as 32 bits in Figure 4.

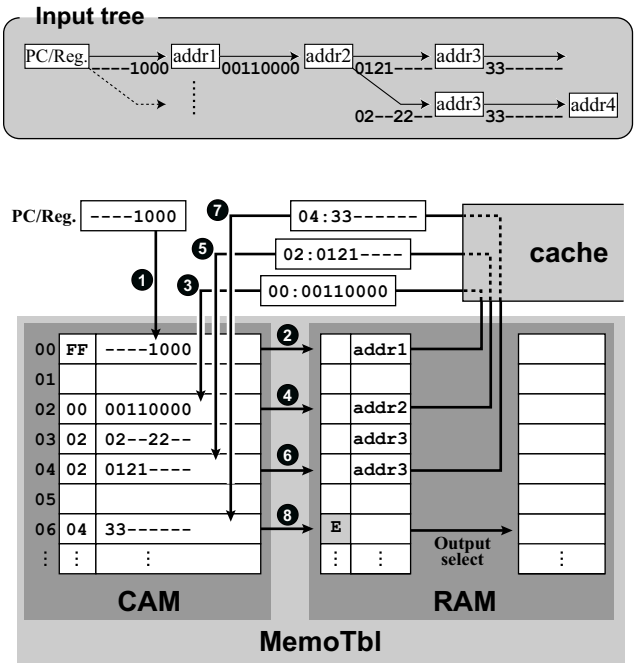

Figure 4. Structure of MemoTbl.

(PC) and registers. Suppose the value was '`----1000`.' The CAM entry stores whole cache line which includes the input. Hence, the non-input bits are masked. They are implemented by *don't care* bits on ternary CAM. The memoization engine searches the concatenation of '`FF`' and '`----1000`' through the MemoTbl CAM. '`FF`' represents the tree root. The entry '`00`' is picked up and '`addr1`' is adopted as next input address. Memoization engine reads the memory address '`addr1`' and get the value '`00110000`.' Next search key is the concatenation of the index '`00`' and '`00110000`'. In this way, the subsets of inputs are searched repeatedly. When the flag '`E`' which represents the tail of inputs is detected, current inputs are proved to be equivalent to a series of past inputs, and then the execution of the instruction region can be omitted.

3 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. But how can we use these *many-core* processors effectively is still under review between researchers.

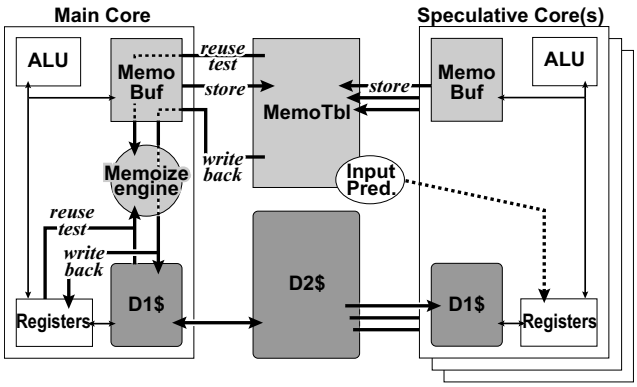

Figure 5. Auto-Memoization Processor with Speculative Cores.

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 append some speculative cores to our auto-memoization processor. These cores help the memoization-unfriendly regions mentioned above. Figure 5 shows the new structure of the auto-memoization processor with speculative 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, speculative 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. With three speculative cores, the instruction region is executed using three predicted input sets concurrently with the main core. Speculative 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, 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.

4 Performance Evaluation

We have developed a single-issue simple SPARC-V8 simulator with auto-memoization structures. This section discusses the performance of the processor.

4.1 Processor Configuration

The simulation parameters are shown in Table 1. The cache structure and the instruction latency are based on SPARC64[2]. As for memo tables, we defined the size of MemoBuf as 32KB (32Bytes $\times$ 256 lines $\times$ 4set), and the size of MemoTbl CAM as 2MB (32Bytes $\times$ 64K lines). The size of MemoBuf is equal to D1 cache, and the size of MemoTbl is equal to D2 cache.

Table 1. Processor Parameters

D1 Cache	32 K Bytes
Line size	32 Bytes
Ways	4
Latency	2 cycles
Cache Miss Penalty	10 cycles
D2 Cache	2 M Bytes
Line size	32 Bytes
Ways	4
Latency	10 cycles
Cache Miss Penalty	100 cycles
Register Windows	4 sets
Window Miss Penalty	20 cycles/set
Load Latency	2 cycles
Integer Mult.	8 cycles
Integer Div.	70 cycles
Floating Add/Mult.	4 cycles
Single Div.	16 cycles
Double Div.	19 cycles
MemoBuf size	32 K Bytes
MemoTbl size	2 M Bytes

Table 2. GENEsYs Parameters

Mutation Probability	0.1 %
Crossover Probability	60.0 %
# of Crossover Points	2
Population Size	50
# of Trials per Experiment	1000
Other paramseters	defaults

The on-chip CAM is modeled on MOSAID DC18288[3]. We assume the latency for input matching between CAM and the register as 32Byte/cycle, and the latency between CAM and main memory as 32Byte/2cycle.

4.2 Genetic Algorithms

First, we evaluated our processor with no speculative cores mentioned in Section 3. We used GENEsYs programs which are compiled by gcc-3.0.1 with `-msupersparc -O2` options. GENEsYs is an implementation of genetic algorithms based on Grefenstette's GENESIS[4]. GENEsYs provides 24 fitness functions, including De Jong's functions, TSP-problem, fractal function, and so on. The typical GA parameters are shown in Table 2.

Figure 6 shows the result in the form of normalized execution cycles. Each bar is normalized to the number of executed cycles without memoization. Each workloads are represented by two bars in this chart. The left bar plots the clock cycles that the original GENEsYs workload costs, and the right bar plots the clock cycles with auto-memoization.

The legend shows the cycles for eight components

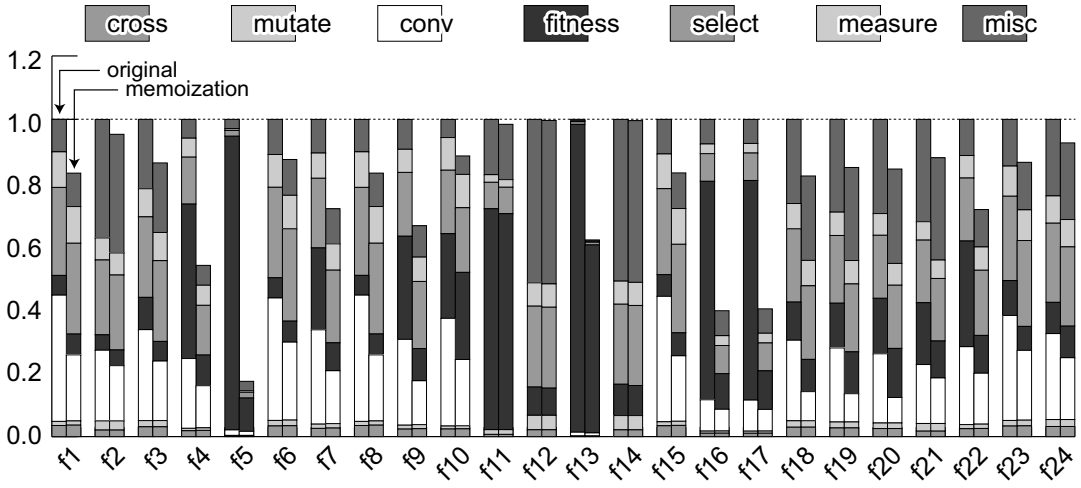

Figure 6. Normalized execution time (GENEsYs)

of the GENEsYs workloads. They represent the cycles for crossover operation (‘cross’), mutate operation (‘mutate’), genotype format conversion (‘conv’), fitness calculation (‘fitness’), genotype selection (‘select’), convergence measurement (‘measure’), and other miscellaneous processes such as variable initializations (‘misc’) respectively.

As the bars clearly indicate, auto-memoization reduces up to 83% execution cycles. The average of reduced cycles is 28%. Note that the workloads with high-load fitness functions, such as f4 (Quartic function with noise), f5 (Shekel’s foxholes), f13 (Weierstrass-Mandelbrot fractal function), f16 and f17 (Fletcher and Powell), gain large speedup. In other words, the fitness functions with higher load tend to gain larger speedup by auto-memoization. Auto-memoization gains over 2-fold speedup with these high-load functions on average.

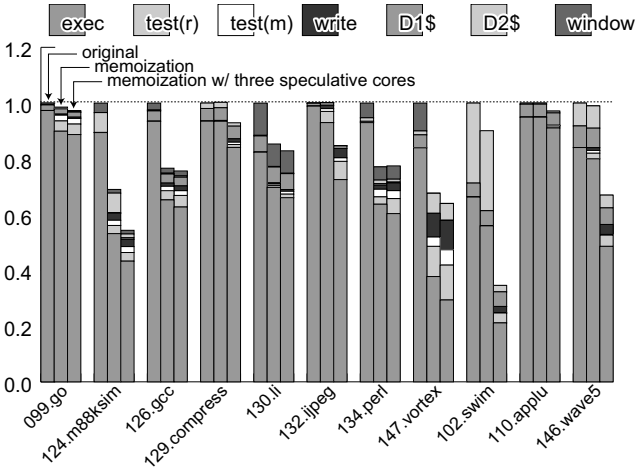

Figure 7. Normalized execution time (SPEC CPU95)

4.3 SPEC CPU95

Next, we evaluated our auto-memoization processor with three speculative cores. Workloads are eleven benchmarks in SPEC CPU95 suits and are executed with ‘train’ dataset. All benchmarks are compiled by gcc-3.0.1 with `-msupersparc -O2` options.

Figure 7 shows the result in the form of normalized execution cycles. Each bar is normalized to the number of executed cycles without memoization. Each benchmarks are represented by three bars in this chart. The left bar plots the baseline that the original benchmark costs. The center bar plots the clock cycles using auto-memoization with no speculative cores. The right bar plots the clock cycles using auto-memoization and three speculative cores.

The legend shows the itemized statements of cycles. They represent the executed instruction cycles (‘exec’), the overhead for the comparison between MemoTbl and the registers (‘test(r)’), the overhead for the comparison between MemoTbl and the cache (‘test(m)’), the overhead to write from MemoTbl to the registers/cache (‘write’), D1 cache miss penalty (‘D1$’), shared D2 cache miss penalty (‘D2$’), and register window miss penalty (‘window’) respectively.

The normal auto-memoization reduces up to 32% execution cycles, and 13% cycles on average. With three speculative cores, auto-memoization reduces up to 65% cycles, and 27% cycles on average. Note that not only ‘exec’ but also ‘D2$’ is considerably reduced with speculative cores for 124.m88ksim and 102.swim. This result shows that parallel execution by speculative cores works also just like pre-fetching for shared cache.

5 Related Work

A general-purpose reuse technique for single instructions was proposed by Sodani[5]. Fully associative reuse buffer

(RB) with maximum 1024 is assumed. Each entry holds the operand values and the result of an instruction. The mem-valid bit and the mem-address fields are maintained to ensure a load instruction can be reused. On a store the mem-address fields are associatively searched and the mem-valid bits are reset. González[6] evaluated reuse trace memory (RTM) with maximum 256K entries. Each entry is indexed by part of PC (e.g. 8-way set-associative and 16 entries per PC) and is assumed to hold 8-in 8-out register values and 4-in 4-out memory values. The maximum size of RTM is over 32MB. This work provides the upper bounds achieved with an infinite reuse trace memory. Costa[7] proposed a reuse scheme which employs a fully associative table that does not include load/store instruction. The program counter and the operand values are associatively compared. This work approaches trace-level reuse by providing an evaluation of a feasible reuse mechanism against [6].

Wu[8] proposed a SpMT using computation reuse and value prediction. The compiler profiles the target programs in advance, and insert special *reuse* instructions at the beginning of memoizable regions. This scheme also predicts the outputs of memoizable regions and execute the code after the regions speculatively. When the output prediction proves to be incorrect, speculative execution needs backout. It achieves speedup in the range from 1.33- to 1.4-fold with SPEC CPU95 benchmarks. On the other hand, our auto-memoization processor achieves up to 2.9-fold and 1.4-fold speedup on average without special instructions, profiling and recompilation. Molina[9] proposed a trace-level speculative multithreaded architecture. The instructions executed by speculative threads are stored in FIFO. The main thread fetches instruction from the FIFO, compares source operands, and reuse the results when source operands match. It achieves speedup up to 1.33-fold and 1.16-fold on average.

6 Conclusion

In this paper, we proposed a auto-memoization processor. This processor dynamically detects the functions and loops in the binary programs as memoizable instruction region, and memoize them automatically. Memo table for storing past inputs and outputs is implemented by general ternary CAM. The superior point of the processor is that only the comparison between current input values and recorded input values is required and the verification of the results is unnecessary. Moreover, the number of instructions in the memoizable regions does not affect the complexity of the memoization mechanism.

We also proposed additional speculative cores for the auto-memoization processor. Parallel execution by these multiple speculative cores can omit the execution of instruction regions even if whose inputs show monotonous increase/decrease.

The performance evaluation results with GENEsYs genetic algorithm programs prove the efficiency of auto-memoization showing a large degree of speedup up to 7.1-, and 1.6-fold on average. The experiment result with SPEC CPU95 benchmarks shows that speculative cores push up the maximum eliminated cycles from 32% to 65%. Furthermore, parallel execution by speculative cores reduces miss rates of shared data cache well.

Our future work is to evaluate the auto-memoization on pipelined superscalar processor models. Other important issues including further detailed investigation, hardware cost estimation, energy-efficiency, and examination of management algorithms for memo tables are also left for future works.

Acknowledgment

This research was partially supported by a Scientific Research (B), #17300015, 2005, and a Grant-in-Aid for Exploratory Research, #18650005, 2006, and a Grant-in-Aid for Exploratory Research, #18650020, 2006, from the Ministry of Education, Science, Sports and Culture of Japan and the Okawa foundation for information and telecommunications.

References

[1] Peter Norvig. *Paradigms of Artificial Intelligence Programming*. Morgan Kaufmann, 1992.

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

[3] MOSAID Technologies Inc. *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition, Feb. 2003.

[4] John J. Grefenstette. GENESIS: A system for using genetic search procedures. In *Proc. of the 1984 Conference on Intelligent Systems and Machines*, pages 161–165, 1984.

[5] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proc. 24th International Symposium on Computer Architecture*, pages 194–205, Jun. 1997.

[6] Antonio González, Jordi Tubella, and Carlos Molina. Trace-level reuse. In *Proc. International Conference on Parallel Processing*, pages 30–37, Sep. 1999.

[7] Amarildo T. da Costa, Felipe M. G. França, and Eliseu M. C. Filho. The dynamic trace memorization reuse technique. In *PACT*, pages 92–99, 2000.

[8] Youfeng Wu, Dong-Yuan Chen, and Jesse Fang. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *28th ISCA*, pages 98–108, 2001.

[9] Carlos Molina, Antonio González, and Jordi Tubella. Trace-level speculative multithreaded architecture. In *ICCD*, 2002.