Regular Paper

Limits of Thread-Level Parallelism in Non-numerical Programs

Akio Nakajima,[†,☆] Ryotaro Kobayashi,[†] Hideki Ando[††] and Toshio Shimada[†]

Chip multiprocessors (CMPs), which recently became available with the advance of LSI technology, can outperform current superscalar processors by exploiting thread-level parallelism (TLP). However, the effectiveness of CMPs unfortunately depends greatly on their applications. In particular, they have so far not brought any significant benefit to non-numerical programs. This study explores what techniques are required to extract large amounts of TLP in non-numerical programs. We focus particularly on three techniques: thread partitioning with various control structure levels, speculative thread execution, and speculative register communication. We evaluate these techniques by examining the upper bound of the TLP, using trace-driven simulations. Our results are as follows. First, little TLP can be extracted without both of the speculations in any of the partitioning levels. Second, with the speculations, available TLP is still limited in conventional function-level and loop-level partitioning. However, it increases considerably with basic block-level partitioning. Finally, in basic block-level partitioning, focusing on control-equivalence instead of post-domination can significantly reduce the compile time, with a modest degradation of TLP.

1. Introduction

Chip multiprocessors (CMPs) are expected to be the major next-generation microprocessors. A CMP exploits thread-level parallelism (TLP) in addition to instruction-level parallelism (ILP) by executing multiple threads in parallel. Because of the diminishing returns from investment of resources in the exploitation of ILP, TLP is expected to be a major source of performance improvements.

Recently, commercial CMPs have been announced with expectations of such improvements. However, for now, effective applications of CMPs are limited. They are principally intended for explicitly parallel applications (e.g., independent multiple programs, on-line transaction processing (OLTP), and numerical programs). Although the CMP architecture improves the performance of such applications [4], it brings little benefit to implicitly parallel applications, i.e., non-numerical programs, which form the largest group of computer applications. Many academic studies have been carried out [2),8),10),12),13),15),16)], but unfortunately the speedup so far confirmed relative to a single superscalar processor is insufficient.

In general, two approaches are available to find solutions to this problem. One is a bottom-up approach that consists in proposing a technique and evaluating the amount of TLP in a particular architecture. The other is a top-down approach that consists in examining the amount of TLP contained in a program by imposing only constraints associated with a particular technique and relaxing all other constraints. A study of the latter approach is called a *limit study*, because the TLP obtained is the upper bound. A limit study suggests that the technique is potentially beneficial if the upper bound is large enough; otherwise, it is definitely inadequate.

A pioneering investigation of TLP limit studies for non-numerical programs was carried out by Lam and Wilson [9]. Their study focuses on control flows when partitioning a program into threads, and shows that the limit of parallelism becomes much higher than that of single-thread execution if a program is partitioned into control-independent threads at the basic block level. The results of this study are quite encouraging as regards the possibility of building a CMP for non-numerical programs, but there is still scope for further investigation.

We are interested in two aspects of techniques that can significantly affect the limit of TLP. One is the level of partitioning. In general, a compiler partitions a program into threads based on control structures, and the level of a control structure for thread partitioning is the source of TLP. The levels that usually exist

† Department of Electrical Engineering and Computer Science, Nagoya University
†† Department of Computational Science and Engineering, Nagoya University
☆ Presently with Hitachi Ltd.

are functions, loops, and basic blocks. Function and loop levels are often used in traditional numerical parallelizing compilers (e.g., Stanford SUIF [3]), because parallelism at those levels is generally abundant in numerical programs. We question whether those levels also have a sufficient amount of TLP in non-numerical programs, or whether the compiler must further search for TLP at a lower level (i.e., the basic block level). Basic block level partitioning is costly in compile time, and it imposes more overhead related to parallel thread execution (e.g., inter-thread communication and thread creation) because the size of a thread is small relative to that in the upper levels. Furthermore, it is technically undeveloped, though some studies have been carried out [6),16)]. Thus, loop or function levels may be more desirable if their upper bound is large enough.

Another aspect we are interested in is techniques for relaxing control dependence constraints. Intuitively, control dependences can severely limit the amount of available parallelism unless special care is taken, because conditional branches frequently appear in non-numerical programs. However, their negative effect can be minimized if the compiler successfully partitions a program into control-independent threads with sufficient parallelism. This study introduces speculative thread execution and speculative inter-thread register communication as techniques for relaxing control dependence, and explores their effect on the TLP limit.

The remainder of this paper is organized as follows. Section 2 explains the basic assumptions of a multiprocessor and the processor cores in our simulation. Section 3 models thread execution associated with thread partitioning levels. Section 4 describes the speculative thread execution and register communication. Section 5 presents evaluation results. Section 6 compares our study with previous work. Finally, Section 7 concludes the paper.

2. Basic Assumptions

This section describes our basic assumptions about processor cores and multiprocessors. They are common to any model examined here in our simulation.

2.1 Processor Core

Each processor core in the multiprocessor fetches all instructions on the branch-predicted path simultaneously with infinite bandwidth until a mispredicted branch is encountered. No instruction can be fetched until the immediately preceding mispredicted branch has been executed. We assume a local-history two-level branch predictor, PAs [17), with sufficiently large tables. No branch misprediction penalty is imposed. Fetched instructions are stored in an instruction window of infinite size. Those instructions whose data dependences are resolved are then issued and executed simultaneously. The latency of any instruction is a single cycle. The number of registers is assumed to be infinite, and thus all of the anti- and output dependences are resolved by register renaming. In addition, memory disambiguation is ideally removed. The issue width, function units, and memory ports are assumed to be infinite, and consequently no resource constraints exist. We call this model the *SP model*.

We further define the ORACLE model to find the upper bound of parallelism in our experiment. It has perfect branch prediction, and the execution order of its instructions is constrained only by true data dependences.

2.2 Multiprocessor

An infinite number of threads can be executed simultaneously with an infinite number of processor cores. Memory is shared, with an infinite number of ports having a single-cycle access latency. A thread is created at a fork point with zero-cycle latency. Thread creation is performed non-speculatively on control dependences unless explicitly described; that is, a thread is created when the control is guaranteed to reach its fork point.

Register values are communicated directly, not via memory, among threads. Note that this is not a radical assumption in a CMP [7),8),15)]. Register communication has a zero-cycle latency with infinite bandwidth, and it is performed non-speculatively on control dependences unless explicitly described; that is, a defined value in a thread is sent to the consumer threads when the definition is found to *reach* [1)] the consumer after resolution of control dependences (a detailed discussion will be provided in Section 4.2). On the other hand, in memory value communication, the last definition of a memory variable is assumed to be known *a priori*. More generally, memory disambiguation among threads is assumed to be perfect.

3. Models of Thread Partitioning Levels

This section first describes the definitions of models associated with thread partitioning levels, and then gives a small example that explains the models.

3.1 Model Definitions

We define the following four models:

- The *FC model* partitions a program at the function level. It attempts to create a new thread that contains a callee function at each function call. The current thread continues beyond the function call.
- The *LP model* partitions a program at the loop level. It attempts to create N new threads at the header of the loop, where N is the number of loop iterations. Each thread contains each single-loop iteration other than the first iteration, and the portion succeeding to the loop; the first iteration is contained in the current thread. The model knows N *a priori*, even if it is determined only dynamically. This implies perfect speculation regarding the control of loop iteration, unlike other models. We introduce this perfection to ascertain the upper bound of the loop-level partitioning.
- The *PD model* partitions a program at the basic block level. At the top of each basic block, the model attempts to create new threads, where each starts from its *post-dominating*[18] block. Note that block Y is said to post-dominate X if Y appears on any path from X to the EXIT on the control flow graph. From the definition, the created threads are control-independent of the current thread.
- The *CE model* also partitions a program into control-independent threads at the basic block level, like the PD model, but limits the starting point of a new thread to a *control-equivalent*[14] block, not a post-dominating block. Note that block X is control-equivalent with block Y if X dominates[1] Y and Y post-dominates X. Although there are various situations in partitioning of the CE model, typically it partitions a program at the IF-THEN-ELSE level. While the PD model can create a new thread from any part of the IF, THEN, and ELSE, the CE model can only create a new thread from the IF part. Therefore, the number of candidates of partitioning is reduced. This makes it possible to reduce the calculation cost incurred by the compiler for selecting beneficial partitioning.

In any model, instructions within a thread are executed as in the SP model.

In our simulation, we optimally partition a program by referring to a trace. We first calculate the partitioning candidates associated with a model. Then, in the order that appears on the trace, we simulate instruction execution for each candidate for cases of both partitioned and non-partitioned programs for sufficient long cycles. If the partitioning is found to be beneficial, the candidate is adopted; otherwise, it is discarded.

To examine the upper bound, we perform perfect function inlining; that is, we ignore dependences caused by the conventions of register-use and function call. For example, we ignore all operations of stack frame allocation and deallocation, and those of saving and restoring preserving registers. Also, we ignore loop induction variables, because they severely limit parallelism as loop-carried dependences, but can be easily removed by the compiler.

3.2 Example

We explain the ORACLE, SP, FC, LP, and PD models, using a small-scale example. The CE model is omitted because it can be easily understood from the PD model.

Consider a program presented by the control flow graph shown in **Fig. 1**a. The program consists of two functions: `func1` and its callee `func2`. Assume for the sake of simplicity that each node representing a basic block has a single instruction that is numbered, and that there are no data dependences among the instructions. The edges represent control flows. Branch predicted flows are highlighted by bold edges.

Consider the trace of this program shown in Fig. 1b. A letter `a` or `b` is attached to each instruction number to distinguish the specific instance if necessary. Mispredicted branch instructions are circled.

The execution order in each model is shown in **Fig. 2**. The instructions in a rectangle with a broken outline compose a single thread. The instructions at the same level are executed simultaneously. Here, for the sake of simplicity, we do not consider the benefit of partitioning, and instead show all of the partitioning candidates. As explained before, we discard candidates that are not beneficial in our simulation.

The ORACLE model executes all of the in-

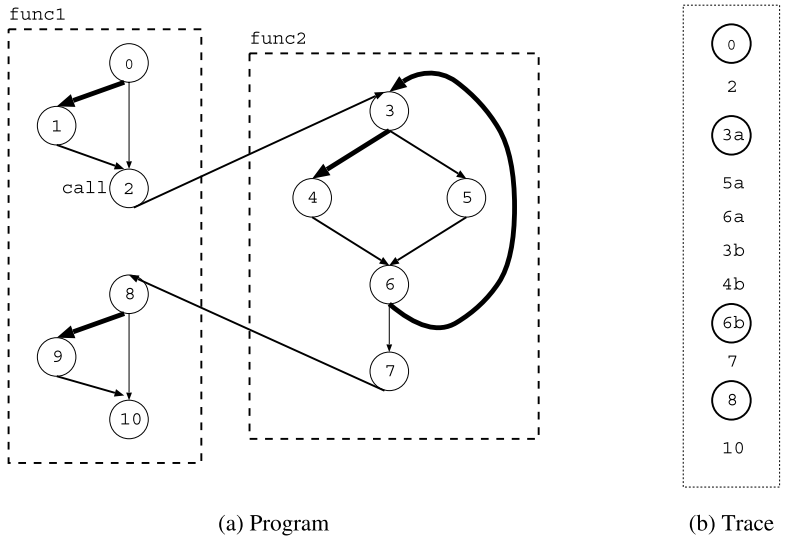

Fig. 1 Example program and trace.

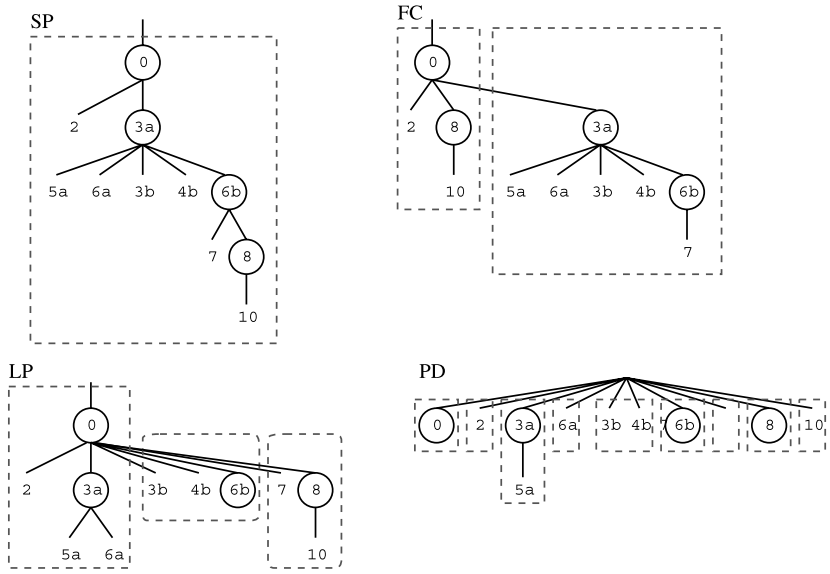

Fig. 2 Execution order.

structions simultaneously, because there are no data dependences (we omit the figure). The SP model simultaneously fetches instructions between mispredicted branches, and executes them in the order that satisfies data dependences. In this example, because there are no data dependences, all of the fetched instructions are executed simultaneously. The FC model creates a new thread with `func2` when call instruction `2` is fetched. The LP model creates two new threads when instruction `3a` in the loop head is fetched: the second iteration of the loop and the instructions succeeding to the loop. Note that the first iteration is included in the current thread. The PD model creates new threads starting from `2`, `3a`, `6a`, `3b`, `6b`, `7`, `8`, and `10` when instruction `0` is fetched, because the blocks to which these instructions belong post-dominate the block to which instruction `0` belongs.

4. Relaxation of Control Dependence

This section describes speculative thread execution and register communication, which relax control dependences.

4.1 Speculative Thread Execution

We can easily infer that control dependences significantly limit TLP in non-numerical programs, because branches frequently appear. Thus, speculative thread execution on branches should have a great benefit. Control speculation generally relies on branch prediction to maximize the benefit. Although potentially there are many ways to realize speculative thread execution, we believe that there are three approaches: One relies on static branch prediction. The compiler predicts a likely path, and adequately partitions the path. Another approach relies on dynamic branch prediction. The compiler partitions a program into threads in such a way that a newly created thread has no control dependences on its fork point. However, at run time, the new threads are created and start execution soon after the fork point is speculatively fetched. The third approach combines both of the approaches described above.

It is difficult to determine which approach is the best, because the first approach has a disadvantage in that it relies on low-accuracy static branch prediction, the second approach limits the opportunity for partitioning to static control independence, and the third approach is complicated. While this is an important issue, it is beyond the scope of this study. In our simulation, we adopt the second approach. That is, if the fetched predicted path between consecutive mispredicted branches includes fork points, each immediately and simultaneously creates new threads.

4.2 Speculative Register Communication

As explained in Section 2.2, register communication must wait until the definition is determined to reach the consumer. We explain this constraint using the following code example:

```
i0: r1 = 1;
i1: if (r2)
i2:   r1 = 2;
i3: r3 = r1;
```

Consider that we partition this program into two threads: (`i0`, `i1`, `i2`) and (`i3`). Instruction `i3` uses register `r1`, but which of the register values defined by `i0` and `i2` will reach instruction `i3` is unknown before branch `i1` is executed. Consequently, the communication of register `r1` will be delayed until the branch execution, constraining TLP.

We can relax this constraint by introducing speculative register communication that relies on branch prediction. That is, a register value can be sent if the definition of that register is found to be the last on the predicted path. In the example above, assume that branch `i1` is predicted to be untaken. In this case, the value of `r1` is sent immediately after `i0` is executed, without waiting for the execution of branch `i1`.

In our simulation, if the fetched instructions on the predicted path between consecutive mispredicted branches include a last register definition instruction in the current thread, such a register is immediately communicated to the following threads.

5. Evaluation Results

We built a trace-driven simulator for evaluation. The trace is collected using the simulator from SimpleScalar Tool Set Version 3.0a [5]. The instruction set is the SimpleScalar/PISA, which is an extension of the MIPS R10000 instruction set [11]. We use eight benchmark programs of SPECint95.

5.1 Thread Partitioning Levels and Control-Speculations

This section evaluates the effect of thread partitioning levels on TLP, with and without speculative thread execution and register communication.

Figures 3, **4**, **5**, **6** show the values of TLP in the LP, FC, PD, and CE models, respectively. We define the speedup or the TLP for each model as the IPC of the corresponding model divided by that of the SP model. The four bars for each benchmark program represent the TLP with four different speculation techniques: no speculation, speculative thread execution only, speculative register communication only, and both the speculations. As a reference, **Table 1** shows the IPCs for the SP and ORACLE models as well as the speedup of the ORACLE model over the SP model.

As shown in Fig. 3, the FC model exhibits little TLP without the control speculations in any benchmark program. This is not surprising, because branches are very frequent in non-numerical programs, and consequently control dependence constraints severely limit TLP. Since functions are often data-independent, considering the programming abstraction man-

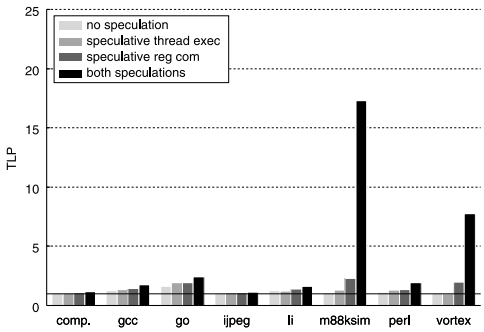

Fig. 3 TLP of the FC model.

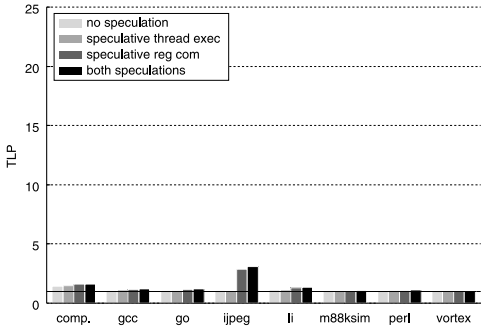

Fig. 4 TLP of the LP model.

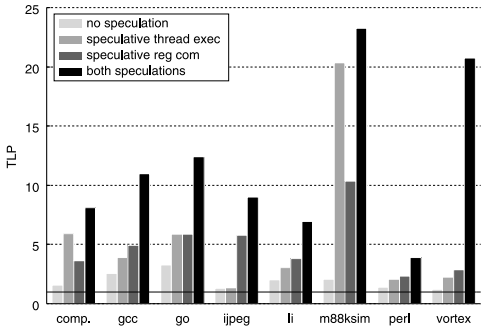

Fig. 5 TLP of the PD model.

ner, we expected high TLP if the speculations were introduced. Surprisingly, however, the TLP is still low in most benchmark programs. We believe the reason for this to be that the overlap executions of multiple functions are limited because there are many branches between function calls; branch misprediction sequentializes function execution even though data dependences among functions are fairly loose. The only exceptions are m88ksim and vortex. Those benchmark programs exhibit high TLP with both the speculations. This is because many dominant function calls fortunately depend on a small number of branches, and also these

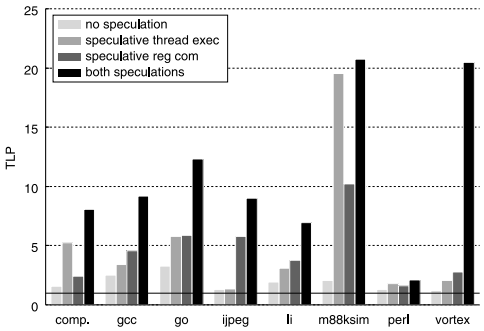

Fig. 6 TLP of the CE model.

Table 1 IPCs of the SP and ORACLE models.

Benchmark	IPC SP	IPC ORACLE	ORACLE Speedup
compress95	6.4	205.7	32.0
gcc	7.8	275.2	35.5
go	4.5	139.6	31.1
ijpeg	22.7	245.1	10.8
li	8.6	69.5	8.1
m88ksim	6.8	481.9	71.3
perl	10.0	173.2	17.3
vortex	38.3	815.8	21.3
G.M.	10.2	234.6	23.1

Table 2 Weighted average number of loop iterations.

Benchmark	Loop Iterations
compress95	8.2
gcc	2.4
go	2.9
ijpeg	16.8
li	2.4
m88ksim	2.8
perl	3.2
vortex	2.4

branches are highly predictable.

The TLP of the LP model is severely limited, as shown in Fig. 4. The first reason for this is that the number of loop iterations is generally very small. **Table 2** shows the average number of loop iterations, weighted by the number of times that the control entered each loop. The only programs for which we can expect high potential TLP are compress95 and ijpeg. These programs have a few dominant loops that iterate many times (approximately 50% in the dynamic instruction count), but they also have many loops with small iterations. The fraction of the execution time with such loops of small iterations limits the TLP by Amdahl's law. Moreover, as long as we look at the dominant loops in these programs, they have loop-carried dependences that

severely limit the TLP.

Since the PD model can create many control-independent threads beyond many branches without a rigid control structure constraint, unlike the FC and LP models, we expected that control dependence constraints would have a weak effect on TLP. Unexpectedly, however, control-dependence constraints strongly limit TLP, as in the FC and LP models (see Fig. 5). The TLP is only 1.7 on average, though it is larger than that of the FC and LP models. However, either speculative thread execution or speculative register communication increases the TLP. Combined, the speculations significantly further increase the TLP to 10.3 on average. The reason speculative thread execution is effective in the PD model is as follows. The PD model tends to create many new control-independent threads soon after the current thread is created. Considering that there is no control dependence among the current and newly created threads, the created threads depend on only a small number of branches. Since a control path with a small number of branches is highly predictable with the dynamic branch predictor, speculative thread execution effectively relaxes the control dependence constraint on threads. A similar argument can be made for the effectiveness of speculative register communication.

As can be seen by comparing Fig. 6 with Fig. 5, the CE model exhibits a similar amount of TLP to that of the PD model. The degradation from the PD model is only 11% on average. On the other hand, the amount of computation the compiler needs to do for partitioning is dramatically decreased. **Table 3** shows the number of static partitioning candidates per basic block in the PD and CE models, and the percentage reduction realized by use of the CE model. As shown in the table, the number of partitioning candidates is significantly reduced in most benchmark programs. Therefore, partitioning focusing on control-equivalence is attractive in terms of the extraction of large amounts of TLP with a short compilation time.

Table 3 Number of partitioning candidates per basic block and percentage reduction.

	#Cand./BB		%
Benchmark	PD	CE	Reduction
compress95	4.89	0.92	81.2%
gcc	8.07	3.01	62.8%
go	4.48	0.97	78.3%
ijpeg	4.18	1.06	74.8%
li	4.32	1.50	65.2%
m88ksim	30.17	26.90	10.8%
perl	5.44	1.04	80.8%
vortex	4.41	1.88	57.4%

6. Related Work

Olukotun et al. compared the performance of a CMP with a traditional architecture to a wide-issue single superscalar processor under the same die size [12]. They found that the parallelism achieved by the two processors is comparable in non-numerical programs, but they concluded that the CMP would outperform the single processor because of the advantage of the high clock rate. More recently, the Compaq Piranha project carried out similar performance comparisons on OLTP by designing a chip [4]. They extrapolated from their ASIC design to a custom design that a CMP with eight simple processor cores delivers five times the OLTP performance of a more complex single superscalar processor. Unlike those studies, we explore the upper bound of the TLP with various parallel execution techniques in abstract machines.

As mentioned in Section 1, Lam and Wilson did pioneering work in a TLP limit study [9]. They found that high TLP can be achieved if a program is partitioned into control-independent threads. Unlike their study, ours explores the effectiveness of various thread partitioning levels and control speculations.

7. Conclusions

In this paper, we have measured the limits of TLP in non-numerical programs to determine the effectiveness of three techniques: thread partitioning with various control structure levels, speculative thread execution, and speculative register communication. We found that, without the control speculations, little TLP is extracted from the loop and function levels, though a small amount is extracted from the basic block level. On the other hand, with the control speculations, although their effectiveness is limited at the loop and function levels, a large amount of TLP is obtained from the basic block level. In other words, basic block-level partitioning with the control speculations is essential to obtain high TLP.

Multithreading at the basic block level is immature, unlike that at the function and loop levels. In real systems, there are many hurdles to overcome in all areas, from compilers

to hardware. One of the issues is the large overhead related to parallel thread execution including thread creation and inter-thread communication. The size of a thread at the basic block level usually becomes small in comparison with that at the conventional levels, implying that the overhead is frequently imposed. With the advantage of integration into a single chip, CMP implementation can allow this overhead to be small in comparison with traditional on-board implementation, but further architectural research must be carried out to boost performance to a satisfactory level.

Another issue of concern is the misprediction penalty of control speculations. Because we focused on investigating the upper bound, we did not take misprediction penalties into account. However, in real systems, misprediction penalties considerably degrade performance under high TLP execution. To decrease the misprediction penalty, we must find branch predictors with higher accuracy, along with methods of recovering from misprediction with very small penalties.

From a software point of view, compile time is a concern. The compiler must optimally select beneficial partitions from a vast number of partition candidates, and the compile time increases according to the number of candidates. In this study, we have described a method that focuses on control-equivalence rather than post-domination. This method significantly reduces the size of the candidate set with a modest degradation of TLP.

References

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, MA (1986).
2) Akkary, H. and Driscoll, M.: A Dynamic Multithreading Processor, *Proc. 31st Int. Symp. on Microarchitecture*, pp.226–236 (1998).
3) Amarasinghe, S., Anderson, J., Lam, M. and Tseng, C.: The SUIF Compiler for Scalable Parallel Machines, *Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing* (1995).
4) Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R. and Verghese, B.: Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing, *Proc. 27th Int. Symp. Computer Architecture*, pp.39–51 (2000).
5) Burger, D. and Austin, T.M.: The Simplescalar Tool Set, Version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin-Madison (1997).
6) Kasahara, H., Honda, H., Mogi, A., Ogura, A., Fujiwara, K. and Narita, S.: A Multi-grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor), *Proc. Fourth Int. Workshop on Languages and Compilers for Parallel Computing*, pp.856–864 (1991).
7) Keckler, S.W., Dally, W.J., Maskit, D., Carter, N.P., Chang, A. and Lee, W.S.: Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor, *Proc. 25th Int. Symp. on Computer Architecture*, pp.306–317 (1998).
8) Kobayashi, R., Iwata, M., Ogawa, Y., Ando, H. and Shimada, T.: An On-Chip Multiprocessor Architecture with a Non-blocking Synchronization Mechanism, *Proc. 25th EUROMICRO Conference*, pp.432–440 (1999).
9) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proc. 19th Int. Symp. on Computer Architecture*, pp.46–57 (1992).
10) Marcuello, P., Gonzales, A. and Tubella, J.: Speculative Multithreaded Processors, *Proc. 1998 International Conference on Supercomputing*, pp.13–17 (1998).
11) MIPS Technologies, Inc.: MIPS R10000 Processor User's Manual, Version 2 (1996).
12) Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K. and Chang, K.: The Case for a Single-Chip Multiprocessor, *Proc. Seventh Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.2–11 (1996).
13) Park, I., Falsafi, B. and Vijaykumar, T.N.: Implicitly-Multithreaded Processors, *Proc. 30th Int. Symp. Computer Architecture*, pp.39–51 (2003).
14) Smith, M.D., Horowitz, M.A. and Lam, M.S.: Efficient Superscalar Performance through Boosting, *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.248–259 (1992).
15) Sohi, G.S., Breach, S.E. and Vijaykumar, T.N.: Multiscalar Processor, *Proc. 22nd Int. Symp. on Computer Architecture*, pp.414–425 (1995).
16) Vijaykumar, T.N. and Sohi, G.S.: Task Selection for a Multiscalar Processor, *Proc. 31st Int. Symp. on Microarchitecture*, pp.81–92 (1998).
17) Yeh, T-Y. and Patt, Y.: Two-Level Adaptive Branch Prediction, *Proc. 24th Int. Symp. and Workshop on Microarchitecture*, pp.55–61 (1991).
18) Zima, H. and Chapman, B.: *Supercompilers*

for Parallel and Vector Computers, Addison-Wesley Publishing Company, Inc., New York, NY (1991).

(Received September 21, 2005)
(Accepted December 2, 2005)

Akio Nakajima was born in 1978. He recieved the B.E. and M.E. degrees from Nagoya University in 2002 and 2004, respectively. Since 2004, he has been engaged in research and development on disk array subsystems at Systems Development Laboratory, Hitachi, Ltd.

Ryotaro Kobayashi received his B.E., M.E., and D.E. degrees from Nagoya University in 1995, 1997, and 2001, respectively. Since 2000 he has been a research assistant in Nagoya University. His research interests include computer architecture and multithreaded architecture. He is also a member of IEEE and IPSJ.

Hideki Ando received the B.S. and M.S. degrees in electronic engineering from Osaka University, Suita, Japan, in 1981 and 1983, respectively. He received the Ph.D. degree in information science from Kyoto University, Kyoto, Japan, in 1996. From 1983 to 1997 he was engaged in the research and development of digital signal processors for ISDN, microprocessors for inference machines of the Japanese fifth-generation computer systems project, and general-purpose VLIW machines at the LSI Research and Development Laboratory, Mitsubishi Electric Corporation, Itami, Japan. From 1991 to 1992 he was a visiting scholar of Stanford University. In 1997 he joined the faculty of Nagoya University, Nagoya, Japan, where he is currently a Professor in the department of computational science and engineering. His research interests include computer architectures and compilers.

Toshio Shimada received his B.S. and M.S. degrees in Mathematical Engineering and Instrumentation Physics in 1968 and 1970 respectively from Tokyo University. He received his Ph.D. degree in Information Science and Technology in 1992 from Tokyo University. He had worked at Electrotechnical Laboratory from 1970 to 1992. He is currently a professor at Nagoya University. Dr. Shimada's research interests involve low power consumption processors and special purpose LSI.