Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

Performance Improvement Techniques in Tightly Coupled Multicore Architectures for Single-Thread Applications

Keita Doi[1,†1] Ryota Shioya[2,a)] Hideki Ando[2,b)]

Received: September 22, 2017, Accepted: March 6, 2018

Abstract: Current multicore processors achieve high throughput by executing multiple independent programs in parallel. However, it is difficult to utilize multiple cores effectively to reduce the execution time of a single program. This is due to a variety of problems, including slow inter-thread communication and high-overhead thread creation. Dramatic improvements in the single-core architecture have reached their limit; thus, it is necessary to effectively use multiple cores to reduce single-program execution time. Tightly coupled multicore architectures provide a potential solution because of their very low-latency inter-thread communication and very light-weight thread creation. One such multicore architecture called *SKY* has been proposed. SKY has shown its effectiveness in multithreaded execution of a single program, but several problems must be overcome before further performance improvements can be achieved. The problems this paper focuses on are as follows: 1) The SKY compiler partitions programs at a basic block level, but does not explore the inside of basic blocks. This misses an opportunity to find good partitioning. 2) The SKY processor always sequentializes a new thread if the forking core in which it is supposed to be created is busy. However, this is not necessarily a good decision. 3) If the execution of register communication instructions among cores is delayed, the other register communication instructions can be delayed, causing the following thread execution to stall. This situation occurs when the instruction window of a core becomes full. To address these problems, we propose the following three software and hardware techniques: 1) *Instruction-level thread partitioning*: the compiler explores the inside of basic blocks to find a better program partition. 2) *Selective thread creation*: the hardware selectively sequentializes or waits for the creation of a new thread to achieve better performance. 3) *Automatic register communication*: register communication is automatically performed by a small hardware support instead of using instruction window resources. We evaluated the performance of SKY using SPEC2000 benchmark programs. Results on four cores show that the proposed techniques improved performance by 4% and 26% on average (maximum of 11% and 206%) for SPECint2000 and SPECfp2000 programs, respectively, compared with the case where the proposed techniques are not applied. As a result, performance improvements of 1.21 and 1.93 times on average (maximum of 1.52 and 3.30 times) were achieved, respectively, compared with the performance of a single core.

Keywords: multicore, single-thread performance

1. Introduction

Current multicore processors increase throughput by executing multiple programs in parallel. Good scalability is achieved because the programs executed are independent. Transistor counts on a single die increase according to Moore's law. Thus, modern processors integrate increasingly large numbers of cores on a single die, further increasing the throughput.

The breakdown of Dennard scaling where the clock frequency does not scale even with advances in LSI technology makes it necessary to use multiple cores to improve single program performance. However, it is difficult to use even a small number of cores to reduce the execution time of a single program. **Figure 1** shows the performance improvement for four threads when using four cores relative to a single thread. The programs are parallelized by Intel Composer XE [*1]. The benchmark programs are selected from the SPEC2000 suite and are used for evaluation in Section 4. "GM int" and "GM fp" are the geometric means of the performance improvements for integer and floating-point programs, respectively. The processor used for the evaluation is an Intel Xeon E5-2670. The main memory size is 128 GB. As shown in the figure, no performance improvement is observed for the integer programs. For floating-point programs, a large performance improvement is observed only in *mgrid*.

There are a variety of reasons for this poor performance improvement. Matrix-operation-based scientific applications can be successfully parallelized in many cases. In contrast, general applications are often difficult to parallelize, owing to irregular data dependencies, complex control flows, and difficult memory aliases. Also, communication among the cores must be

[1] Department of Electrical Engineering and Computer Science, Nagoya University, Nagoya, Aichi 464–8603 Japan
[2] Department of Information and Communication Engineering, Nagoya University, Nagoya, Aichi 464–8603 Japan
[†1] Presently with Okuma Corporation
[a)] shioya@nuee.nagoya-u.ac.jp
[b)] ando@nuee.nagoya-u.ac.jp

[*1] Intel Composer XE is a successor to the Intel C++ Compiler, and is known to have a high capability for sophisticated optimizations. The aggressiveness of the parallelization can be specified using a compiler option. Unfortunately, however, aggressive parallelization does not necessarily achieve better performance when using this compiler. We select the most suitable aggressiveness by attempting to compile each benchmark program with variations.

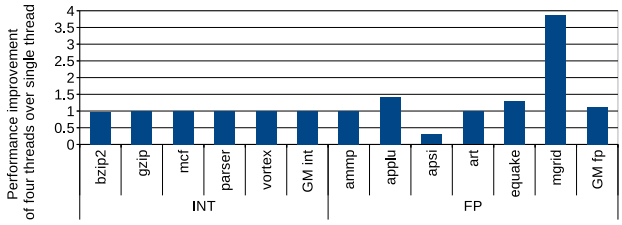


Fig. 1 Performance improvements for four threads relative to a single thread using Intel Composer XE.

performed via the memory. This long-latency communication means that the compiler selects only program partitioning with few inter-thread dependences per thread length. Therefore, the compiler tends to partition the program into coarse-grain threads with either zero or few data dependencies. Furthermore, although the memory context is shared among cores, the register context must be copied to the forking core when a thread is created, and this task is time-consuming. Again, this makes the compiler select only coarse-grain threads, losing many of the benefits of parallelization with fine-grain threads.

Tightly coupled multicore architectures have been proposed to improve single-program execution performance [4], [9], [12], [14], [18], [21], [23], [25]. These architectures commonly support inter-core register communication via a bus connecting cores. Binaries are usually generated for these architectures by their own parallelizing compiler. Low-latency inter-core communication and light-weight thread creation allow the compiler to partition a program into fine-grain threads, even if there are many inter-thread data dependences. Dramatic improvements to single-thread performance through single-core improvements, including increasing the clock frequency, are difficult. Therefore, tightly coupled multicore architectures, such as *SKY* [9], are now worth reconsidering. The main feature of SKY is *non-blocking inter-core register communication*, which allows instruction-level parallelism (ILP) and thread-level parallelism (TLP) to be exploited without interfering with each other. Using the automatic parallelization compiler for SKY, a program is partitioned into fine-grain threads at a basic block level, and the single-program execution performance is improved.

Although SKY has achieved a relatively high performance level, several problems must be overcome for further performance improvement. In this paper, we solve the following problems:

(1) The SKY compiler misses an opportunity to find good partitioning for high performance because partitioning is at a basic block level and the compiler does not explore the inside of basic blocks.

(2) The conventional SKY processor sequentializes a new thread to be created if the core where the new thread is supposed to be created is busy. However, this misses another opportunity for achieving higher performance because it can be beneficial if the thread creation process waits for the forking core to become idle.

(3) If the execution of inter-core register communication instructions is delayed, the other register communication instructions can be delayed, causing the following thread execution to stall. This situation occurs when the instruction window of a core becomes full.

The present paper proposes the following solutions for each problem:

(1) *Instruction-level partitioning*: The compiler attempts to search for a better partitioning boundary inside a basic block. In this process, instructions are reordered if necessary.

(2) *Selective thread creation*: The hardware dynamically selects between the options of sequentializing a new thread or waiting for the forking core to become available to achieve better performance.

(3) *Automatic register communication*: We prepare support hardware to execute register communication instructions. This hardware automatically sends registers without using instruction window resources. Because the instruction window is not used, the instruction window never becomes full due to register communication instructions, allowing the following instruction to be executed.

We evaluated the effectiveness of the three optimizations described above using SPEC2000 benchmark programs. The results of execution on four cores show that the proposed optimizations improve the performance by 4% and 26% on average (maximum of 11% and 206%) for integer and floating-point programs, respectively.

The remainder of this paper is organized as follows. Section 2 gives an overview of the SKY architecture and its automatic parallelizing compiler. The three problems and our proposed solutions are described in Section 3, and evaluation results are presented in Section 4. Related work is discussed in Section 5, and our conclusions are given in Section 6.

2. Overview of SKY

This section provides an overview of SKY. Section 2.1 describes the multithreading model that is assumed in SKY. Section 2.2 gives an overview of the hardware organization. Finally, Section 2.3 explains the compiler used for SKY.

2.1 Multithreading Model

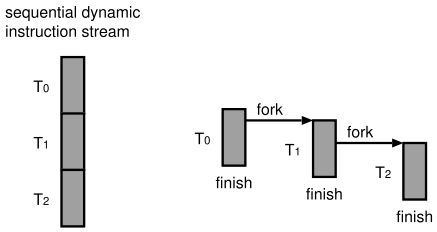
The multithreading model in SKY is not a general model, but is constrained to a certain degree to simplify both the hardware and the compiler. The model is similar to that used in the *multiscalar* architecture [21]. The features of this model are described as follows.

First, a thread consists of a dynamically continuous instruction stream and not of the disjointed parts of a stream. As shown in **Fig. 2** (a), each thread is simply a single part of the dynamic stream. Second, each thread is created *sequentially*, as shown in Fig. 2 (b). Multiple threads are executed in parallel in an overlapping manner. Thread creation and termination are carried out using special instructions called `fork` and `finish`, which are inserted by the compiler.

2.2 Hardware Overview

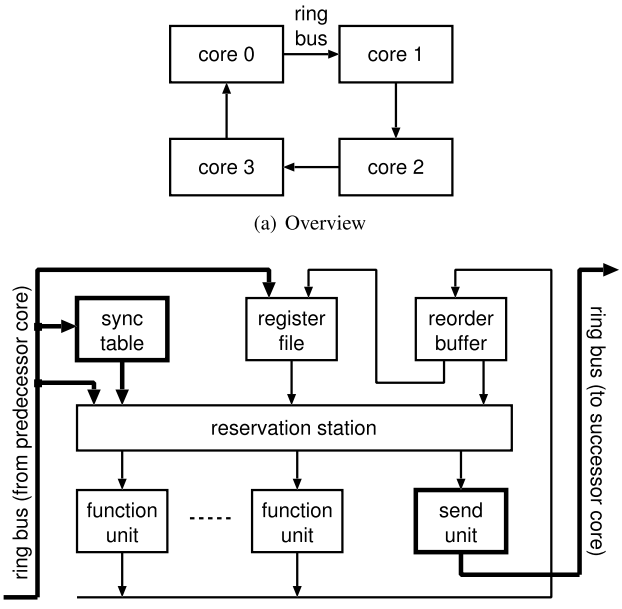
This section gives an overview of the SKY hardware. For further details, see Ref. [9].

SKY has a multicore architecture; however, unlike conventional multicores, each core is tightly coupled, to enable register



(a) Thread partitioning (b) Parallel thread execution

Fig. 2 Multithreading model in SKY.



(a) Overview

(b) Core architecture

Fig. 3 SKY processor architecture.

communication using a unidirectional ring bus, as shown in **Fig. 3** (a). Although core architecture modification is required for this tightly coupled architecture, the modifications needed are very modest. Thus, migration from existing architectures is simple. In the explanation of this section, we assume use of the Intel P6-type architecture [7] (i.e., reservation station-based architecture) as a base core architecture.

If a `fork` instruction is executed in a thread on a particular core, then a new child thread is created in its successor core in the direction of the ring bus. Note that `fork` instructions are executed non-speculatively in terms of their control dependencies, because rolling back thread creation as a result of a misspeculation complicates the hardware.

The ring bus is responsible for inter-thread register communication. Any specific core can send register values only to its successor core and can receive register values only from its predecessor core. To send a register value to a non-adjacent core, the cores that lie between the sending and receiving cores must relay the communicating register. To support register communication, each core contains two unique structures: a *send unit* and a *sync table*. These are shown in Fig. 3 (b) and are highlighted with thick lines. The send unit executes a special instruction called `send`, which sends the source register value to the successor core. Physically, this unit simply drives it to the ring bus. The sync table (see Fig. 11) is responsible for receiving registers. The sync table also plays a central role in *non-blocking register communication*, where an instruction that is waiting for a register value does not block the execution of subsequent instructions. This allows instructions to proceed down the pipeline to be dispatched to the reservation station and to wait to receive its register value. The instruction waiting in the reservation station receives its register value sent later using the existing forwarding mechanism. The tag is supplied from the sync table at receive. The waiting instruction also obtains the tag from the sync table at decode. This non-blocking feature does not prevent ILP from being exploited within a thread to enable exploitation of TLP.

Rolling back register sending due to misspeculation also complicates the hardware. Therefore, similar to the `fork` instructions, `send` instructions are executed non-speculatively in terms of their control dependencies.

2.3 Compiler Overview

This section overviews the compiler for SKY. The compiler for SKY first partitions a program into multiple threads so that the performance increase gained by executing these threads in parallel is maximized. It then computes the inter-thread communication registers. The following three sections explain the three steps in these compilation phases.

2.3.1 Finding Candidates for Program Partitions

We partition programs at a basic block level. This fine-grain partitioning can potentially extract more parallelism from programs than the other levels (e.g., loop or function levels) according to the limit studies of TLP [10], [13], [15], [16]. Note that fine-grain partitioning at a basic block can flexibly exploit the opportunity at various levels. In other words, the basic block partitioning can exploit parallelism among functions and among loop iterations, as well as purely among basic blocks.

As a candidate for program partitioning, we focus on a *control-equivalent* basic block pair, X and Y, where X and Y include a `fork` instruction and child thread start point, respectively [20]. We say that blocks X and Y are control-equivalent, if X *dominates* [2] Y and Y simultaneously *post-dominates* [20] X. The main reasons that we focus on the control equivalence are 1) post-domination ensures that the control always reaches Y if X is executed, and thus child thread execution is non-speculative in terms of control dependence (note that SKY does not perform speculative thread creation, as described in Section 2.2), and 2) basic block relationships with control equivalence are accommodated by general program structures (e.g., if-then-else and loops, where (if-block, merging-block) pair and (head-block, tail-block) pair are control-equivalent, respectively).

2.3.2 Estimation of Performance Gain for Partition Candidates

The reduction in program execution time caused by execution of two parallel threads is the number of clock cycles in the child thread that are executed in parallel with the parent thread. Our compiler approximates this number based on the number of sequentially-executed instructions in the child thread that can be executed in parallel with the parent thread. We call this number the *performance gain*.

The performance gain is obtained based on the *dependence distance*, which is the number of instructions between two instructions with a data dependency relationship. If there is a *single* data dependency over different threads (`def`→`use`), the performance gain is obviously equal to the dependence distance between `def` and `use`. Because there are generally multiple control paths between `def` and `use`, we compute the dependence distance using the average distance weighted by the execution probability for each path. The execution probability is obtained for each path by profiling.

In general, there are multiple register and memory dependences. We determine the performance gain for a specific thread partitioning as being the minimum dependence distance for all dependences (if there is no inter-thread data dependence, the performance gain is the number of instructions between `fork` and `finish` instructions). The compiler then eliminates thread candidates with performance gains that are lower than a predetermined threshold. The compiler then selects the thread candidates to observe the constraints of the multithreading model in SKY, which are described in Section 2.1. Finally, it inserts the `fork` and `finish` instructions for the selected threads.

2.3.3 Inserting Send Instructions

If a register value that is defined in thread T_i is used in thread T_j, the compiler then inserts a `send` instruction in thread T_i to send the register value. We call such a `send` instruction a *true* `send` instruction. Technically, the compiler inserts the true `send` instructions by computing the *reaching of definitions* [2].

As described in Section 2.2, if at least one thread exists between threads T_i and T_j, then `send` instructions are also required in the threads between T_i and T_j. These `send` instructions are used to simply transfer the register values from the parent thread to the child thread. We call this type of `send` instruction a *transfer* `send` instruction. The compiler inserts the transfer `send` instructions immediately after a `fork` instruction.

3. Problems and Solutions

This section describes the three problems that we discuss in this paper and explains the techniques used to solve each problem.

3.1 Instruction-Level Partitioning

3.1.1 Problem

Because the SKY compiler partitions a program on basic block basis, the thread start point is always at the beginning of a basic block. While this level in program partitioning is a smaller grain when compared with most other parallelizing compilers, it does sometimes miss an opportunity to find good partitioning with high performance gain.

Figure 4 shows an example for this case. Figure 4 (a) shows a program fragment that consists of several basic blocks, $B_0 \ldots B_i, B_j \ldots B_k$. Blocks B_0 and B_j are control equivalent, and the path through B_i is a frequently executed path between B_0 and B_j. There is a dependence chain consisting of instructions `i0`, `i1`, and `i3`, while instructions `i2` and `i4` are independent of the other instructions in $B_0 \ldots B_i$.

Now consider program partitioning into T_0 and T_1 on a basic block basis, as shown in Fig. 4 (b), where the `fork` and `finish` instructions are inserted at the beginning of the basic blocks B_0 and B_j, respectively. As shown in the figure, because there is a dependence between instructions `i0` and `i1`, and the path through B_i is a frequently executed path, little performance gain is obtained.

However, we now consider reordering of the instructions in basic block B_j, where we place the instructions `i1` and `i3` on the dependence chain at the beginning of the basic block, then place the `finish` instruction, and finally place the independent instructions `i2` and `i4`, as shown in Fig. 4 (c). In this partitioning, the instructions `i1` and `i3`, which depend on `i0`, are included in thread T_0, and no instruction in thread T_1 depends on the instructions in thread T_0. Therefore, a high performance gain can be obtained by executing T_0 and T_1 in parallel.

3.1.2 Solution

As described in the previous section, finding a thread boundary within a basic block combined with reordering of the instructions can improve overall performance. We carry out this task by finding a *cut* in a dataflow graph such that the minimum weight of the edges crossing the cut is maximized. First, we model the problem, and then we explain our algorithm. Finally, a small working example is given.

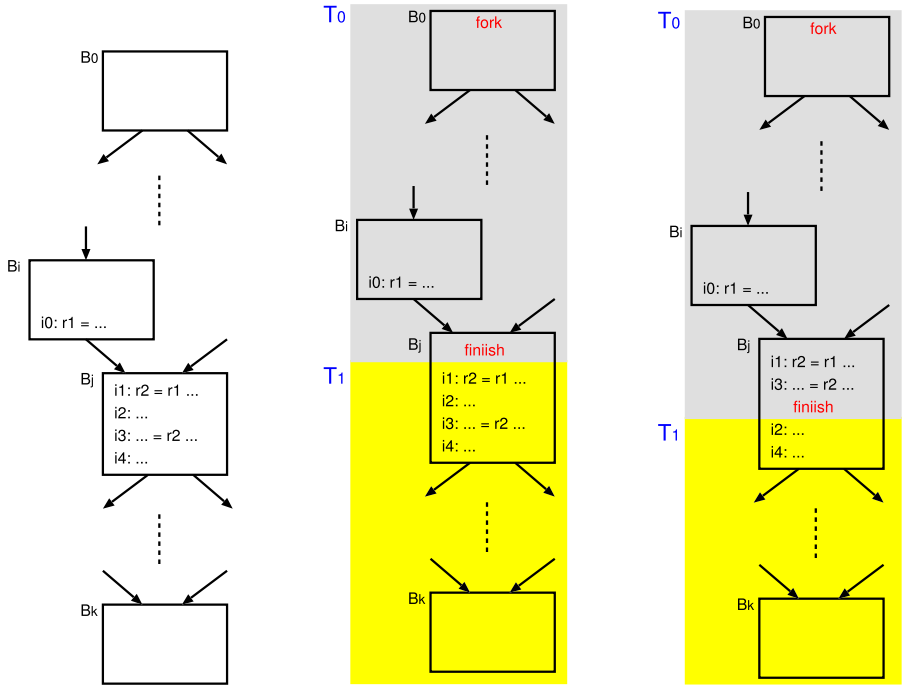
Modeling the problem. We represent the instructions and their data dependencies in the target basic block in which we attempt to find a thread boundary as a weighted directed graph G. This graph G is generally called a dataflow graph, where a node and an edge represent an instruction and the data dependence, respectively. The weight of the edge is the dependence distance in the original code if the associated data dependence type is true, or it is ∞ otherwise (i.e., anti- or output dependence). In addition, there are special nodes, in the form of a start node v_s and an end node v_e, which represent those nodes that collect all instructions preceding and succeeding the target basic block, respectively. If there are true data dependences between an instruction in the target basic block and an instruction in nodes v_s or v_e, then they are connected with an edge with the weight of the dependence distance. However, if the dependence type is an anti- or output dependence, they are ignored because the order of v_s (or v_e) and the nodes in the target basic block do not change in this algorithm, because we only reorder instructions in the target block.

Under these modeling conditions, the problem can be described as follows:

- *Goal*: Find a cut that partitions G into subgraphs S and T such that the evaluation function score is maximized.
- *Evaluation function*: Output the minimum weight in the cut-set.
- *Constraint*: The direction of any edge that crosses the cut must be from S to T.

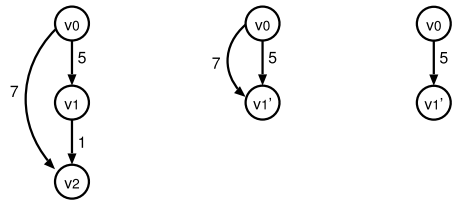
Semantically, S and T represent the instructions included in the parent and child threads, respectively, and the evaluation function score represents the performance gain with respect to these threads.

Reducing the graph. In preparation for execution of the core of our algorithm, we reduce the input graph G to simplify the core algorithm. This reduction process first picks up two nodes (excluding v_s and v_e) connected by an edge with a *finite* weight, and then combines these two nodes into a single node. Inclusion



(a) Code fragment (b) Partitioning on a basic block basis (c) Partitioning with re-ordered instructions

Fig. 4 Program partitioning on a basic block basis and optimal partitioning on an instruction-level basis with instruction reordering.



(a) Before reduction (b) Combining the nodes (c) After reduction

Fig. 5 Graph reduction procedure.

of an edge that connects these nodes to the cut-set is obviously not beneficial, because the number of instructions within a basic block is generally small, and thus the performance gain is small.

We explain this reduction using the example shown in **Fig. 5**. First, we pick up nodes v_1 and v_2, which are connected by an edge with a weight of 1. We remove this edge and reduce the two nodes into node v'_1, as shown in Fig. 5 (b). In this new graph, there are two edges between v_0 and v'_1. Because the evaluation function outputs a minimum weight for the edges in a cut-set, we take the edge with the smaller weight, thus converting the graph as shown in Fig. 5 (c).

During the reduction procedure, if edges that connect nodes with different directions appear, then we also reduce these nodes, because the cut crossing these edges violates the constraint that was described previously (i.e., the direction of any edge crossing the cut must be from S to T).

Because any edge with a finite weight within the target basic block is removed by this reduction procedure, we can obtain the following lemma:

Lemma 1. Edges with a finite weight are only those that lie between the nodes in the target basic block and v_s or v_e, if they exist.

Core algorithm. The algorithm starts with the initialization, letting the cut be one that crosses the edges from v_s to the other nodes. This cut represents our original program partitioning, which was based on the basic blocks. The algorithm evaluates the cut-set, and then carries out the following steps:

(1) The algorithm removes an edge with a minimum weight from the cut-set by node motion from T into S. Because this edge determines the evaluation function score, this removal process can improve the evaluation score. The associated node motion newly adds the edge that is connected to the moved node to the cut-set.

(2) If the above node motion creates an edge that violates the constraint (see the section of "modeling the problem"), the algorithm then removes this edge from the cut-set by moving the problem node from T into S.

(3) The algorithm evaluates the cut-set. If the evaluation score is improved, the algorithm repeats the steps above; otherwise, it stops this loop procedure and outputs the cut-set determined in the previous iteration as an optimal solution.

This hill-climbing algorithm produces an optimal solution because:

(1) The algorithm removes the edges in an ascending order with respect to weights. Therefore, it basically always improves the solution by repeating the node motions.

(2) If any node motion causes the evaluation score to decrease, this means that the edge (the start point of which is the moved node) has been inserted into the cut-set, and its weight becomes the new minimum. However, this edge is connected to v_e from lemma 1, and thus cannot be removed from the cut-set. Consequently, no node motion exists that

```
1  # initialize
2  G = (S, T), where S = {v_s} and T = {v_e and nodes in target BB};
3  cutset = edges crossing from S to T;
4  cutset_prev = cutset;
5  score_prev = eval_func(cutset_prev);
6
7  # main loop
8  while (true) {
9    edge_min = edge in cutset with a minimum weight;
10   node_T = node at the endpoint of edge_min;
11   if (node_T is v_e) break; # v_e cannot be moved
12   move node_T from T to S and update the cutset;
13
14   # remove violating edge from the cut-set
15   foreach violating edge in the cutset {
16     move node at the startpoint of the violating edge from T to S and update the cutset;
17   }
18
19   # evaluation
20   score = eval_func(cutset);
21   if (score < score_prev) break;
22   cutset_prev = cutset;
23   score_prev = score;
24 }
25 output cutset_prev;
```

Fig. 6 Core algorithm for instruction-level partitioning.

would improve the solution.

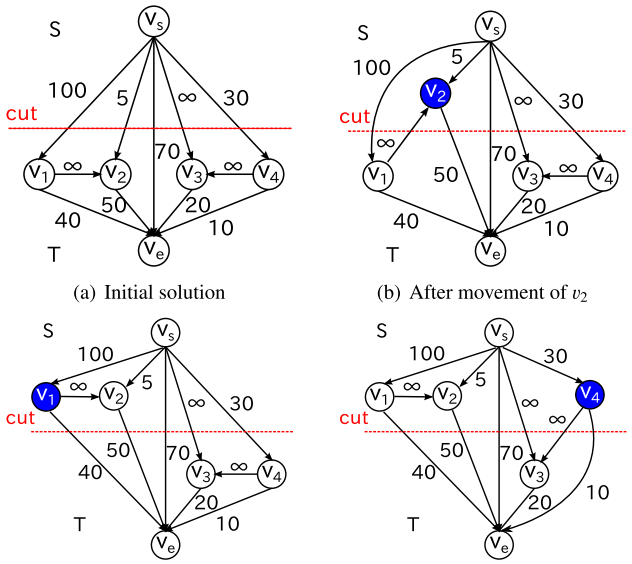
We formally describe the algorithm using the pseudo code shown in **Fig. 6**. The algorithm initializes the cut-set as a set of edges crossing from v_s to the other nodes (lines 2 and 3), and then evaluates this cut-set (line 5). The algorithm then finds the edge that is crossing the cut with the minimum weight (line 9), and attempts to remove this edge from the cut-set by moving the node that forms the end point of this edge from T to S. If this node is v_e, then the iteration of the algorithm is terminated (line 11), because v_e must be in T and thus cannot be moved. Otherwise, the algorithm moves this node and updates the cut-set (line 12). If this node motion creates an edge that violates the constraint, then the algorithm moves the node at the start point of each violating edge from T to S to solve the violation and then updates the cut-set (lines 15 and 16). Finally, the iteration evaluates the cut-set (line 20). If the evaluation has not improved, the algorithm then ends (line 21) by outputting the cut-set from the previous iteration (line 25); otherwise, the cut-set and the evaluation score are saved and the algorithm loop is repeated (lines 22 and 23).

Working example. We describe a working example for the core algorithm using **Fig. 7**.

Figure 7 (a) shows the initial state. The graph is cut immediately below node v_s. The edge with the minimum weight in the cut-set is $v_s \rightarrow v_2$, which is 5.

The algorithm then moves node v_2 from T to S to remove this edge, as shown in Fig. 7 (b). The edge $v_1 \rightarrow v_2$ now violates the constraint. The algorithm thus moves node v_1 from T to S to remove this edge, as shown in Fig. 7 (c).

The algorithm then evaluates the cut-set and obtains a score of 30 (the weight of the edge $v_s \rightarrow v_4$). Because the evaluation score has been improved, the algorithm continues. The algorithm moves node v_4 from T to S, as shown in Fig. 7 (d). There is no violating edge, so the algorithm evaluates the cut-set and obtains



(a) Initial solution (b) After movement of v_2

(c) After movement of v_1 (d) After movement of v_4

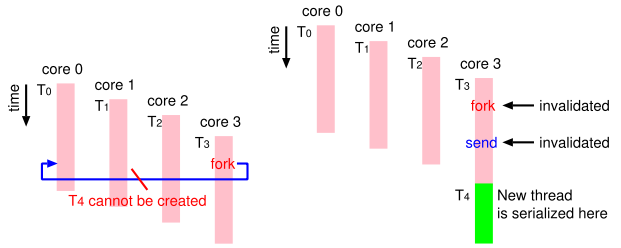
Fig. 7 Working example of core algorithm for instruction-level partitioning.

a score of 10. This evaluation score is worse than that in the previous iteration. Thus, the partition obtained in the previous iteration, $S = \{v_s, v_1, v_2\}$ and $T = \{v_3, v_4, v_e\}$, is the optimal solution.

3.2 Selective Fork

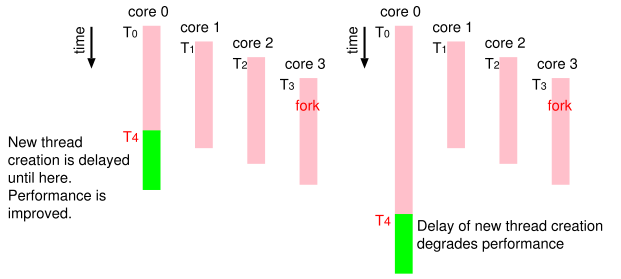
3.2.1 Problem

When a `fork` instruction is executed but the successor core is busy with execution of another thread, a new child thread cannot be created at this time. We explain how the conventional SKY architecture handles this case using **Fig. 8**. In this example, the processor has four cores, and threads T_0 to T_3 are executed in cores 0 to 3, respectively. Now, a `fork` instruction is executed in T_3 in core 3. However, the successor core (core 0 in this case) is



(a) Thread T_4 cannot be created (b) Thread T_4 is serialized after T_3

Fig. 8 Thread handling in the conventional SKY architecture when a new thread cannot be created because the successor core is busy.



(a) Creation of T_4 in core 0 improves overall performance (b) Creation of T_4 in core 0 degrades overall performance

Fig. 9 Delaying the creation of a thread until the target core becomes idle.

busy, and thus the new thread T_4 cannot be created at this time, as shown in Fig. 8 (a). The fork instruction is then invalidated, and the thread T_4 that should have been created is executed in core 3 after T_3 is finished, as shown in Fig. 8 (b). We call this behavior *thread serializing*. Note that thread T_4 is *not created* in this case, but is simply executed after T_3 finishes. In this case, the associated `send` instructions are also invalidated, where the `fork` and associated `send` instructions are identified by the thread ID encoded in each instruction [9].

While this policy in the conventional SKY architecture makes the hardware simpler, an opportunity for performance improvement is being missed. In other words, there is a case where if the creation of T_4 in core 0 is delayed until T_0 is finished, the performance can then be improved as shown in **Fig. 9** (a). However, this is not always the case. As Fig. 9 (b) shows, if T_3 is finished earlier than T_0, the creation of T_4 in core 0 degrades the performance, and the execution of T_4 in core 3 as per conventional method is the better choice.

3.2.2 Solution

Delaying the creation of a thread is not necessarily beneficial, as described above. Specifically, if the thread that is currently running in a core where a new thread is to be created finishes earlier than the thread where the `fork` instruction is executed, then delaying the creation of the thread is beneficial; otherwise, it is beneficial when the new thread is serialized after the forking thread. Having considered these situations, we propose the following scheme:

- If a `fork` instruction is executed, but the successor core is busy, then the thread creation is *reserved*. `Send` instructions that send register values to the reserved thread are executed, and the register values and numbers are buffered.
- If the thread in the successor core finishes earlier than the forking thread, then the reserved thread is actually created in the successor core, and the buffered register values and numbers are sent to the successor core; otherwise, the thread reservation is canceled and the thread is serialized after the forking thread. In addition, the buffered registers to be sent are invalidated.

To implement this scheme, we prepared two structures. The first is a single register called a *thread creation reservation register* (TCRR), which holds and reserves fork instructions with a valid bit. The other is a *register send reservation buffer* (RSRB), which is a FIFO buffer that has the same number of entries as the number of logical registers [*2], where each entry holds a register number and a value to send.

If a `fork` instruction is executed and the successor core is busy, it is then written into the TCRR. If an associated `send` instruction is executed, then the register number and the value that should have been sent to the successor thread are written into the RSRB. If the thread in the successor core finishes and the core becomes idle before the forking thread finishes, then a child thread is created by referring to the TCRR and the TCRR is then invalidated. In addition, the registers that were held in the RSRB are sent to the successor core. Conversely, if the forking thread finishes while the successor core is still busy and the TCRR is valid, the TCRR and the RSRB are simply invalidated. This means that the child thread must be serialized after the forking thread. Note that in the child thread creation case, the `send` instructions that are executed after the child thread creation send the register values to the successor core as usual.

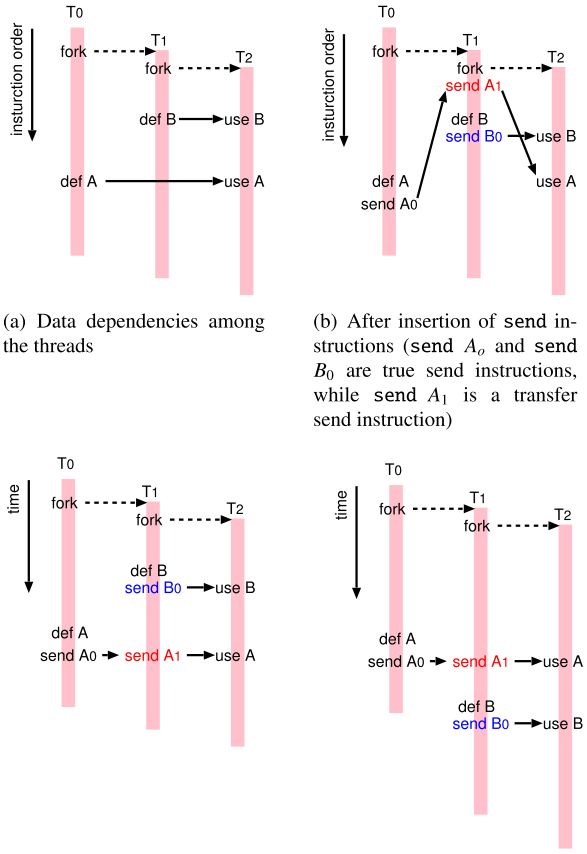
3.3 Automatic Register Send

3.3.1 Problem

As described in Section 2.3, `send` instructions are inserted in the last phase of compilation. This means that although the compiler considers the data dependency between the definitions and the uses (i.e., `def`→`use`), it does not consider the dependency at runtime, where the `send` instructions are involved (i.e., `def`→`send`→`use`). True `send` instructions do not affect the actual performance gain significantly, because the point at which a definition is determined to reach the use points is usually close to the definition point. However, the transfer `send` instructions can affect the actual performance gain in a specific situation.

We explain this situation using the example shown in **Fig. 10**. Figure 10 (a) shows two data dependencies among three threads, T_0, T_1, and T_2, i.e., `def A`→`use A` and `def B`→`use B`. In Fig. 10 (b), the `send` instructions are inserted for these two dependencies, where `send` A_0 and `send` B_0 are true `send` instructions, while `send` A_1 is a transfer `send` instruction. Note that the transfer `send` instructions are inserted immediately after the `fork` instruction, as described in Section 2.3.3. Because `send` A_1 is fetched early, i.e., before `send` A_0 is executed, the execution of `send` A_1 is delayed. However, this delay does not necessarily delay the execution of `send` B_0 because of the non-blocking

[*2] Only the registers that are *alive* [2] at the start point of the child thread in each thread must be sent, as described in Section 2.3.3. Therefore, the maximum number of entries required for the RSRB is the total number of logical registers.



(a) Data dependencies among the threads

(b) After insertion of send instructions (send A_o and send B_0 are true send instructions, while send A_1 is a transfer send instruction)

(c) Case where send A_1 and send B_0 can be reordered

(d) Case where send A_1 and send B_0 cannot be reordered

Fig. 10 Performance degradation due to transfer send instructions.

register communication scheme that uses the instruction execution reordering capabilities of superscalar processors. Note also that no instruction depends on the transfer send instructions within the thread, and thus any instruction that follows the transfer send instructions can be executed before execution of the transfer send instructions. Figure 10 (c) shows this instruction execution reordering, where def B and send B_0 are executed before send A_1. The delay of send A_1 does not adversely affect the performance.

However, this instruction reordering is constrained by the instruction window size, and particularly by the reorder buffer (ROB), of superscalar processors. If send A_1 is stalled for a long time at the head of the ROB, the ROB becomes clogged and finally becomes full. If send B_0 is separated from send A_1 by more than the ROB size in the instruction count, then send B_0 cannot be inserted into the ROB. As a result, the execution of send B_0 is delayed until the execution of send A_1, without being reordered, as shown in Fig. 10 (d). The delay in execution of send B_0 causes a delay in execution of use B, and thus degrades the performance.

3.3.2 Solution

We solve the problem described in Section 3.3.1 using a hardware support. Specifically, when a transfer send instruction is decoded, and its source register has not yet been received, the transfer send instruction is immediately removed, and the hardware support instead will automatically send the register later when the register is received from the predecessor core. Because transfer send instructions are not inserted into the instruction window, the problem that was described in Section 3.3.1 does not occur.

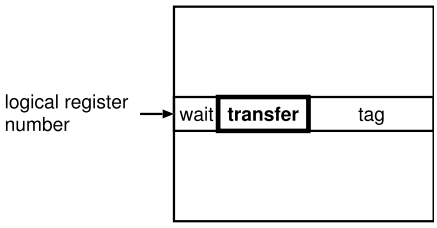


Fig. 11 Sync table with a transfer flag.

As the hardware support, we add an extra flag, called a *transfer*, to each entry in the sync table, as shown in **Fig. 11** [*3]. When a thread is created, all transfer flags in the sync table are cleared. If a transfer send instruction is decoded and its source register has not been received, then the transfer flag of the associated entry is set. The transfer send instruction is then removed and is thus not inserted into the instruction window.

In contrast, when a register is received from the predecessor core, the transfer flag in the associated entry in the sync table is then checked.

- If the flag is set, it is found that the transfer send instruction that should have sent the register has been removed. Thus, the received register value is simply forwarded to the successor core through the ring bus.
- If the flag is not set, then the received register value is written into the register file or is invalidated, depending on the *wait* flag, as normal [*4]. In the former case, if a transfer send instruction is fetched later, it will then send the received register to the successor core.

Readers sometimes misunderstand that the scheme using the transfer flag is not new because they believe that the multiscalar architecture [21] implements this scheme using the *forward bit* [3]. However, it is a tag attached to a define instruction, and the define instruction with the forward bit sends the destination register to the succeeding core. In other words, it corresponds to the combination of a define instruction and the true send instruction in the SKY architecture. Therefore, it is not related to the problem addressed in this section.

4. Evaluation

Section 4.1 describes our evaluation environment, while Section 4.2 determines the latency of send and fork instructions through a circuit simulation for the following performance evaluation. Section 4.3 evaluates the performance when using the optimization methods described thus far. Section 4.4 shows the performance scalability relative to the number of cores. Finally, Section 4.5 discusses on the evaluation results.

4.1 Evaluation Environment

For evaluation, we used five integer programs and six

[*3] The total number of entries of the sync table is the number of logical registers [9], which can be easily inferred from the fact that the index of the table is the logical register number, as shown in Fig. 11

[*4] The normal behavior of SKY controlled by the wait flag is not detailed here, because it is not directly related to the topics in this paper. For details, see Ref. [9].

Table 1 Processor configuration.

core	
Pipeline width	4-instruction width for each of fetch, decode, issue, and commit
Reorder buffer	128 entries
Res. station	64 entries
Load/store queue	64 entries
Branch prediction	4-bit history, 1K-entry BHT and 16K-entry PHT PAp 10-cycle misprediction penalty
Function unit	4 iALU, 1 iMULT/DIV, 4 Ld/St, 4 fpALU, 1 fpMULT/DIV/SQRT, 4 send
caches and main memory	
L1 I-cache	64 KB, 2-way, 32 B line
L1 D-cache	64 KB, 2-way, 32 B line, 2-cycle hit latency
L2 cache	8 MB, 4-way, 64 B line, 36-cycle hit latency
Main memory	300-cycle latency

floating-point programs from the SPEC2000 benchmark suite, which were successfully compiled by the cross-compiler GCC when targeting the SimpleScalar PISA [19] [*5] with options -O6 -funroll-loops, and which could also be correctly compiled by our SKY parallelizing compiler. The SKY compiler performs the tasks described in Section 2.3, and generates the `fork`, `finish`, and `send` instructions, using the SimpleScalar PISA binaries as inputs. In the performance gain computation phase, profiling is carried out using the *train* inputs (SPEC recommends using *train* inputs for feedback directed optimization of a compiler [22]). The compilation is automatic, with no need for manual intervention.

For performance evaluation, we built a simulator. We used the *ref* inputs for the evaluation, where we simulated the 1B instructions selected by SimPoint 3.2 [5]. The configurations of the core, the caches, and the main memory are given in **Table 1**. The processor has four cores if not explicitly specified. The caches are currently shared among the cores.

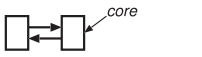
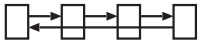
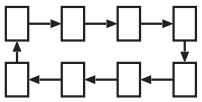
4.2 Latency of Send and Fork Instructions

Before the performance evaluation, we determine the latency of `send` and `fork` instructions. Although the operation of these instructions is simple, they need inter-core communication. Because the delay of long wires is significant in modern LSI technology [6], [17], the latency of these instructions is long. We evaluate the inter-core wire delay using an HSPICE simulation with a 16 nm predictive transistor model [1] developed by the Nanoscale Integration and Modeling Group of Arizona State University. We assume the resistance and capacitance per unit length of the wire that are predicted by the International Technology Roadmap for Semiconductors [8]. In addition, we assume the Intel Skylake with 3 GHz clock frequency as a core size, where the height is longer than the width. Because the Skylake is fabricated using 14 nm LSI technology, we linearly scale the size for the simulation assuming 16 nm LSI technology.

Table 2 shows the core placement and the evaluated latency of `send` and `fork` instructions for the different numbers of cores for which we evaluate performance in the following sections. Regarding four and eight cores, we evaluated the configurations of one- and two-dimensional placement. Table 2 shows the latency of better configurations in terms of the performance. In the HSPICE simulation, we assume that the length of the communication wires between horizontally adjacent cores (short path) is the Skylake core width, while that between vertically adjacent cores (long path) is the Skylake core height. In the four-core case, we assume that the wire length of the long path from the rightmost core to the leftmost core is three times the Skylake core width.

[*5] To the best of our knowledge, there is no cross-compiler that targets PISA and can correctly compile SPEC2006 programs. Thus, we used the SPEC2000 benchmark suite.

Table 2 Latency of `send` and `fork` instructions.

core placement	latency of send and fork inst (cycles) short path	long path
	2	N/A
	2	6
	2	3

4.3 Performance

Figure 12 shows the performance improvement with four threads compared to single-thread execution using a single core. The left and right bars represent the performance improvement without and with the optimizations, respectively.

As shown in Fig. 12, in the SPECint2000 programs, the average performance improvement over the single thread is 16% without optimization, but optimization improves it by 4.4% (maximum of 11%). These programs are known as hard-to-parallelize programs (e.g., the performance improvement is 1.05 in Ref. [24]), but a resulting performance improvement of 21% is achieved on average (maximum of 52%).

In the SPECfp2000 programs, a relatively scalable performance is achieved even without optimization in many programs; the average performance improvement compared to a single thread is 53%. By applying the optimizations, the overall performance is significantly improved by 26% (maximum of 206%) when compared with the case without optimization, and thus a more scalable performance is achieved. The resulting performance improvement is as much as 1.93 times on average (maximum of 3.30 times).

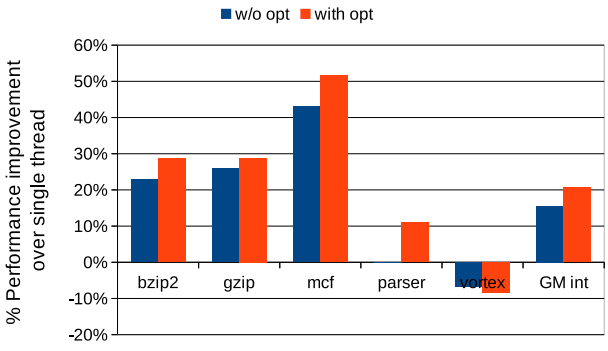
Figure 13 shows the contributions of each optimization to the overall performance. The contribution of a specific single optimization, $Contrib(opt)$, is given by the following equation:

$$Contrib(opt) = (Perf_{full} - Perf_{elim}(opt))/Perf_{full},$$

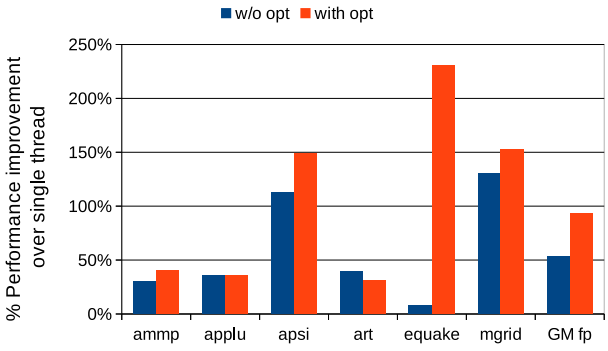
where $Perf_{full}$ and $Perf_{elim}(opt)$ are the performance with full optimization and the performance with a single optimization, opt, eliminated, respectively.

In the SPECint2000 programs, as shown in Fig. 13 (a), the automatic register send (auto-send) is significantly effective in *bzip2*, and boosts the performance by 9.6%. Instruction-level partitioning (IL part) and the selective fork (sel fork) are also effective in several programs.

In the SPECfp2000 programs, as shown in Fig. 13 (b), IL partitioning is highly effective in *equake*, with a contribution of as much as 65%. The selective fork is widely effective;

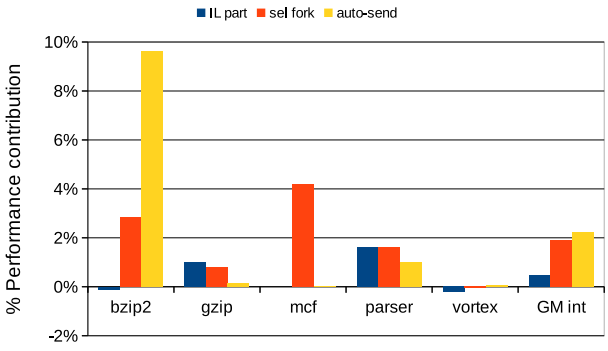


(a) SPECint2000

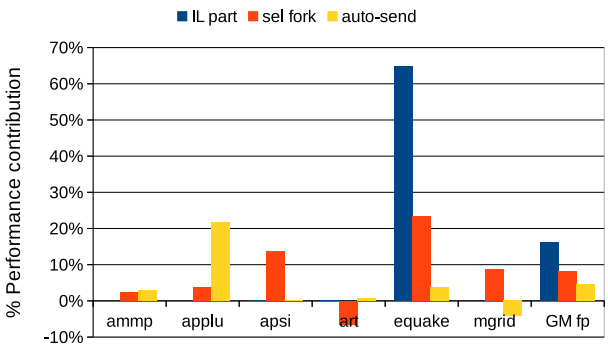


(b) SPECfp2000

Fig. 12 Performance improvement with four threads over a single thread execution.



(a) SPECint2000



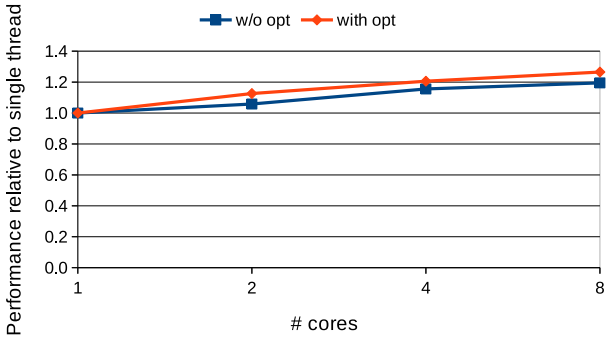
(b) SPECfp2000

Fig. 13 Performance contributions of each optimization.

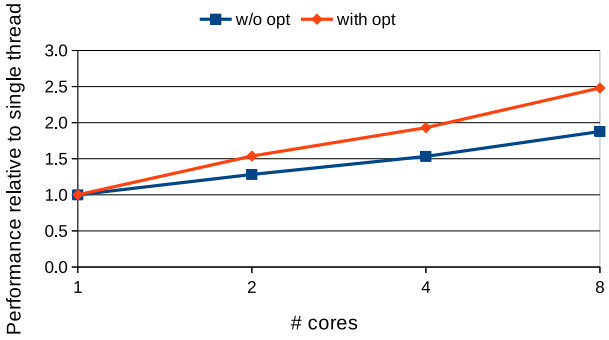
its contribution is nearly equal to or more than 10% in three programs.

4.4 Scalability

Figure 14 shows the performance relative to that for single thread execution using a single core, when the number of cores (threads) is varied. The blue and red lines represent the relative performance without and with optimizations, respectively.



(a) SPECint2000



(b) SPECfp2000

Fig. 14 Performance for various numbers of cores.

As shown in the figure, in the SPECint2000 programs, there is little scalability. This is because these programs are very hard to parallelize. In contrast, in the SPECfp2000 programs, although scalability is less impressive in the case without the optimization, good scalability is achieved in the case with the optimization. When the optimization case is compared with the no optimization case, the performance improvement caused by the optimizations is 20 to 32%.

4.5 Discussion

This section discusses the evaluation results in the previous sections.

4.5.1 Reason for Lack of Performance Improvement in *parser* without the Optimizations

As shown in Fig. 12, no performance improvement is obtained in *parser* in the case *without* the optimizations, compared with the case of a single thread. According to the simulation log, although the compiler inserted 220 `fork` instructions, no fork instructions were executed during the simulation. This can occur because the compiler relies on the profile collected by execution using the *train* inputs, while the simulation is carried out using the *ref* inputs and the simulated sections are selected by SimPoint, as described in Section 4.1.

In contrast, in the case *with* the optimizations, the compiler inserts 353 `fork` instructions, and the total number of dynamic `fork` instructions are 1,883,908. The increase in the number of static `fork` instructions arises from applying the instruction-level

partitioning optimization. This optimization increases the performance gain (see Section 2.3.2 regarding the definition of performance gain), and consequently, more partitions are selected in the thread selection process of the compiler because they have gains larger than the predetermined threshold. The additional `fork` instructions are executed during the simulation.

4.5.2 Reason for the Performance Decrease in *vortex*

As shown in Fig. 12, the performance improvement is slightly negative in *vortex*. The reason is as follows. Multithreaded execution does not necessarily increase the performance. This is due to the speculative execution of superscalar processors. Assume a single `def`→`use` relationship. If these two instructions are executed in a single thread, these can be executed speculatively. In contrast, if these two instructions are allocated to different threads, the following instruction sequence is executed: `def`→`send`→`use`. As described in Section 2.2, `send` instructions are not executed speculatively. Thus the execution of `use` can be delayed. In addition, the latency of the `send` instructions is more than a zero cycle, as shown in Table 2. These substantially increases the latency of the `def` instruction.

To avoid selecting partitions whose performance gain does not outweigh the negative effect described above, the compiler selects partitions with performance gains that are more than the predetermined threshold, as described in Section 2.3.2. However, this is only a heuristic and not a perfect solution. Performance therefor will sometimes drop when multithreaded execution is performed.

4.5.3 Reasons for the Different Effectiveness of Instruction-Level Partitioning

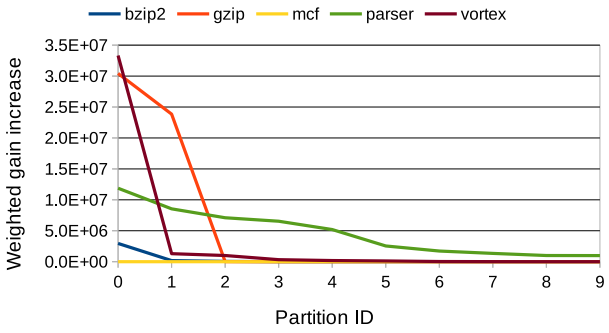
Whichever optimization is used, its effectiveness depends on whether there are opportunities for it to be applied or not in general. This is also true for instruction-level partitioning optimization. If there exist the cases we showed in Fig. 4 and the optimization improves the performance gain, a performance improvement is the result; otherwise, the optimization has no effect.

Figure 15 shows the *weighted performance gain increase* (WPGI) for each thread, which is defined as follows:

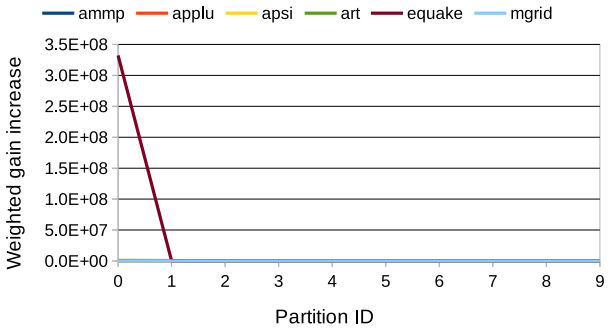
$$WPGI = (G_{IL\_part} - G_{No\_IL\_part}) \times C_{thread}, \quad (1)$$

where G_{IL_part} and $G_{No\_IL\_part}$ are the performance gain estimated by the compiler with and without the instruction-level partitioning optimization, respectively. Further, C_{thread} is the execution count of the associated thread. WPGI implies how many opportunities exist for applying the optimization, how significant the optimization is in the executed program, and how much the optimization is estimated to improve the performance using the associated new partitioning. The larger the WPGI, the more effective the optimization is, and thus the performance can be improved. The graph shows only the top ten partitions of WPGI for each program.

As shown in Fig. 15 (a), *gzip*, *parser*, and *vortex* have a positive WPGI in SPECint2000. This results in the performance improvement in *gzip* and *parser*. Note that a WPGI with a magnitude of 10^7 increases the performance by several percent, because the total dynamic instruction count we simulated is 10^9. In contrast, the performance is degraded in *vortex* slightly, because the increase in thread execution incurs the adverse effect



(a) SPECint2000



(b) SPECfp2000

Fig. 15 Weighted performance gain increase (WPGI).

of multithreaded execution related to the problem of speculative execution described in Section 4.5.2. For the same reason, the performance is degraded slightly in *bzip*.

As shown in Fig. 15 (b), in the SPECfp2000 programs, the WPGI is positive only for *equake*, and it is significant. This results in a considerable performance improvement in *equake*, as shown in Fig. 13. Because this yields a significant performance improvement, we show the details of the optimization.

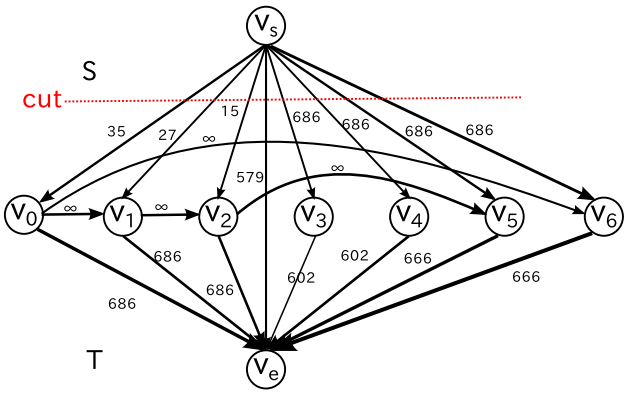
The basic block the compiler successfully optimized is only a single block. **Figure 16** (a) shows the reduced graph of this basic block, which is the input to the core algorithm of the instruction-level partitioning. Before the optimization, a `finish` instruction is placed at the top of the basic block. This means that the cut is placed immediately below v_s. As the graph shows, the performance gain of this partition is only 15 (the weight of the edge from v_s to v_2).

The instruction-level partitioning optimization moves the nodes v_2, v_1, and v_0 from T to S in this order, and then the process ends. Figure 16 (b) shows the resultant cut. As the figure shows, the performance gain is increased to 579, which is 39 times larger than the original gain.

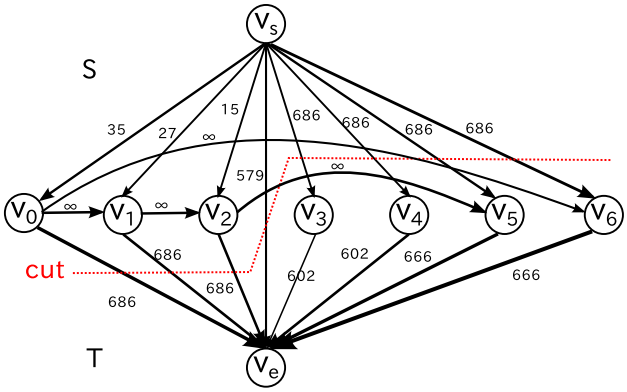
4.5.4 Computation Time of the Instruction-Level Partitioning Optimization

We measured the computation time needed to perform instruction-level partitioning optimization. The computer we used for this measurement has an Intel Xeon E5-1620 processor with 3.5 GHz clock frequency and main memory of 8 GB.

The process of instruction-level partitioning includes 1) dataflow analysis, 2) the reduction of dataflow graphs, and 3) execution of the core algorithm (see Section 3.1.2). We



(a) Before processing



(b) After processing

Fig. 16 Graph of the basic block in *equake* to be processed by the instruction-level partitioning optimization.

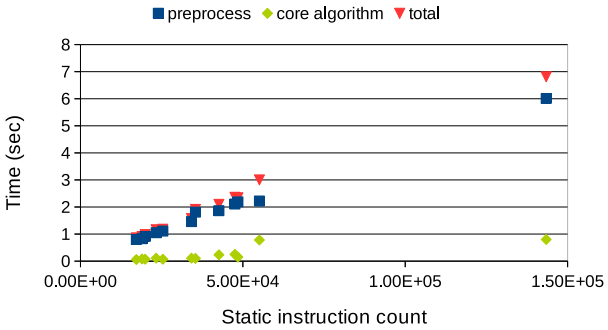
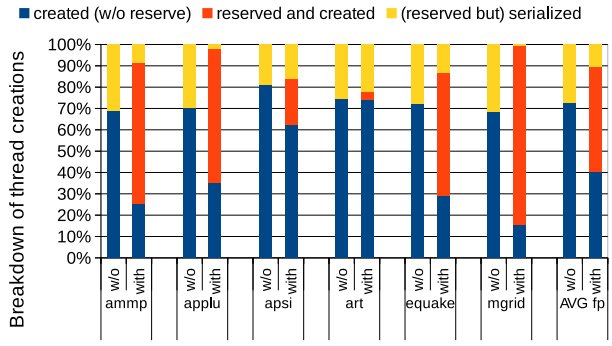


Fig. 17 Computation time for performing instruction-level partitioning.

call the processes of 1) and 2) preprocessing. The software for the preprocessing and core algorithms are written in Perl and C++, respectively. The Perl interpreter and C++ compiler we used are perl ver.5.18.2 and g++ ver.4.8.4, respectively.

Figure 17 shows the measured time for the static instruction count of each program. As shown in the figure, the time needed for the preprocessing is much longer than the time needed to execute the core algorithm. The major reason is that the software for the preprocessing is written in Perl so the interpretation is much slower than compiled binary execution. The time for both preprocessing and executing the core algorithm increases linearly with the instruction count, hence increasing the total time linearly. However, the time for any program is very short: the average time is only 2.10 sec.

(a) SPECint2000



(b) SPECfp2000

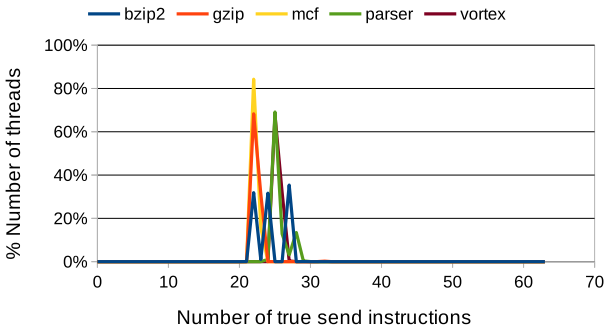
Fig. 18 Breakdown of the thread creations.

4.5.5 Discussion of the Selective Fork Optimization Evaluation Results

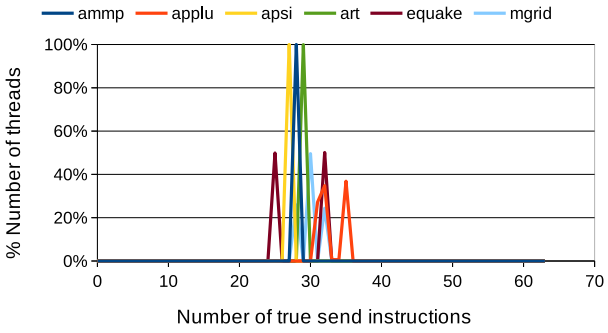
Figure 18 shows the breakdown of the number of thread creations. The two bars for each program represent the breakdown without and with the selective fork optimization, respectively. The left bar is broken down into "created" and "serialized," while the right bar is broken down into "created without reservation," "reserved and created," and "reserved but serialized". "AVG int" and "AVG fp" represent the average numbers for the SPECint2000 and SPECfp2000 programs, respectively.

As shown in the figure, selective fork optimization significantly increases the rate of successful thread creation (i.e., "created without reservation" + "reserved and created") in many programs. In general, "reserved and created" occurs quite often in the case with selective thread optimization, which contributes to the increase in successful thread creation.

In many programs, the increase in the successful thread creation rate contributes to increased performance. However, a clear positive correlation is observed only in a few programs (e.g., in *mcf* and *equake*); the correlation is unclear in many programs. For example, in *applu*, the successful thread create rate increases to nearly 100%, but the performance improvement is not very large. Conversely, in *apsi*, the successful thread create rate is only slightly increased, but the performance improvement is large. These uncorrelated relationships arise for the following reasons. First, not all threads have the same performance gain. The most important factor to increase performance is how many threads with large performance gains are not serialized, but are created. Second, the performance gain is constrained by data dependences. **Figure 19** shows the distribution of the number of

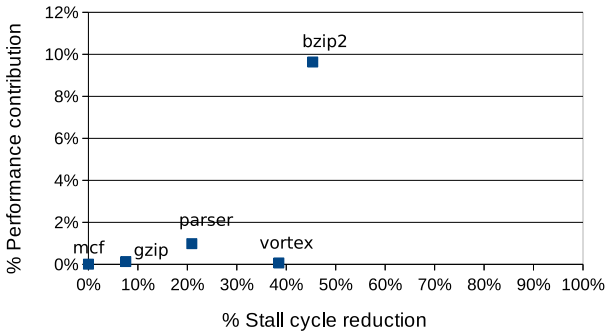


(a) SPECint2000

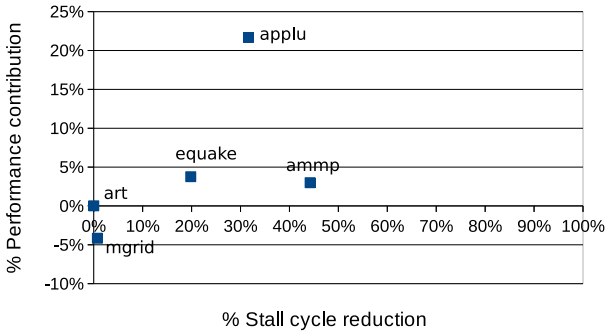


(b) SPECfp2000

Fig. 19 Distribution of the number of threads with respect to the number of true send instructions per thread.



(a) SPECint2000



(b) SPECfp2000

Fig. 20 Correlation between the performance contribution of the automatic register send and stall cycle reduction rate of the instruction window resources.

threads with respect to the number of true send instructions per thread. For example, the point for the Y-value of 80% on the X-value of 30 true send instructions indicates that the number of threads where 30 true send instructions were sent is 80% of the total number of dynamic threads. Note that true send instructions are executed because there are true data dependences between threads. As the figures show, all threads have many data dependences. These dependences constrain the amount of parallelism of threads. So the performance does not increase in proportion to an increase in thread creations.

4.5.6 Discussion of the Automatic Register Send Evaluation Results

As described in Section 3.3, the mechanism of the automatic register send becomes effective if the instruction window resources are unavailable when transfer send instructions attempt to use these resources. Thus, it is intuitively found that there are correlations between the performance contribution of the automatic register send optimization and the reduction rate of the pipeline stall cycles on the instruction window resources owing to this optimization. The correlations are shown in **Fig. 20**. Weak correlations are found in both SPECint2000 and SPECfp2000.

However, *vortex* and *ammp* do not follow this trend. We believe that the reason is the adverse effect problem associated with speculative execution described in Section 4.5.2 in *vortex*. As discussed in Section 4.5.2, the partitions decided by the compiler include those that are not beneficial in *vortex*. The automatic register send reduces the execution cycles of threads and this increases the number of thread creations. The more the number of thread creations is increased, the more the negative effect is increased. This negative effect offsets the reduced cycles gained by the automatic register send.

In contrast, we have not found a clear reason for the uncorrelation in *ammp*. In general, if there are multiple constraints that prevent parallelization, we cannot parallelize the program without *all* constraints eliminated. The sequential part of the program severely limits the performance improvement according to Amdahl's law. Although we solved three of the problems that created the constraints in this paper, there must be remaining problems that are as yet unknown. Such problems could be preventing parallelizing in *ammp*.

4.5.7 Reason for the Difference in Scalability of Integer and Floating-Point Programs

In general, multithreaded architecture and parallelizing compilers are responsible for the extraction of parallelism contained in programs. If the contained parallelism is high, they have more chances to extract it and can turn it into performance improvement; otherwise, they have fewer chances to extract it, and it is difficult to improve performance.

It is very widely known that floating-point programs (most are numerical programs) contain more parallelism than integer programs. In fact, the limit study [10] presented such results. Our results, which show that good scalability can be achieved for the floating-point programs, arises from this fact.

In contrast, integer programs are also widely known to be very hard to parallelize and thus performance improvement is difficult. This is because they have many data dependences with short distances and contain little parallelism. Again, the limit study [10] showed that integer programs contain only a small amount of parallelism. In addition, the previous studies [11], [14], [18], [24],

Table 3 Hardware cost for the automatic register send.

hardware	entries	(additional) bits per entry	cost (bytes)
TCRR	1	95	12
RSRB	64	70	560
sync table	64	1	8
total			580

[25] achieved poor performance improvement with less scalability. For example, the performance improvement is only 1.05 times in Ref. [24]. Although the performance improvement in this study is better than those in the several previous studies, our results showing the poor scalability of the integer programs corroborate the results in previous studies.

4.5.8 Hardware Cost for the Automatic Register Send

The hardware cost for the automatic register send is listed in **Table 3**. In the third column, the bits per entry are shown for the TCRR and RSRB, while the additional bit (the transfer flag) per entry for the automatic register send is shown for the sync table. Note that the number of logical registers is 64 (32 for integer and 32 for floating-point registers). As the table shows, the required cost is very small; it is only 0.9% compared with the L1 I-cache or L1 D-cache, for example.

5. Related Work

Sohi et al. first proposed a tightly coupled multicore architecture, called *multiscalar* architecture, which supports inter-core register communication using a unidirectional ring bus [21]. Our SKY architecture is motivated by this multiscalar architecture. However, the performance improvement techniques proposed in this paper were not presented in the multiscalar studies.

Marcuello et al. and Tubella et al. proposed an architecture that performs multithreaded execution by hardware alone, i.e., without compiler assistance [14], [23]. Each core is tightly coupled via a ring bus with register communication ability. The hardware dynamically detects loops and then parallelizes them. The inter-thread data dependences are relaxed by predicting the live-in values of the threads. Although this architecture has the advantage of realizing parallel thread execution without parallelization by the compiler, the hardware approach means that it is difficult to identify the benefits of the parallelization from a high-level view, unlike the compiler approach; thus, it usually parallelizes the inner-most loop and misses the potential benefits of parallel thread execution.

Renau et al. proposed the *out-of-order* thread fork [18], although the threads are conventionally forked in-order, i.e., in the same order as they would be executed sequentially. The out-of-order fork increases the overall performance even in non-numerical applications (1.30× in the SPECint2000). To support the out-of-order thread fork, the thread order must be managed, and the authors proposed a scheme to do so. While the mechanism is simplified when compared with a previous scheme, it has not achieved sufficient performance improvements to justify the hardware complications.

Zhong et al. proposed an architecture that supports various levels of grain parallelism [25]. Narrow-issue cores are tightly coupled with supporting register communication via dedicated buses. There are two modes for this architecture: coupled and decoupled. In the coupled mode, the cores work as a VLIW processor, exploiting ILP. In the decoupled mode, the cores execute fine-grain communicating threads that are extracted by their compiler. Although ILP can be exploited to collect the cores with low-latency register communication, the latency of the register communication has a negative impact on the exploitation of ILP. Wide-issue superscalar processors can exploit ILP more effectively, and this architecture is thus less useful for performance enhancement in programs with less TLP.

Luo et al. proposed a scheme that dynamically determines the level of a loop that is beneficial for parallelization [12]. According to the authors, profiling to find the benefits of parallelization, which is adopted in most of their other work (e.g., Ref. [11]), is highly dependent on the input data. Also, changing the phases of execution alters the beneficial levels of the loops. By combining a compiler hint with their proposed dynamic scheme, they achieved 9.5% performance improvement when compared with a static scheme in the SPEC2000 programs.

Campanoni et al. proposed a low-latency inter-core commutation architecture, similar to that of the multiscalar architecture, with a parallelizing compiler that exploits its feature [4]. The evaluation results using SPEC2000 programs show that the performance is very sensitive to the latency of the inter-core communication, and they claim that the low-latency communication is thus very important.

Most studies on parallelization of single-thread applications are carried out by a single partitioning level (i.e., either loop, function, or basic block levels); they do not evaluate performance by varying the partitioning level. This is because changing levels requires exhaustive changes of the compiler, and comparison is thus very difficult. However, the limit of parallelism depending on partitioning levels has been previously assessed. Marcuello et al. assessed loop and function levels, and concluded that loop-level partitioning yields higher parallelism, and function-level partitioning hardly yields any parallelism [13]. Nakajima et al. assessed basic block level in addition to the levels Marcuello et al. assessed [15], [16]. Their study results corroborate poor parallelism in function-level partitioning as Marcuello et al. found, but Nakajima et al. concluded that loop-level partitioning is also not attractive. In contrast, they showed that basic-block-level partitioning exhibited significant amounts of parallelism, as previously indicated by Lam et al. [10]. Our partitioning policy of the basic block level is based on these studies.

6. Conclusion

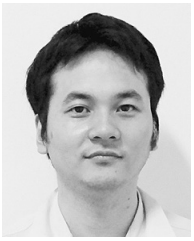
Single-thread performance has not seen dramatic improvements for more than a decade. Tightly coupled multicore architectures provide a potential solution for this single-thread performance barrier, because they enable very low-latency inter-thread communication and very lightweight thread creation. These features are advantageous in fine-grain parallel thread execution of hard-to-parallelize programs with many inter-thread data dependences. SKY is a typical tightly coupled multicore architecture. Only slight modification is necessary to migrate from conventional architecture to SKY.

In this paper, we proposed three software and hardware techniques to improve the performance of SKY. Our evaluation results using SPEC2000 benchmark suite demonstrate that the proposed techniques achieved mean performance improvements of 4% and 26% (maximum of 11% and 206%) over the base performance of SKY for a four-core processor executing integer and floating-point programs, respectively. The resulting mean performance improvements over a single core were as much as 1.21 times and 1.93 times, respectively (maximum of 1.52 times and 3.30 times, respectively).

Acknowledgments This work is supported by the Ministry of Education, Culture, Sports, Science and Technology Grant-in-Aid for Scientific Research (C) (No. 16K00070). This work is also supported by VLSI Design and Education Center (VDEC), the University of Tokyo with the collaboration with Synopsys Inc.

References

[1] PTM, available from ⟨http://ptm.asu.edu/⟩.
[2] Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts (1986).
[3] Breach, S.E., Vijaykumar, T.N. and Sohi, G.S.: The Anatomy of the Register File in a Multiscalar Processor, *Proc. 27th International Symposium on Microarchitecture*, pp.181–190 (1994).
[4] Campanoni, S., Brownell, K., Kanev, S., Jones, T.M., Wei, G.-Y. and Brooks, D.: HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs, *Proc. 41st Annual International Symposium on Computer Architecuture*, pp.217–228 (2014).
[5] Hamerly, G., Perelman, E., Lau, J. and Calder, B.: SimPoint 3.0: Faster and more flexible program analysis, *Journal of Instruction-Level Parallelism*, Vol.7, pp.1–28 (2005).
[6] Hou, C.: A Smart Design Paradigm for Smart Chips, *2017 IEEE International Solid-State Circuits Conference, Digest of Technical Papers, Plenary Session* (2017).
[7] Intel: *P6 Family of Processors - Hardware Developer's Manual* (1998).
[8] International Technology Roadmap for Semiconductors, available from ⟨http://www.itrs2.net/⟩.
[9] 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).
[10] Lam, M.S. and Wilson, R.P.: Limits of control flow on parallelism, *Proc. 19th Annual International Symposium on Computer Architecture*, pp.46–57 (1992).
[11] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J. and Torrellas, J.: POSH: A TLS compiler that exploits program structure, *Proc. Eleventh Symposium on Principles and Practice of Parallel Programming*, pp.158–167 (2006).
[12] Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N. and Tarkas, A.: Dynamic performance tuning for speculative threads, *Proc. 36th Annual International Symposium on Computer Architecture*, pp.462–473 (2009).
[13] Marcuello, P. and González, A.: A Quantitative Assessment of Thread-Level Speculation Techniques, *Proc. 14th International Symposium on Parallel and Distributed Processing*, pp.595–601 (2000).
[14] Marcuello, P., González, A. and Tubella, J.: Speculative multithreaded processors, *Proc. 12th International Conference on Supercomputing*, pp.77–84 (1998).
[15] Nakajima, A., Kobayashi, R., Ando, H. and Shimada, T.: Limit of Thread-Level Parallelism on Partitioning Levels and Speculations in Non-Numerical Programs, *Proc. 8th Symposium on Low-Power and High-Speed Chips*, pp.465–472 (2005).
[16] Nakajima, A., Kobayashi, R., Ando, H. and Shimada, T.: Limits of Thread-Level Parallelism in Non-numerical Programs, *IPSJ Trans. Advanced Computing Systems*, Vol.47, No.SIG 7 (ACS 14), pp.12–20 (2006).
[17] Palacharla, S., Jouppi, N.P. and Smith, J.E.: Quantifying the complexity of superscalar processors, Technical Report CS-TR-1996-1328, University of Wisconsin-Madison (1996).
[18] Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K. and Torrellas, J.: Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation, *Proc. 19th Annual International Conference on Supercomputing*, pp.179–188 (2005).
[19] SimpleScalar LLC., available from ⟨http://www.simplescalar.com/⟩.
[20] Smith, M.D., Horowitz, M.A. and Lam, M.S.: Efficient Superscalar Performance Through Boosting, *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.248–259 (1992).
[21] Sohi, G.S., Breach, S.E. and Vijaykumar, T.N.: Multiscalar Processors, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.414–425 (1995).
[22] SPEC, available from ⟨https://www.spec.org/cpu2000/docs/runspec.html⟩.
[23] Tubella, J. and González, A.: Control Speculation in Multithreaded Processors Through Dynamic Loop Detection, *Proc. 4th International Symposium on High-Performance Computer Architecture*, pp.14–23 (1998).
[24] Zhai, A., Colohan, C.B., Steffan, J.G. and Mowry, T.C.: Compiler optimization of scalar value communication between speculative threads, *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.171–183 (2002).
[25] Zhong, H., Lieberman, S.A. and Mahlke, S.A.: Extending multicore architectures to exploit hybrid parallelism in single-thread applications, *Proc. 13th International Symposium on High Performance Computer Architecture*, pp.25–36 (2007).



Keita Doi received his B.E. and M.E. degrees from Nagoya University, Nagoya, Japan, in 2012 and 2014, respectively. Since then, he has been with Okuma Corporation.

Ryota Shioya was born in 1981. He received his M.E. and Ph.D. degrees in Information and Communication Engineering from the University of Tokyo in 2008 and 2011, respectively. He was a research fellow of the Japan Society for the Promotion of Science from 2009. From 2011, he was an assistant professor at the Graduate School of Engineering, Nagoya University. Since 2016, he has been an associate professor at the Graduate School of Engineering, Nagoya University. He is a member of IPSJ, IEICE and IEEE.

Hideki Ando received his B.S. and M.S. degrees in Electronic Engineering from Osaka University, Suita, Japan, in 1981 and 1983, respectively. He received his Ph.D. degree in Information Science from Kyoto University, Kyoto, Japan, in 1996. From 1983 to 1997 he was with Mitsubishi Electric Corporation, Itami, Japan. From 1991 to 1992 he was a visiting scholar at Stanford University. In 1997 he joined the faculty of Nagoya University, Nagoya, Japan, where he is currently a Professor in the Department of Information and Communication Engineering. He received IPSJ Best Paper Awards in 1998 and 2002, and a Best Paper Award at the Symposium on Advanced Computing Systems and Infrastructures in 2013. His research interests include computer architecture and compilers.