Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

Scalar Replacement Considering Branch Divergence

Junji Fukuhara[1,a)] Munehiro Takimoto[1,b)]

Received: June 18, 2021, Accepted: September 9, 2021

Abstract: GPU with the Single Instruction Multiple Data (SIMD) execution model enables a program to work efficiently. However, the efficiency may decrease because of branch divergence that occurs when SIMD threads follow different paths in some branches. Once the divergence occurs, some threads must wait until completion of the execution of the others. Thus, it is important to reduce branch divergence to improve the efficiency of GPU programs. On the other hand, branch divergence may be increased by some traditional code optimizations based on code motion such as partial redundancy elimination (PRE) and scalar replacement (SR). These methods insert some expressions into some paths, on which insertion points may be included in divergent branches. That is, the insertion may increase branch divergence, which may result in the decrease of execution efficiency of GPU programs. In this paper, we propose a new SR approach, called Speculative SR based on Question Propagation (SSRQP), which not only removes redundant memory accesses but also reduces branch divergence. SSRQP achieves SR based on speculative code motion, which not only eliminates inter-iteration redundant memory accesses without increasing branch divergence but also decreases branch divergence that originally exists through hoisting memory accesses in true and false sides of a divergent branch out of it. To prove the effectiveness of our method, we have conducted experiments through applying it to some benchmarks with divergent branches. The experimental results show that it can improve the efficiency about 40% in the best case in comparison with traditional techniques.

Keywords: GPU, branch divergence, code optimization, scalar replacement, speculative code motion

1. Introduction

GPU plays an increasingly important role in general purpose applications. Most GPU exploits *Single Instruction Multiple Data* (SIMD) execution model, which enables programs to be executed efficiently. However, the efficiency may decrease when SIMD threads in a warp follow different paths of execution, which is called *branch divergence*. In face of the divergence, some threads have to wait the completion of the execution of the others, so that it decreases execution efficiency of GPU programs. In addition, some traditional code optimizations based on code motion may *increase* branch divergence, because they may insert some expressions into the destinations of divergent branches. *Partial Redundancy Elimination* (PRE) [3], [4], [5], which is effective code optimization technique that not only removes partially redundant expressions but also moves invariant expressions out of loops, is one of such code motion approaches. PRE may decrease the execution efficiency of GPU programs, so that it is difficult to apply PRE to them [21]. *Scalar Replacement* (SR) [11], [12], [13], [14], which removes inter-iteration redundant memory accesses in a loop, is also one of such code motion approaches. SR may increase branch divergence because it inserts memory accesses into some paths as well as PRE. The property of SR, which can actually be achieved as an extension of PRE, makes its application to GPU programs difficult.

In this paper, we propose a new approach, which is called *Speculative Scalar Replacement based on Question Propagation* (SSRQP). The approach not only removes redundant memory accesses in a loop but also reduces branch divergence through speculative code motion. The redundancy checking is based on *Question Propagation* [15], which checks whether each expression is redundant while propagating questions on a control flow graph. Furthermore, the speculative code motion takes advantage of a property where GPU programs execute both true and false sides of divergent branches. The property enables SSRQP to speculatively hoist an expression in one side of a divergent branch out of it without decreasing execution efficiency. The speculative code motion hoists all of the possible expressions in both sides of a divergent branch out of it, which contributes to reducing branch divergence. In addition, the speculative code motion makes more expressions redundant [7], [8], [9], [10], so that SSRQP eliminates more expressions than traditional methods, which contributes to further improvement of execution efficiency. On the other hand, the speculative code motion for non-divergent branches may decrease execution efficiency because it may introduce new expressions on some paths [7], [8]. To avoid decreasing performance, SSRQP applies the speculative code motion to only divergent branches. Notice here that SSRQP does not insert any expression into destinations of divergent branches, so that it does not increase branch divergence.

The contributions of this paper are as follows:

(1) The SSRQP removes redundant memory accesses in GPU programs without increasing branch divergence.
(2) The SSRQP speculatively hoists an expression out of a divergent branch without decreasing execution efficiency.

[1] Department of Information Sciences, Tokyo University of Science, Noda, Chiba 278–8510, Japan
[a)] 6320703@ed.tus.ac.jp
[b)] mune@rs.tus.ac.jp

(3) The SSRQP contributes to reducing branch divergence through speculative code motion.
(4) The SSRQP eliminates more expressions through speculative code motion.

The rest of this paper is organized as follows: Section 2 presents the preliminaries of our approach. Section 3 describes branch divergence. Section 4 provides brief explanations of a method based on question propagation. Section 5 gives our proposed method. Section 6 presents experimental results of our methods. Section 7 discusses related works. Finally, we conclude our paper and show future works in Section 8.

2. Preliminaries

2.1 Basics

We assume that a *Control Flow Graph* (CFG) has already been generated for each function defined in the source program. The CFG is a directed graph $G(N, E, \mathbf{s}, \mathbf{e})$ with a node set N and an edge set $E \subset N \times N$. Each node $n \in N$ represents a *basic block* consisting of continuous statements without any branch in the middle. Each edge $(n, m) \in E$ represents the flow of control between basic blocks n and m. $\mathbf{s}$ and $\mathbf{e}$ denote the unique *start node* and *end node* of G. Every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$. $pred(n) =_{df} \{m \mid (m, n) \in E\}$ and $succ(n) =_{df} \{m \mid (n, m) \in E\}$ denote sets of all the predecessors and successors of a node n, respectively.

A node m *dominates* a node n if and only if every path from $\mathbf{s}$ to n contains m. Also, a node m *postdominates* a node n if and only if every path from n to $\mathbf{e}$ contains m. A node m is called the *immediate dominator* of a node n if and only if m dominates n, m is not equal n, and the node except m which dominates n does not exist on every path from m to n. Inversely, a node m is called the *immediate post-dominator* of a node n if and only if m postdominates n, m is not equal n, and the node except m which postdominates n does not exist on every path from n to m [1]. A node m is *control-dependent* on a node n if and only if there is a non-empty path from n to m such that m postdominates all the nodes except n on the path [2]. We compute a control dependency for an *augmented CFG*, which is the CFG augmented with a special node ENTRY that has one edge going to the start node $\mathbf{s}$ and another edge going to the end node $\mathbf{e}$ [26]. We assume that the node ENTRY is non-divergent.

As well as the other code motion methods, *critical edges* [4], which lead from nodes with more than one successor to nodes with more than one predecessor, may block an effective code motion. We assume that critical edges are eliminated through inserting a new node on the edges.

A *dependence* exists between two memory references if there exists some paths of CFG from the first reference to the second reference and both references access the same memory location [12]. The dependence is called *loop-carried dependence* if two dependent references are in different iterations of a loop. If two dependent references are in the same iteration of a loop, the dependence is called *loop-independent dependence*. The *threshold* of a loop-carried dependence is the number of loop iterations between two dependent references. If the threshold is constant throughout the execution of the loop, the threshold is called *consistent threshold*.

2.2 CUDA

The *Compute Unified Device Architecture* (CUDA) [30] provides a C-extended programming model for GPU. The programmer writes the host code processed on the CPU side and the device code processed on the GPU side separately in the program. The device code is also called a *kernel*. The programmer launches a kernel with hierarchical execution configuration, called *grid*. A grid consists of multiple *blocks*. Each block contains multiple *threads*. A group of 32 threads is called a *warp*. A kernel can access multiple GPU memories during execution, including three kinds of off-chip memories and one kind of on-chip memory. Off-chip memory includes *global*, *constant*, and *texture memory* that are shared by all GPU threads and the CPU. On-chip memory includes *shared memory* that is shared among threads within a block on the GPU.

2.3 Partial Redundancy Elimination

An expression e is *available* at node n if each path from the start node $\mathbf{s}$ to n includes node m that has e and any operands of e are not modified on the path between m and n. Also, e is *partially available* at node n if e is available at some nodes of the predecessors of n. If e is available at node n, n is *up-safe* for e. If an expression e exists at node n and is available immediately before n, e is *totally redundant* at n, and can be eliminated by replacing it with the variable that holds the value of e. On the other hand, if e exists at node n and is partially available immediately before n, e is *partially redundant* at n, and cannot simply be removed as totally redundant expressions.

An expression e is *anticipated* at node n if each path from n to the end node $\mathbf{e}$ includes node m that has e and any operands of e are not modified on the path between n and m. Moreover, e is *partially anticipated* at node n if e is anticipated at some nodes of the successors of n. If e is anticipated at node n, n is *down-safe* for e.

The PRE removes partially redundant expressions through inserting expressions into the appropriate program points. At this time, since PRE inserts expressions at the down-safe nodes, it can remove partially redundant expressions without increasing the number of expressions on any execution path. The code motion that inserts new expressions at non down-safe nodes is called *speculative code motion*. The motion may increase the number of executed statements, so that it may decrease execution efficiency of a program.

3. Branch Divergence

Branch divergence is known as a problem that causes significant performance bottlenecks for GPU programs with SIMD execution model. In the model, GPU programs must execute the statements in both true and false sides of a divergent branch because each warp has just a single control flow.

In **Fig. 1** (a), we assume that the conditional branches at nodes 1 and 2 cause branch divergence. As illustrated in Fig. 1 (c), we assume that a warp has eight threads, which are represented by squares. The black squares represent executing

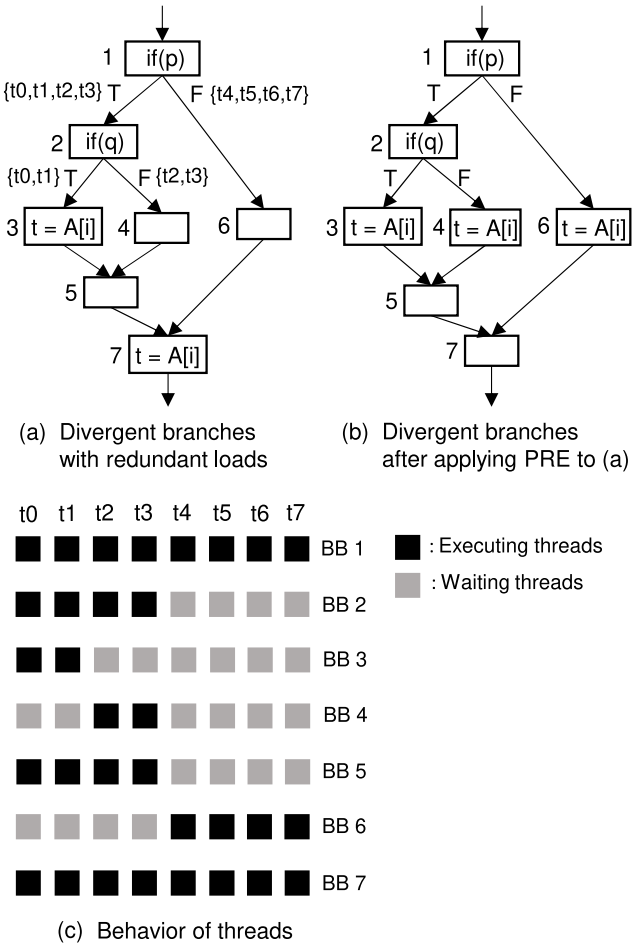

Fig. 1 Divergent branches and behavior of threads.

threads, and the gray ones represent waiting threads. First, all the threads execute statements at node 1 in parallel. Second, the threads from `t0` through `t3` execute statements at node 2 in the true side of node 1. At this time, the threads from `t4` through `t7` wait without doing anything. Moreover, `t0` and `t1` execute statements at node 3 in the true side of node 2. At this time, `t2` and `t3` wait until the completion of the execution at node 3. Then, `t2` and `t3` execute statements at node 4 in the false side of node 2. At this time, `t0` and `t1` wait. After the execution at node 4, the threads from `t0` through `t3` execute statements at node 5. After that, the threads from `t4` through `t7` execute statements at node 6 in the false side of node 1. At this time, the threads from `t0` through `t3` wait. Finally, all the threads execute statements at node 7.

Volta GV100 or later GPUs of NVIDIA support *Independent Thread Scheduling* (ITS) [33], which enables a program to be executed based on finer-grain parallel algorithms where threads in the same warp may synchronize and cooperate. ITS interleaves execution of statements in divergent branches, so that it can reduce some overhead of branch divergence in a program. However, branch divergence still decreases the execution efficiency of the program because the execution of Volta and later GPUs is also in SIMD fashion, which causes the branch divergence.

As discussed above, branch divergence takes both costs of the true and false sides of a divergent branch to execute it, so that it decreases execution efficiency of GPU programs. Several methods have been proposed to reduce branch divergence and improve execution efficiency [19], [20], [21], [22], [23], [24], [25]. On the other hand, some traditional optimizations may increase branch divergence and decrease execution efficiency. Since code motion-based approaches such as PRE or scalar replacement are included in them, they cannot simply be applied to GPU programs with branch divergence.

As shown in Fig. 1 (a), expressions originally exist in one side of the divergent branch node 2 and in node 7, and the expression in node 7 is partially redundant. The traditional PRE transforms Fig. 1 (a) into Fig. 1 (b). In Fig. 1 (b), the redundancy is removed by PRE. However, the expressions appear in both sides of the divergent branches at nodes 1 and 2. Therefore, execution efficiency is reduced compared with Fig. 1 (a) because of the divergence at nodes 1 and 2.

4. Question Propagation

In this section, we first give an outline of question propagation. Then, we describe our extension of SR based on question propagation.

4.1 Outline

Question propagation [15] is a method that checks whether each expression is redundant or not while propagating questions on CFG. As methods related to question propagation, *partial redundancy elimination based on question propagation* (PREQP) [16], [18], and *scalar replacement based on question propagation* (SRQP) [17] have been proposed. First, these methods visit each CFG node in topological sort order. Then, they check whether each expression e can be eliminated through propagating a query about availability backwardly in order to determine whether the lexically same expressions as e exist at the node where the query is propagated. We call this expression e a *questionary expression*. If a query is propagated to a node that has the same expression as a questionary expression, the node returns *true* as its answer which is forwardly propagated to the originating node of the query. Conversely, if a query is propagated to a node that has the statement that modifies the operands of a questionary expression, the node returns *false*. If both *true* and *false* are returned to a node n from the predecessors, another question propagation checks whether n is down-safe through propagating a query about anticipatability forwardly. If the query is propagated to a node that has the same expression as the questionary expression, the node returns *true*. Once n is found to be down-safe, the questionary expression is inserted into the node that returns *false* as its answer about availability to make the questionary expression totally redundant at n, so that n returns *true* as its answer about availability toward the originating node of the propagation. Consequently, e is decided to be redundant if the originating node gets *true* as its answer about availability.

We illustrate the question propagation in PREQP about the expression $a + b$ at node 4 in **Fig. 2** (a). First, it propagates queries about availability to check whether $a + b$ is available to the predecessors of node 4. Second, the query propagated to node 3 returns *true* as its answer because $a + b$ is available in node 3, as shown in Fig. 2 (b). On the other hand, the query propagated to node 2 does not solve its answer because there are no statements

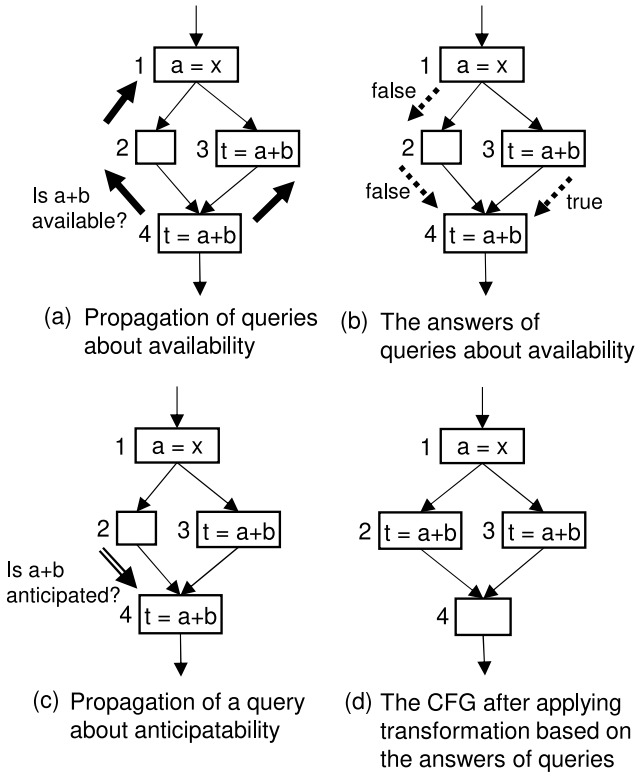

Fig. 2 An example of PREQP.

in node 2. In this case, the query is propagated further to the predecessor of node 2. Then, the query propagated to node 1 returns *false* because the operand of $a + b$ is modified by the statement $a = x$ at the node. As a result, both *true* and *false* are obtained at node 4, which denotes the questionary expression is partially redundant. To remove the redundancy, as illustrated in Fig. 2 (c), we propagate a query about anticipatability in order to check whether node 4 is down-safe. Consequently, we obtain *true* as the answer, so that we find out node 4 is down-safe. Finally, we remove $a + b$ at node 4 through inserting the same expression to node 2, which results in the CFG shown in Fig. 2 (d).

4.2 Scalar Replacement Based on Question Propagation

Scalar replacement based on question propagation (SRQP) is an extension of PREQP to scalar replacement. SRQP checks whether each array reference expression e at each CFG node is redundant through question propagation over iterations.

In SRQP, the answer of a query about availability is a tuple of *isAvail* and *isReal*, where *isAvail* represents whether a questionary expression is available and *isReal* represents the fact of reaching some occurrences of the same expression as a questionary expression. The expression that originates question propagation is an array reference expression such as $A[i]$, of which the operands of the questionary expression include A and i. In the propagation process, the answer of a query about availability at node n is determined as follows:

(1) If n is the start node **s**, the answer is (*false*, *false*).
(2) If n is the node where the query has already been propagated twice, the answer is (*false*, *false*).
(3) If n is the node where the same query has already been propagated, the answer is (*true*, *false*).
(4) If n contains a statement that may modify the operands of a questionary expression or a memory location accessed through it, the answer is (*false*, *false*).
(5) If n has the same expression as a questionary expression, the answer is (*true*, *true*).
(6) If n has definitions of the operands of a questionary expression and is visited for the first time, the operands are replaced with the right-hand side of the definition statement for algebraic conversion.
(7) If any above rules are not applied, the query is propagated to all of the predecessors of n.

We apply the above rules in order from the top to the bottom. In the process of question propagation, an answer can be solved locally in each node if any of rules (1) to (6) is satisfied. If an answer cannot be solved locally, SRQP propagates a query according to the rule (7) and applies the rules to the predecessors.

As shown in the rule (2), we allow different queries to visit a node at most twice to analyze loop-carried availability and remove inter-iteration redundant expressions. As shown in the rule (3), the answer of *isAvail* is *true* if the same query has already been propagated to n. This is because the result of question propagation corresponds to the maximum fixed point of the dataflow equations. Moreover, as shown in the rule (4), a questionary expression must not be propagated beyond a modification statement of the questionary expression. The modification includes a store statement to the same memory location as the array reference. In general, it is difficult to decide whether two memory references access the same memory location because one may be an alias of the other. In order to expose such alias relation, pointer analysis or alias analysis may be required. If a store statement may modify the memory location referenced, SRQP conservatively considers that the referenced memory location is modified. As shown in the rule (6), the operand i of a questionary expression is replaced with the right-hand side of a definition statement such as $i = i + 1$. That is, the operand i is replaced with $i + 1$. By means of this rule, SRQP can analyze loop-carried availability.

If all of the queries propagated to predecessors of n have *true* as their answers of *isAvail*, the answer of *isAvail* at n is also *true*. If the answers of *isAvail* from predecessors of n have both *true* and *false*, a questionary expression at n is partially available. At this time, SRQP inserts a questionary expression into the predecessors where the answers are *false* if the expression is anticipated at n, so that the answer at n becomes *true*. Notice here that if predecessors that have *true* as their answers of *isAvail* are caused by repropagating a query to the same point without reaching an occurrence of the same expression as a questionary expression, unnecessary insertions at predecessors with *false* result in such as ineffective code motion through empty loops (called *hoisting-through-the-loop effect* [6]), as illustrated in **Fig. 3** (b). To avoid the unnecessary insertions, SRQP defines a predicate *isReal*, which represents the fact of reaching some occurrences of the same expression as a questionary expression. SRQP inserts new expressions

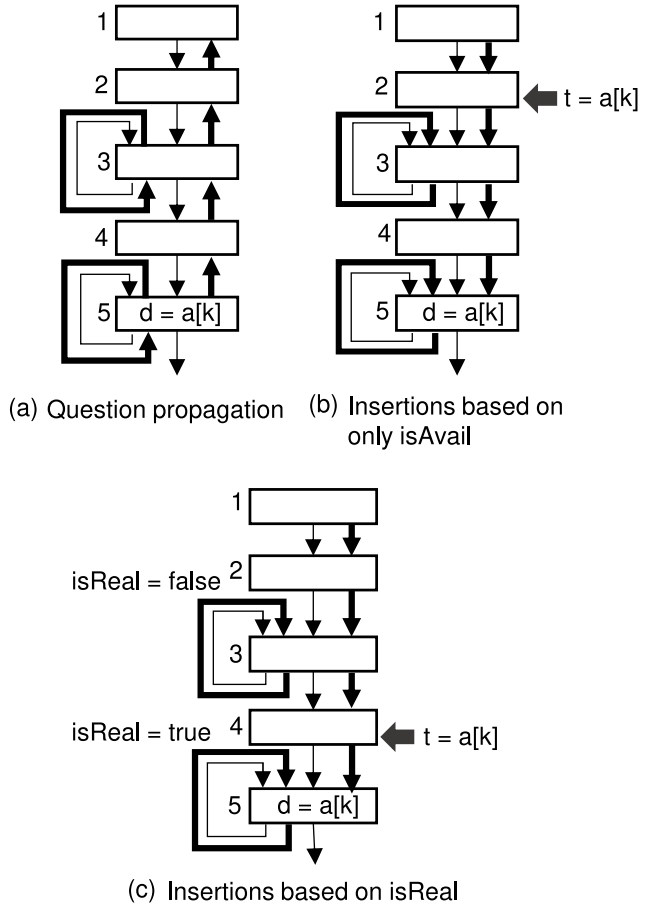

Fig. 3 An example of unnecessary code motion.

only if some of predecessors with *true* as their answers of *isAvail* have also *true* as *isReal*. As shown in Fig. 3 (c), the propagation to node 3 does not satisfy this requirement, whereas the one to node 5 satisfies *isReal*. As a result, SRQP inserts a new expression into node 4. Note that the result is same as that of *Lazy Code Motion* (LCM) [4], [5], and the predicate *isReal* works the same way as the predicate *Latest* used in LCM.

SRQP checks anticipatability through propagating another query as well as availability. The query about anticipatability is forwardly propagated contrary with the one about availability. In addition, a questionary expression of a query about anticipatability contains all the operands that have been algebraically converted during the propagation of a query about availability. That is, it contains i and $i + 1$ if the operand i have been converted to $i + 1$, which denotes that SRQP checks the anticipatability of $A[i]$ and $A[i + 1]$ at the same time. The answer of a query about anticipatability at node n is decided as follows:

(1) If n is the end node **e**, the answer is *false*.
(2) If n is the node where the same query has already been propagated, the answer is *true*.
(3) If n contains a statement that may modify the operands of a questionary expression or a memory location accessed through it, the answer is *false*.
(4) If n has the same expression as a questionary expression, the answer is *true*.
(5) If any above rules are not applied, the query is propagated to all of the successors of n.

As described above, SRQP checks availability and anticipatability for each array reference expression e at n. Let ($isAvail_p$, $isReal_p$) be the value returned as the result of propagation from the predecessors of n, and let *isDownSafe* be the result of propagation of a query about anticipatability at n. The condition for the availability of e at n is as follows:

$$\prod_{p \in pred(n)} isAvail_p \vee isDownSafe \wedge \bigcup_{p \in pred(n)} isReal_p$$

If the above condition is *true*, n returns *true* toward the originating node of the propagation.

Consider the question propagation for the expression $A[i]$ in node 8 in **Fig. 4** (a), where an array reference $A[i]$ denotes the memory access to the i-th element of the array A. First, SRQP propagates a query about availability to the predecessors of node 8 to check the availability of $A[i]$. When the query visits node 2, it is propagated further to nodes 1 and 9. The query propagated to node 9 is propagated further to node 8. At this time, SRQP performs algebraic conversion of the questionary expression because node 8 has the statement $i = i + 1$. Hence, SRQP converts the questionary expression $A[i]$ to $A[i + 1]$. Then, the query about $A[i + 1]$ is propagated to the predecessors of node 8 as shown in Fig. 4 (b). The query propagated to node 7 is propagated further to the predecessors of node 7. The one about $A[i+1]$ propagated to node 5 gives *true* as its answer since the expression $A[i + 1]$ exists in the node. On the other hand, the one propagated to node 6 is propagated further to the predecessors of node 6, resulting in *false* as its answer according to the rules (2) and (4). Consequently, at node 7, both *true* and *false* as answers of queries about availability are obtained as illustrated in Fig. 4 (c), so that SRQP forwardly propagates a query about anticipatability to check whether node 7 is down-safe. In the process of the propagation, it checks whether the expression $A[i]$ or $A[i + 1]$ is anticipated. As a result, the query about anticipatability gives *true* as the answer of node 8, which denotes that node 7 is down-safe. Thus, node 7 returns *true* as its answer of the query about availability to node 8, and node 6, which returns *false* as its answer of a query about availability, is marked as the destination of insertion of the questionary expression $A[i + 1]$. Remember that the query for $A[i + 1]$ is propagated from node 8 to node 3. It is propagated further to the predecessors of node 3, and it gives *false* as its answer as the case of node 6. Consequently, as shown in Fig. 4 (d), both *true* and *false* as answers of queries about availability are obtained at node 8. Similar to the case in node 7, SRQP checks whether node 8 is down-safe through propagating a query about anticipatability. The query results in *true* for down-safety at node 8, so that node 8 returns *true* as its answer of the query about availability to node 9, and node 3 is marked as an insertion point of the questionary expression $A[i + 1]$. As shown in Fig. 4 (e), node 2 obtains *true* as the answer of node 9 and *false* as the answer of node 1. Similar to the case in nodes 7 and 8, SRQP checks whether node 2 is down-safe through propagating a query about anticipatability. The query results in *true* for down-safety at node 2, so that node 2 returns *true* as its answer of the query about availability, and node 1 is marked as an insertion point of the questionary expression $A[i]$. As a result, node 8 that

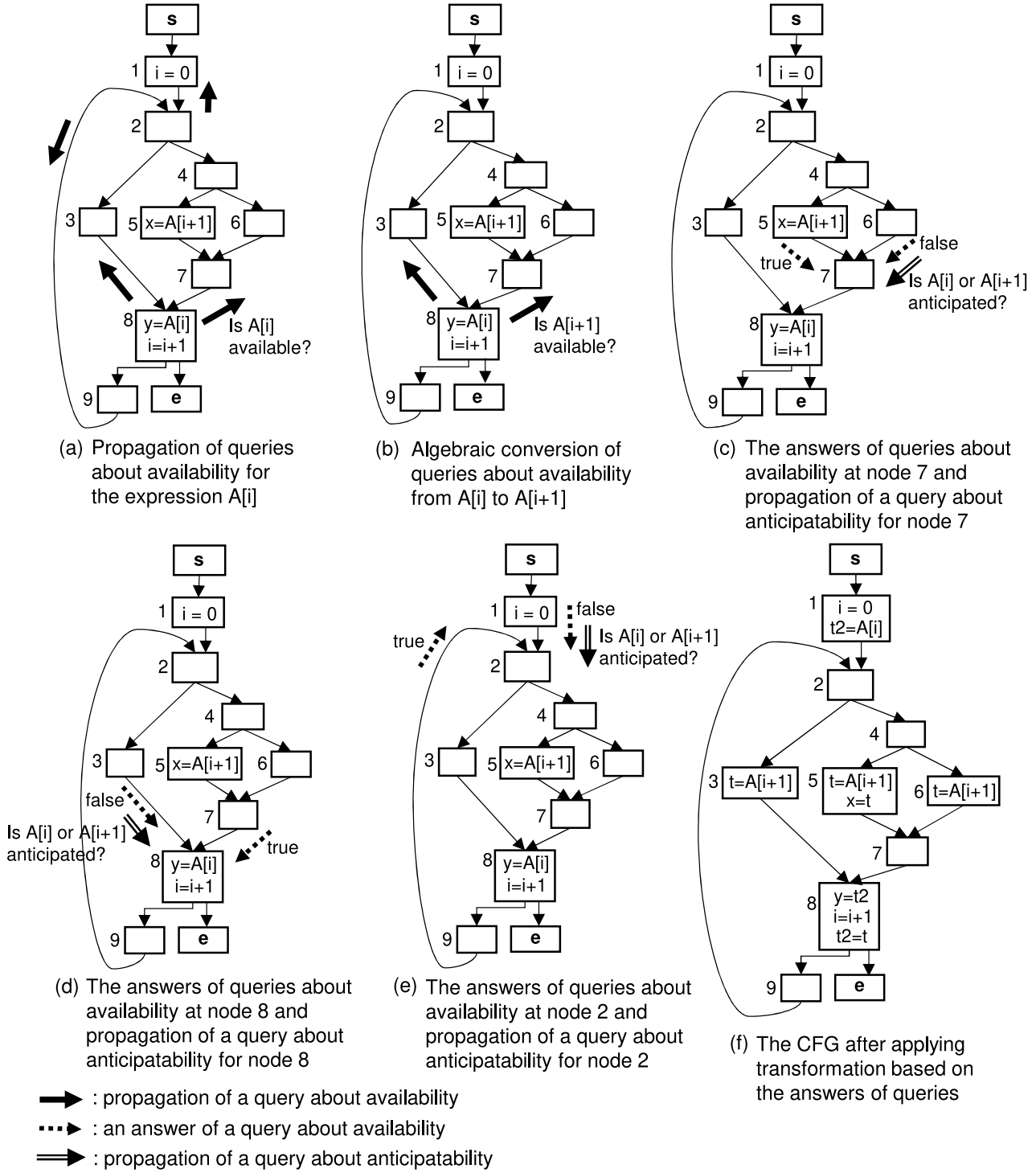

Fig. 4 Propagation of queries about availability and anticipatability in SRQP.

is the originating node of a query obtains *true* as the answer of the query about availability. Based on the result of the question propagation, SRQP transforms the CFG in Fig. 4 (a) to the one in Fig. 4 (f) through inserting $t2 = A[i]$ into node 1 and $t = A[i + 1]$ into nodes 3 and 6, which are the nodes marked during the propagation.

In the CFG illustrated in Fig. 4 (f), the redundancy of an array reference between $A[i]$ and $A[i + 1]$ is removed. However, if the branch nodes 2 and 4 are divergent, branch divergence is increased compared to the CFG in Fig. 4 (a). This is because of the insertion into nodes 3 and 6, which are the destinations of divergent branches at nodes 2 and 4.

5. Extension of SRQP

In this section, we extend SRQP in the following points:

(1) insertion of expressions considering branch divergence, and
(2) propagation of a query about speculation and code motion based on the answer.

5.1 Insertion of Expressions Considering Branch Divergence

As mentioned in the previous section, the application of SRQP

may increase branch divergence. We suppress the increase through insertion of expressions based on a control dependency property. Here, we define a *control dependence region of non-divergence* (CDRND) as follows:

definition. *A control dependence region of non-divergence is the set of nodes that are control-dependent on only non-divergent branches.*

A CDRND denotes a region where branch divergence is not increased by inserting expressions to make a partially redundant expression totally redundant. When inserting an expression to a node, we check whether the node is included in CDRND. If a node where an expression is inserted is not in CDRND, we suppress the insertion in order not to increase branch divergence.

Consider a CDRND of the CFG in Fig. 4, where we assume that the branch nodes 2 and 4 are divergent while the node 8 is non-divergent. In addition, we compute a control dependency for the augmented CFG [26], of which the special node ENTRY to be non-divergent. nodes 3, 4, and 7 are control-dependent on node 2, and nodes 5 and 6 are control-dependent on node 4. On the other hand, nodes 2, 8, and 9 are control-dependent on node 8, and nodes **s**, 1, 2, and 8 are control-dependent on the special node ENTRY. Thus, the CDRND of the CFG consists of nodes **s**, 1, 2, 8, and 9. Therefore, we suppress the insertion of expressions into nodes 3 and 6 in Fig. 4 (f).

We extend SRQP to *Extended SRQP* (ESRQP) through suppression of insertion based on a CDRND. As mentioned above, if a node obtains *true* and *false* as the answers of queries about availability from the predecessors, SRQP inserts a questionary expression into the node that returns *false*. On the other hand, ESRQP checks whether the node that returns *false* is included in a CDRND. Then, it inserts a questionary expression into the node if it gives *true* for whether the node is included in a CDRND, resulting in suppression of increase in branch divergence. We show the pseudocode of the algorithm of ESRQP in Appendix A.1.

5.2 Propagation of a Query about Speculation and Code Motion Based on the Answer

Utilizing the property that both destinations of a divergent branch are executed, we can speculatively hoist an expression that exists only in one side of a divergent branch out of it without decreasing execution efficiency. We realize the SSRQP through propagating of a query about speculation. We extend ESRQP through adding the speculative code motion, which we call *Speculative SRQP* (SSRQP).

We illustrate the effectiveness of speculative code motion for a divergent branch in **Fig. 5**. In the figure, the shaded nodes 1 and 4 cause branch divergence. In Fig. 5 (a), expressions exist only in one side of the destinations of these branches, and the expression in node 5 is partially redundant. Traditional PRE or scalar replacement cannot remove the redundancy because of safety. On the other hand, utilizing the property of branch divergence, we can hoist the expression in node 2 to node 1 speculatively as shown in Fig. 5 (b). In consequence of this, the expression $A[i]$ becomes available at node 5, so that it can be eliminated as shown in Fig. 5 (c). In addition, comparing Fig. 5 (a) with Fig. 5 (c), we see that branch divergence of the CFG in Fig. 5 (c) is reduced

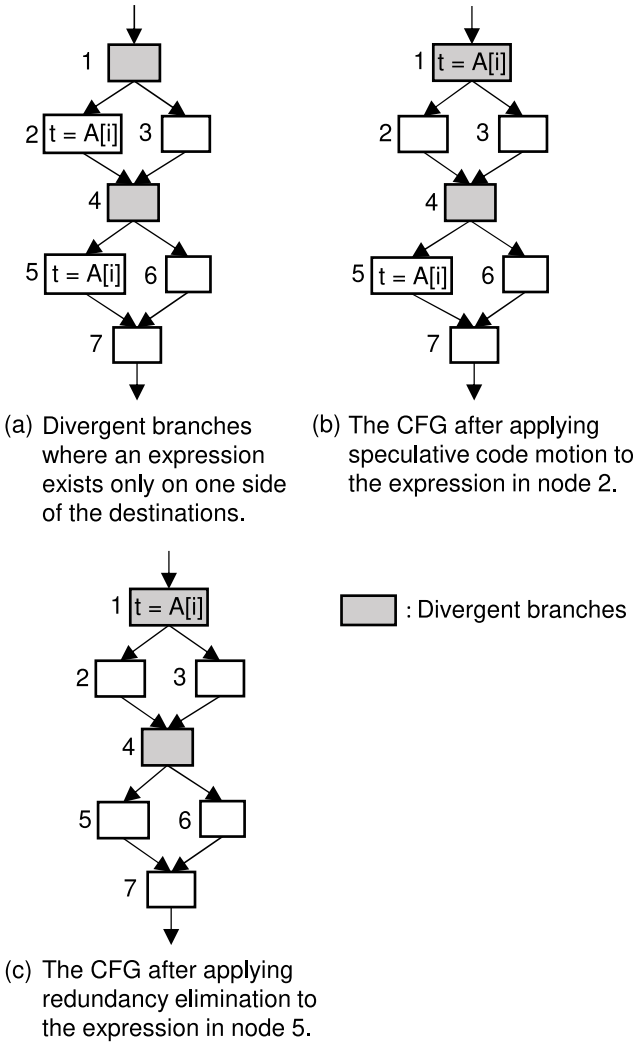

Fig. 5 Speculative code motion for a divergent branch.

more than that in Fig. 5 (a) because the statements at nodes 2 and 5, which are control-dependent on divergent branches, are suppressed. Thus, speculative code motion for a divergent branch makes more expressions available without decreasing execution efficiency and reduces branch divergence.

We propagate a query about speculation to check whether we can hoist an expression out of a divergent branch speculatively. The propagation realizes speculative code motion from node m to node n based on a control dependency relation between m and n. As discussed above, we can reduce branch divergence through hoisting an expression from a node that is control-dependent on a divergent branch out of the branch. The answer of a query about speculation is a tuple of an answer that denotes whether speculative code motion can be performed and a set of nodes to which a questionary expression e is inserted if the answer is *true*. We backwardly propagate the query about speculation (e, m, n) based on the following rules:

(1) If n is the start node **s**, the answer is $(false, \emptyset)$.

(2) If n is a node where the same query has already been propagated, the answer is $(true, \emptyset)$.

(3) If n contains a store statement which may modify the memory location referenced by the questionary expression, or a definition statement of the operands of the questionary ex-

pression, the answer at n is decided by the following rules:
(i) If n is a divergent branch node, the answer is decided by the following rules:
(a) If m is control-dependent on n, the answer is (*true*, $\{n\}$).
(b) If m is not control-dependent on n, the answer is (*false*, $\emptyset$).
(ii) If n is not a divergent branch node, the answer is (*false*, $\emptyset$).

(4) If n is a divergent branch node, the answer is decided by the following rules:
(i) If m is control-dependent on n, the query is propagated further to the predecessors of n to check whether e can be hoisted to an earlier node; therefore m is replaced with n. Let q_{ans} be the answer in which the propagation results. The answer at n is decided by the following rules:
(a) If q_{ans} is (*true*, x), the answer is also q_{ans}.
(b) If q_{ans} is (*false*, x), the answer is (*true*, $\{n\}$).
(ii) If m is not control-dependent on n, the answer is the result of the query propagated further.

(5) If n is a non-divergent branch node, the answer is decided by the following rules:
(i) If n is down-safe, the answer is decided by the following rules:
(a) If m is control-dependent on n, the answer is the result of the query propagated further after replacing m with n.
(b) If m is not control-dependent on n, the answer is the result of the query propagated further.
(ii) If n is not down-safe, the answer is (*false*, $\emptyset$).

(6) If any above rules are not applied, the query is propagated to all of the predecessors of n.

We apply the above rules in order from the top to the bottom. Here, m and n are initialized to the node where a query about speculation originates, A query is propagated to the predecessors p of n while n is replaced with p, so that n represents the node where the query is currently being propagated. Let (*canHoist*, *Node*) be the answer of a query about speculation. If *canHoist* is *true*, we can speculatively hoist e to the nodes in *Node*. Conversely, we cannot perform any transformation if *canHoist* is *false*.

In the process of propagation, it is blocked by a store statement which may modify the memory location referenced by e or a definition statement of the operands of e because e cannot be hoisted beyond such a statement. Thus, as shown in the rule (3), the answer at node n is decided based on whether n with a store statement or a definition statement is a divergent branch node. If n is a divergent branch node, we also check whether m is control-dependent on n. If m is control-dependent on n, the answer is (*true*, $\{n\}$) because the hoisting from m to n contributes to reducing branch divergence and making more expressions available. If m is not control-dependent on n, the hoisting from m to n is ineffective, so that the answer is (*false*, $\emptyset$). These correspond to the rule (3)-(i). On the other hand, If n is not a divergent branch node, the hoisting from m to n is also ineffective, so that the answer is (*false*, $\emptyset$). This corresponds to the rule (3)-(ii).

As shown in the rule (4), if n is a divergent branch node and m is control-dependent on n, the query is propagated further to the predecessors of n after m is replaced with n to check whether e can be hoisted to an earlier node than n. This means that we check whether n is control-dependent on an earlier node that is a divergent branch node. If the result of the further propagation is (*true*, x), e can be hoisted to the nodes in x. Conversely, if it is (*false*, x), e cannot be hoisted to the nodes in x. However, e can be speculatively hoisted to n, so that the answer at n is (*true*, $\{n\}$). These correspond to the rule (4)-(i). On the other hand, if m is not control-dependent on n, the hoisting from m to n is ineffective. Thus, we check whether m is control-dependent on an earlier node that is a divergent branch node through the further propagation. The answer at n depends on the result of the propagation, corresponding to the rule (4)-(ii).

As shown in the rule (5), if n is a non-divergent branch node, we need to check whether n is down-safe. If n is down-safe, we can hoist e from m to n safely, so that we also check whether m is control-dependent on n. If m is control-dependent on n, m is replaced with n to check whether n is control-dependent on an earlier node that is a divergent branch node, and then the query is propagated further to the predecessors of n. If m is not control-dependent on n, the query is propagated further without replacing m with n to check the control dependence between m and an earlier node than n. These correspond to the rule (5)-(i). On the other hand, if n is not down-safe, we cannot hoist e to n safely, so that the answer is (*false*, $\emptyset$), corresponding to the rule (5)-(ii).

We propagate a query about speculation and transform a program based on the answer after ESRQP finds that an expression e which originates a query about availability is not redundant. We show the pseudocode of the algorithm of propagation of a query about speculation in Appendix A.2.

Consider the application of SSRQP to the CFG in **Fig. 6**. In the figure, the shaded nodes 2 and 4 cause branch divergence. First, SSRQP visits node 5 and propagates a query about availability for the expression $A[i+1]$. The result of the propagation is *false*, so that as illustrated in Fig. 6 (a) SSRQP propagates a query about speculation ($A[i+1]$, 5, 4) to the predecessor of node 5 in order to check whether it can speculatively hoist the expression to an earlier node than node 5. When the query visits node 4, the rule (4)-(i) is applied because node 4 is a divergent branch node and does not have a store statement or a definition statement, so that the query ($A[i+1]$, 4, 2) is propagated to node 2. Then, at node 2, the rule (4)-(i) is applied as with node 4, so that the query ($A[i+1]$, 2, 1) is propagated to node 1, and the query ($A[i+1]$, 2, 9) is propagated to node 9. At node 1, the rule (3)-(ii) is applied because the statement $i = 0$ is a definition statement of $A[i+1]$ and node 1 is not a divergent branch node. Thus, the former query gives the answer (*false*, $\emptyset$) at node 1 as shown in Fig. 6 (b). On the other hand, at node 9, the rule (6) is applied, so that the query ($A[i+1]$, 2, 8) is propagated to node 8. Then, at node 8, the query gives the answer (*false*, $\emptyset$) because the rule (3)-(ii) is applied. Consequently, at node 2, the answers obtained from the predecessors are both (*false*, $\emptyset$) as illustrated in Fig. 6 (b). In this case, the rule (4)-(i)-(b) is applied, so that

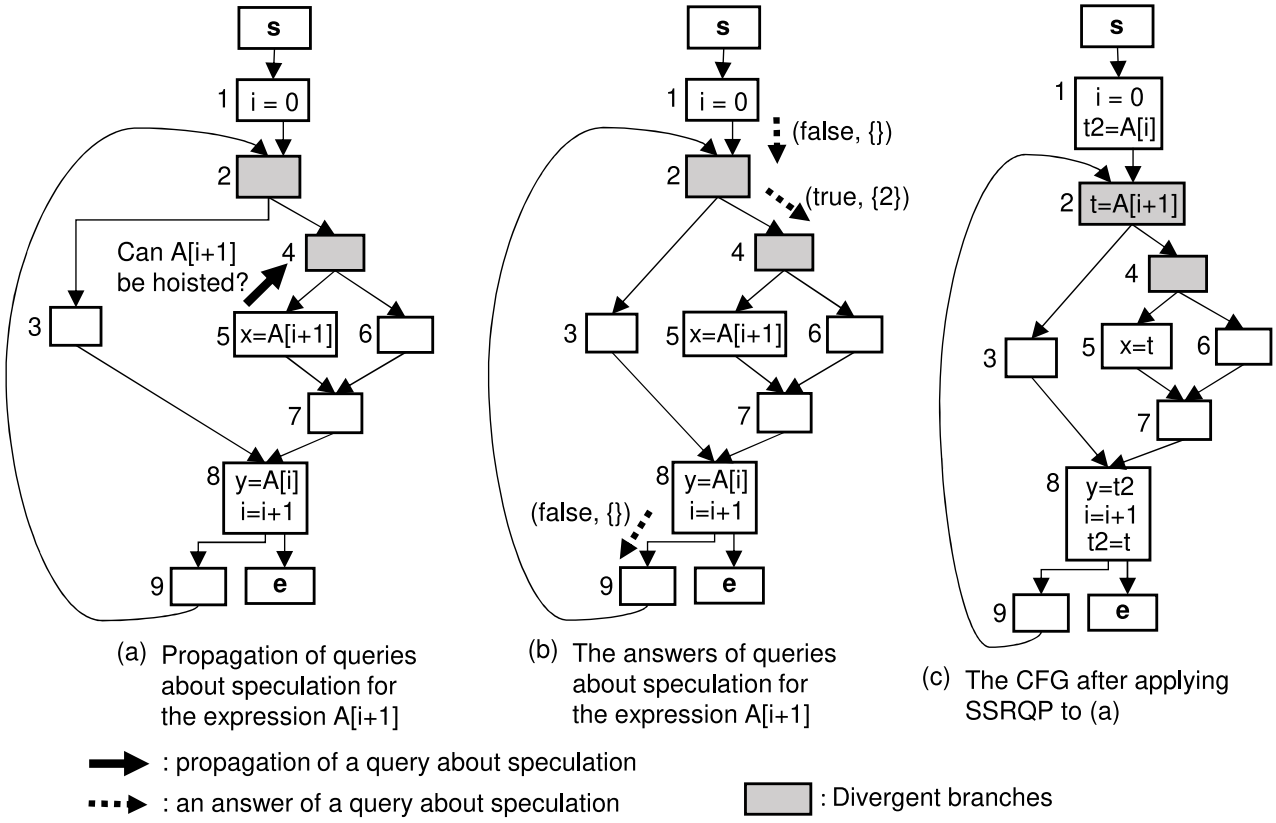

Fig. 6 An example of application of SSRQP.

the answer (*true*, {2}) is returned toward node 5. In consequence, the originating node 5 obtains the answer (*true*, {2}), so that SSRQP hoists $A[i+1]$ from node 5 to node 2 speculatively. Second, SSRQP propagates a query about availability for the expression $A[i]$ at node 8. As a result, it can remove the inter-iteration redundancy between $A[i+1]$ and $A[i]$ because of the speculative code motion of $A[i+1]$ from node 5 to node 2. Finally, it obtains the CFG in Fig. 6 (c).

Remember that SRQP increases branch divergence because it inserts $A[i+1]$ into nodes 3 and 6 as shown in Fig. 4 (f). On the other hand, as illustrated in Fig. 6 (c), SSRQP hoists the expression speculatively, so that it does not increase branch divergence and furthermore makes $A[i+1]$ available across the loop iteration at node 8.

6. Experiments

To evaluate the effectiveness of our method, we compared it with traditional approaches about execution efficiency for three benchmarks. We have implemented our method on the open-source software Ocelot CUDA compiler [27]. The Ocelot is a backend for PTX [32] corresponding to GPU assembly code and also works as a PTX optimizer. Furthermore, we used the divergence analysis [19] implemented in Ocelot to identify divergent branches. The descriptions of environments where we conducted experiments are as follows:

- Environment 1
  - OS: Ubuntu 16.04 LTS,
  - CPU: Intel Core i7-4770K,
  - GPU: Geforce GTX TITAN Black, and
  - CUDA Toolkit 5.0.
- Environment 2
  - OS: Ubuntu 18.04 LTS,
  - CPU: Intel Core i9-9900K,
  - GPU: NVIDIA TITAN RTX, and
  - CUDA Toolkit 11.1.

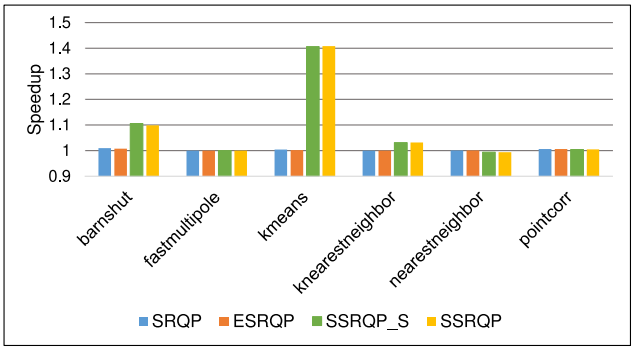

Fig. 9 A comparison of execution speed of Treelogy benchmark in the environment 1. Each result is normalized by the baseline *O3*.

In the experiments, we implemented the proposed method and the related methods SRQP, ESRQP, and SSRQP_S. SRQP neither considers CDRND nor applies propagation of a query about speculation. ESRQP considers CDRND without applying propagation of a query about speculation. SSRQP_S considers CDRND and applies propagation of a query about speculation to move an expression speculatively without considering branch divergence. We compared the execution time of object code for the three benchmarks, Rodinia [29], Treelogy [28], and NVIDIA SDK sample code [31]. For *SobelFilter* and *bilateral-*

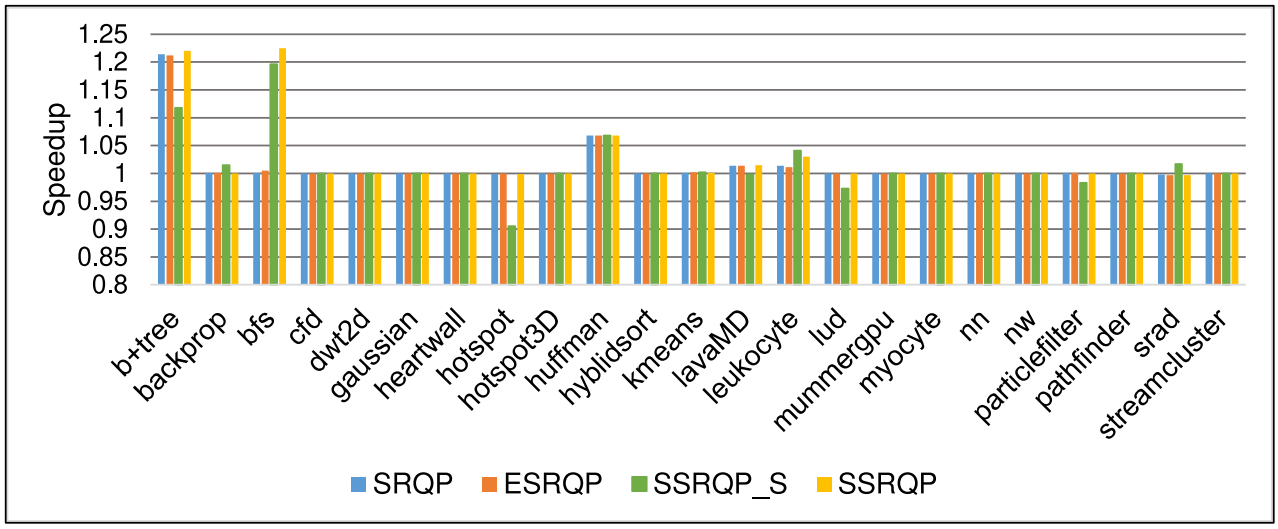

Fig. 7 A comparison of execution speed of Rodinia benchmark in the environment 1. Each result is normalized by the baseline *O3*.

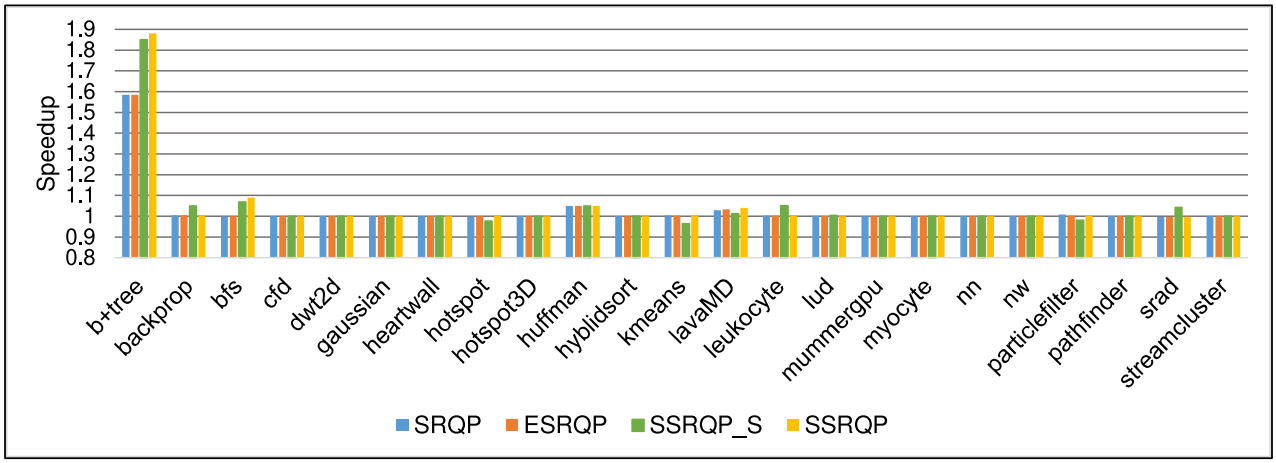

Fig. 8 A comparison of execution speed of Rodinia benchmark in the environment 2. Each result is normalized by the baseline *O3*.

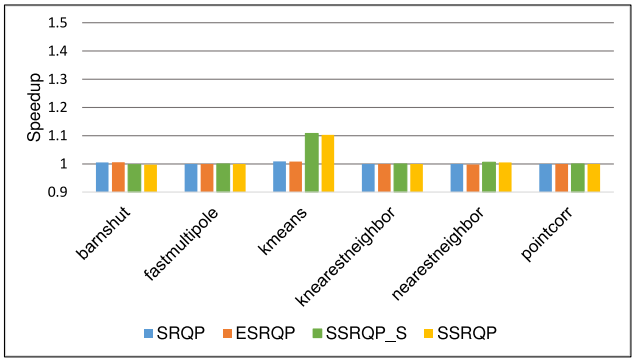

Fig. 10 A comparison of execution speed of Treelogy benchmark in the environment 2. Each result is normalized by the baseline *O3*.

Filter in NVIDIA SDK sample code, we changed the memory access pattern and compared execution efficiency before and after the change. The change will be described later. Each program was executed 10 times. We report the average of all execution time. We measured execution time using the CUDA profiler *nvprof*.

First, we applied these methods to the PTX code generated by the NVIDIA CUDA compiler, *nvcc*. **Figures 7** and **9** show the results of the experiments in the environment 1 for Rodinia and Treelogy benchmark, respectively. **Figures 8** and **10** show the experimental results in the environment 2 for these benchmarks, respectively. Our baseline for comparison is the execution time of object code generated by nvcc with the optimization option O3. In the rest of this paper, we call this baseline *O3*. In the figures, The *SRQP*, *ESRQP*, *SSRQP_S*, and *SSRQP* respectively represent the execution time when applying SRQP, ESRQP, SSRQP_S, and SSRQP to the PTX code used by the baseline *O3*. Each result is normalized by *O3*.

As illustrated in Fig. 7, our method SSRQP improved performance for five programs in Rodinia benchmark in the environment 1: *b+tree*,*bfs*, *huffman*, *lavaMD*, and *leukocyte*. For the program of *b+tree*, SRQP, ESRQP, and SSRQP improved the execution efficiency by 21%, 21%, 22%, respectively. These mothods moved loop-invariant expressions out of the loop. SSRQP could move more loop-invariant expressions out of the loop than SRQP and ESRQP did through hoisting some expressions speculatively. On the other hand, SSRQP_S achieved less efficiency than other method did because it hoisted many expressions without considering branch divergence and introduced new expressions into

some paths. For the program of *bfs*, SSRQP_S and SSRQP improved the execution efficiency by 19.6%, 22.5%, respectively through moving loop-invariant expressions out of the loop. SRQP and ESRQP could not have an opportunity to transform, so that they could not get performance gain. For the program of *huffman*, all methods improved the execution efficiency by 7% through removing intra-iteration redundancy. For the program of *lavaMD*, SRQP, ESRQP, and SSRQP improved the execution efficiency by about 1.5% through removing intra-iteration redundancy. SSRQP_S hoisted many expressions speculatively through the propagation of a query about speculation, but it could not improve the execution efficiency. For the program of *leukocyte*, SRQP and ESRQP could improve the execution efficiency by 1% through removing intra-iteration redundancy. SSRQP_S could remove a lot of intra-iteration redundancy through speculative code motion without considering branch divergence, resulting in the improvement of the efficiency by 4%. Since SSRQP performs speculative code motion that considers branch divergence, it could remove less redundancy than SSRQP_S, resulting in the improvement of the efficiency by 3%. For the programs of *backprop* and *srad*, SSRQP_S could improve the execution efficiency while the other methods could not transform the programs. On the other hand, for the programs of *hotspot*, *lud*, and *particlefilter*, SSRQP_S reduced the execution efficiency of them. This is because SSRQP_S hoists expressions speculatively for not only divergent branches but also non-divergent branches. For the programs of *heartwall*, *kmeans*, *myocyte*, *nw*, and *pathfinder*, SSRQP and other methods had an opportunity to transform, but they could not improve the execution efficiency of those programs. For the other programs, all methods did not have an opportunity to transform, so that they could not get performance gain.

As shown in Fig. 8, the result in the environment 2 showed the almost same tendency as the one in the environment 1. However, for the program of *b+tree*, SSRQP_S improved more execution efficiency than SRQP and ESRQP did. In addition, SSRQP could improve the most efficiency of the four methods. SRQP, ESRQP, SSRQP_S, and SSRQP improved the efficiency by 58.5%, 58.5%, 84.9%, 88%, respectively.

As shown in Fig. 9, SSRQP improved performance for three programs *barnshut*, *kmeans*, and *knearestneighbor* in Treelogy benchmark in the environment 1. SSRQP_S and SSRQP improved the execution efficiency of them by about 10%, 40%, and 3%, respectively while SRQP and ESRQP could not transform the programs. This is because the speculative code motion that SSRQP_S and SSRQP performed made some expressions totally redundant.

As illustrated in Fig. 10, in the environment 2, all methods could not improve performance for five programs in Treelogy benchmark: *barnshut*, *fastmultipole*, *knearestneighbor*, *nearestneighbor*, and *pointcorr*. For the program of *kmeans*, SSRQP_S and SSRQP improved the execution efficiency by about 10%, but it was less efficient than ones in the environment 1.

In these two benckmarks, geometric mean speedup in the environment 1 for SRQP, ESRQP, SSRQP_S, and SSRQP is 1.011, 1.011, 1.026, and 1.034, respectively, and one in the environment 2 is 1.02, 1.02, 1.032, and 1.032, respectively. These methods could not improve execution efficiency of many programs because benchmark programs are well known and have already been heavily optimized. Nevertheless, our method could get great performance gains in three programs. On the other hand, ESRQP obtained the same performance gain as SRQP. Moreover, although SSRQP_S obtained performance gains a little greater than SSRQP in some cases, it obtained much less execution efficiency than SSRQP in some cases. Therefore, SSRQP_S is not practical. From the above, our method, which considers CDRND and speculative code motion based on branch divergence, is the best of these methods.

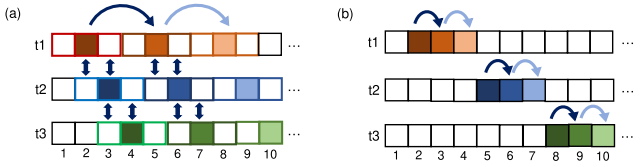

Fig. 11 (a) Coalesced memory accesses. There are redundant memory accesses among threads in each iteration, (b) The access pattern after our modification. There are redundant memory accesses within threads in across iterations.

As mentioned above, for *SobelFilter* and *bilateralFilter* in NVIDIA SDK sample code, we changed the memory access pattern and conducted experiments with them. The access pattern of the original programs was coalesced, so that there were redundant memory accesses among threads, which could not be removed. Therefore, we changed the memory access pattern so that there are redundant accesses within each thread. We show the pattern of the memory accesses before and after our modification in **Figs. 11** (a) and (b). The figure shows the 2nd to 10th elements of an array are calculated on three threads with three iterations of a loop. Assume that we need to access the i-1th and i+1th elements to calculate the i-th element. In Fig. 11 (a), thread `t1` calculates the 2nd, 5th, and 8th elements, `t2` does the 3rd, 6th, and 9th ones, and `t3` does the 4th, 7th, and 10th ones for each loop iteration. There are no redundant memory accesses within each thread, but there are redundant ones among threads. That is, at the first iteration of the loop, since thread `t1` accesses the 1st, 2nd, and 3rd elements, and `t2` does the 2nd, 3rd and 4th ones, the accesses to the 2nd and 3rd elements are redundant between these two threads. As well, since thread `t3` accesses the 3rd, 4th, and 5th elements, the accesses to the 3rd and 4th ones are redundant between `t2` and `t3`. This type of redundancy exists at each iteration, and it is difficult to remove the redundancy. On the other hand, in Fig. 11 (b), in the loop iterations, thread `t1` calculates the 2nd, 3rd, and 4th elements, `t2` does the 5th, 6th and 7th ones, and `t3` does the 8th, 9th and 10th ones, respectively. In this case, there is no redundancy among threads in the same iteration. Instead, there is redundancy across iterations within each thread. Since thread `t1` accesses the 1st, 2nd, and 3rd elements at the 1st iteration of the loop, and it accesses the 2nd, 3rd and 4th ones at the 2nd iteration, the accesses to the 2nd and 3rd ones are redundant in this iteration. This type of redundancy exists in each thread, and it can be removed by scalar replacement.

We compared the execution efficiency before and after changing the memory access pattern as described above. **Figures 12** and **13** show the experimental results of *SobelFilter* and *bilater-*

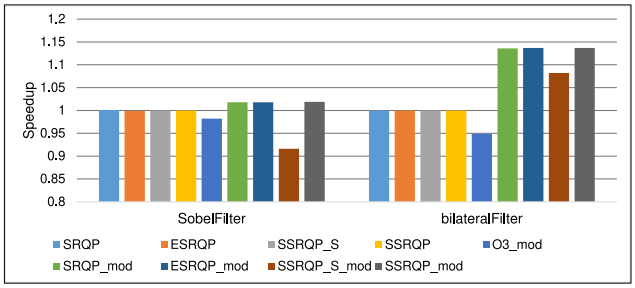

Fig. 12 A comparison of execution speed of *SobelFilter* and *bilateralFilter* in Nvidia SDK in the environment 1. Each result is normalized by the baseline *O3*.

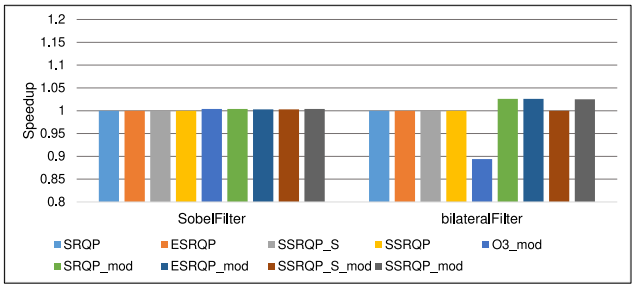

Fig. 13 A comparison of execution speed of *SobelFilter* and *bilateralFilter* in Nvidia SDK in the environment 2. Each result is normalized by the baseline *O3*.

alFilter in the environments 1 and 2, respectively. Each result is normalized by *O3*, which represents the execution time of object code before our modification of the access pattern with the optimization option O3. *SRQP*, *ESRQP*, *SSRQP_S*, and *SSRQP* represent the same ones as in Figs. 7, 8, 9, and 10. *O3_mod* represents the execution time of object code after our modification of the access pattern with the optimization option O3. *SRQP_mod*, *ESRQP_mod*, *SSRQP_S_mod*, and *SSRQP_mod* represent the execution time when respectively applying SRQP, ESRQP, SSRQP_S, and SSRQP to the PTX code of *O3_mod*.

For the program of *SobelFilter*, SRQP, ESRQP, SSRQP_S, and SSRQP could not transform and improve the PTX code of O3. In the environment 1, as shown in Fig. 12, O3_mod decreased the execution efficiency by 2% because the change of the memory access pattern prevents the memory coalescing optimization. SRQP_mod, ESRQP_mod, SSRQP_S_mod and SSRQP_mod could remove inter-iteration redundancy. However, SSRQP_S_mod hoisted expressions speculatively without considering branch divergence, and the number of statements to be executed increased, so that it decreased the execution efficiency by 9% in the environment 1. SRQP_mod, ESRQP_mod and SSRQP_mod could improve the efficiency by 2% in the environment 1 because SRQP_mod and ESRQP_mod do not perform speculative code motion, and SSRQP_mod suppresses speculative code motion for non-divergent branches. In the environment 2, as shown in Fig. 13, all methods could not improve the efficiency after modifying the memory access pattern. O3_mod and SSRQP_S_mod did not decrease the efficiency. Our modification that prevents the memory coalescing optimization and speculative code motion did not affect the performance of this program in the environment 2. For the program of *bilateralFilter*, as well as *SobelFilter*, SRQP, ESRQP, SSRQP_S and SSRQP could not transform and improve the PTX code of O3. In the environment 1, for the same reason as *SobelFilter*, O3_mod decreased the execution efficiency by 5% as shown in Fig. 12. SRQP_mod, ESRQP_mod, SSRQP_S_mod and SSRQP_mod could remove inter-iteration redundancy and improve the efficiency by 14%, 14%, 8%, and 14%, respectively. SSRQP_S_mod was less efficient than other methods for the same reason as *SobelFilter*. In the environment 2, as shown in Fig. 13, O3_mod decreased the execution efficiency by 11%. SRQP_mod, ESRQP_mod, and SSRQP_mod could improve the efficiency by 3%, and SSRQP_S_mod could not improve the efficiency.

Most GPU programs access memory through the pattern shown in Fig. 11 (a). This pattern coalesces memory access, so that it has few memory transactions. In addition, the pattern has a high rate of cache hit because the spatial locality of the pattern is high at the same iteration. In general, the strided memory access shown in Fig. 11 (b) is less efficient than the coalesced access shown in Fig. 11 (a). However, since the access to a register is faster than the memory access, the execution speed of a program can be faster if the data is stored in a register as many as possible. The results of this experiment show that we can improve the execution efficiency of GPU programs by changing the pattern of the memory access and applying scalar replacement to increase accesses through registers.

7. Related Works

Coutinho et al. [19] proposed the *branch fusion* that reduces the computational cost of a divergent branch through combining computations with the same operator in the true and false sides of the branch into a single statement. However, this method may need to insert new branches and select statements. The insertion may decrease execution efficiency of GPU programs. The number of the insertions depends on the order of statements in the original branch. Our method hoists array references out of divergent branches speculatively, so that combining it with the branch fusion results in reduction of more branch divergence than the branch fusion only.

Wu et al. [22] transforms an unstructured control flow graph to structured one, which contributes to reducing branch divergence. In an unstructured CFG, some basic blocks may be executed 2 times or more because of the divergence. On the other hand, in a structured CFG, such redundant execution does not occur. However, this method increases the code size exponentially because it performs transformation through copying code. Anantpur et al. [23] transforms an unstructured CFG to a structured one through the linearization based on the idea of guarded execution of basic blocks. For each basic block of a CFG, the linearization method creates a guard basic block to guard its execution. The mechanism reconverges the divergent threads as early as possible. In addition, it does not duplicate code, so that it incurs only a linear increase in the number of basic blocks. Reissmann et al. [24] proposed control flow restructuring technique that consists of loop restructuring and branch restructuring. Loop restructuring converts all loops to tail-controlled loops, and branch restructuring ensures proper nesting of control flow. These restructuring techniques work by adding predicates and branches

to a CFG, so that they avoid the risk of exponential code inflation. Applying our method together with these method reduces branch divergence further.

Fukuhara et al. [21] proposed the *Speculative Sparse Code Motion* method that reduces branch divergence through hoisting expressions in the true and false sides of divergent branches speculatively. It computes sparse insertion points of expressions by using dataflow analysis. As well as SSRQP, the method allows speculative code motion considering branch divergence to be applied through regarding partially anticipated expressions as anticipated ones at the exit of divergent branch nodes. However, the method cannot take advantage of loop-carried availability because it uses traditional dataflow analyses. In addition, it performs code motion based on sparse insertion points, so that it does not perform speculative code motion as often as SSRQP. The method is exclusive to SSRQP and is expected to have a synergistic effect. Simultaneously applying the method and SSRQP to a program enables eliminating more expressions and reducing branch divergence further.

8. Conclusions

In this paper, we presented a new method for GPU programs to reduce branch divergence through scalar replacement and speculative code motion. Our experimental results have indicated that our method can improve more execution efficiency of GPU programs with branch divergence. However, our method may apply speculative code motion to non-divergent branches because it uses the result of a static divergence analysis. To solve the problem, we hope to achieve selective application based on dynamically checking of branch divergence in the future. In addition, although our method does not insert the destinations of divergent branches through utilizing a CDRND, the restriction may suppress harmless insertion of SRQP. The insertion into the destinations of divergent branches does not always increase execution cost of them. That is, the code motion that SSRQP performs is too conservative. In the future, we hope to develop a method that analyzes an actual execution cost of the programs before and after performing code motion in the presence of branch divergence.

Acknowledgments This work is partially supported by Japan Society for Promotion of Science (JSPS), with the basic research program (C) (No.19K11909), Grant-in-Aid for Scientific Research (KAKENHI).

References

[1] Aho, V.A., Lam, S.M., Sethi, R. and Ullman, D.J.: *Compilers: Principles, Techniques and Tools*, Addison Wesley (1986).
[2] Morgan, R.: *Building an Optimizing Compiler*, Digital Press (1998).
[3] Morel, E. and Renvoise, C.: Global Optimization by Suppression of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
[4] Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (*PLDI 1992*), pp.224–234, ACM (1992).
[5] Knoop, J., Rüthing, O. and Steffen, B.: Optimal Code Motion: Theory and Practice, *ACM Trans. Programming Languages and Systems*, Vol.16, No.4, pp.1117–1155 (1994).
[6] Dhamdhere, M.D.: Practical Adaption of the Global Optimization Algorithm of Morel and Renvoise, *ACM Trans. Programming Languages and Systems*, Vol.13, No.2, pp.291–294 (1991).
[7] Horspool, N.R. and Ho, C.H.: Partial Redundancy Elimination Driven by a Cost-Benefit Analysis, *Proc. 8th Israeli Conference on Computer Systems and Software Engineering*, pp.111–118, IEEE (1997).
[8] Gupta, R., Berson, A.D. and Fang, Z.J.: Path Profile Guided Partial Redundancy Elimination Using Speculation, *Proc. 1998 International Conference on Computer Languages* (*ICCL 1998*), pp.230–239, IEEE (1998).
[9] Cai, Q. and Xue, J.: Optimal and Efficient Speculation-Based Partial Redundancy Elimination, *Proc. International Symposium on Code Generation and Optimization* (*CGO 2003*), pp.91–102, IEEE (2003).
[10] Xue, J. and Cai, Q.: A Lifetime Optimal Algorithm for Speculative PRE, *ACM Trans. Architecture and Code Optimization*, Vol.3, No.2, pp.115–155 (2006).
[11] Callahan, D., Carr, S. and Kennedy, K.: Improving Register Allocation for Subscripted Variables, *Proc. International Conference on Programming Language Design and Implementation* (*PLDI 1990*), pp.53–65, ACM (1990).
[12] Carr, S. and Kennedy, K.: Scalar Replacement in the Presence of Conditional Control Flow, *Software-Practice & Experience*, Vol.24, No.1, pp.51–77 (1994).
[13] Cooper, K., Eckhardt, J. and Kennedy, K.: Redundancy Elimination Revisited, *Proc. 17th International Conference on Parallel Architectures and Compilation Techniques* (*PACT 2008*), pp.12–21, ACM (2008).
[14] Surendran, R., Barik, R. Zhao, J. and Sarkar, V.: Inter-iteration Scalar Replacement Using Array SSA Form, *Proc. Compiler Construction* (*CC 2014*), pp.40–60, Lecture Notes in Computer Science (2014).
[15] Rosen, K.B., Wegman, N.M. and Zadeck, K.F.: Global Value Numbers and Redundant Computations, *Proc. 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 1988*), pp.12–27, ACM (1988).
[16] Takimoto, M.: Speculative Partial Redundancy Elimination Based on Question Propagation, *IPSJ Trans. Programming*, Vol.2, No.5, pp.15–27 (2009) (in Japanese).
[17] Sumikawa, Y., Ojima, R. and Takimoto, M.: Demand-driven Scalar Replacement, *Computer Software*, Vol.32, No.2, pp.93–113 (2015) (in Japanese).
[18] Sumikawa, Y. and Takimoto, M.: Effective Demand-driven Partial Redundancy Elimination, *IPSJ Trans. Programming*, Vol.6, No.2, pp.33–44 (2013).
[19] Coutinho, B., Sampaio, D., Pereira, M.Q.F. and Meira, W. Jr.: Divergence Analysis and Optimizations, *Proc. 2011 International Conference on Parallel Architectures and Compilation Techniques* (*PACT 2011*), pp.320–329, IEEE (2011).
[20] Han, D.T. and Abdelrahman, S.T.: Reducing Branch Divergence in GPU Programs, *Proc. 4th Workshop on General Purpose Processing on Graphics Processing Units* (*GPGPU-4*), pp.1–8, ACM (2011).
[21] Fukuhara, J. and Takimoto, M.: Branch Divergence Reduction Based on Code Motion, *Journal of Information Processing*, Vol.28, pp.302–309 (2020).
[22] Wu, H., Diamos, G., Li, S. and Yalamanchili, S.: Characterization and Transformation of Unstructured Control Flow in GPU Applications, *Proc. 1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems* (*CACHES 2011*), ACM (2011).
[23] Anantpur, J. and Govindarajan, R.: Taming Control Divergence in GPUs through Control Flow Linearization, *Proc. International Conference on Compiler Construction* (*CC 2014*), pp.133–153, Springer (2014).
[24] Reissmann, N., Falch, L.T., Bjørnseth, A.B., Bahmann, H., Meyer, C.J. and Jahre, M.: Efficient Control Flow Restructuring for GPUs, *Proc. 2016 International Conference on High Performance Computing & Simulation* (*HPCS 2016*), pp.48–57, IEEE (2016).
[25] Lin, H., Wang, C. and Liu, H.: On-GPU Thread-Data Remapping for Branch Divergence Reduction, *ACM Trans. Architecture and Code Optimization*, Vol.15, No.3 (2018).
[26] Ferrante, J., Ottenstein, J.K. and Warren, D.J.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Programming Languages and Systems*, Vol.9, No.3, pp.319–349 (1987).
[27] Diamos, G., Kerr, A., Yalamanchili, S. and Clark, N.: Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems, *Proc. 19th International Conference on Parallel Architectures and Compilation Techniques* (*PACT 2010*), pp.353–364, ACM (2010).
[28] Hegde, N., Liu, J., Sundararajah, K. and Kulkarni, M.: Treelogy: A Benchmark Suite for Tree Traversals, *Proc. IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS 2017*), pp.227–238, IEEE (2017).
[29] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, W.J., Lee, S. and Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing, *Proc. IEEE International Symposium on Workload Characterization* (*IISWC 2009*), pp.44–54, IEEE (2009).
[30] NVIDIA Corporation: CUDA C++ Programming Guide, NVIDIA Developer (online), available from ⟨https://docs.nvidia.com/cuda/cuda

-c-programming-guide/index.html⟩ (accessed 2021-04-26).
[31] NVIDIA Corporation: CUDA Samples, NVIDIA Developer (online), available from ⟨https://docs.nvidia.com/cuda/cuda-samples/index.html⟩ (accessed 2021-04-26).
[32] NVIDIA Corporation: PTX: Parallel Thread Execution ISA, NVIDIA Developer (online), available from ⟨https://docs.nvidia.com/cuda/parallel-thread-execution/index.html⟩ (accessed 2021-04-30).
[33] NVIDIA Corporation: Inside Volta: The World's Most Advanced Data Center GPU, NVIDIA Developer (online), available from ⟨https://developer.nvidia.com/blog/inside-volta⟩ (accessed 2021-08-20).

Appendix

A.1 The algorithm of Extended SRQP based on a CDRND

Algorithm 1 Extended SRQP based on a CDRND

function Propagate(n, q)
  let $isDownSafe$:= $antqp(n, q)$
  let $N_f := \emptyset$
  for all $p \in pred(n)$
    let $(isAvail_p, isReal_p) := Local(p, q)$
    if $\overline{isAvail_p}$ **then add** p **to** N_f
  let $isAvail := \prod_{p \in pred(n)} isAvail_p$
  let $isReal := \bigcup_{p \in pred(n)} isReal_p$
  if both *true* and *false* are in $isAvail_p$ **then**
    if $N_f \subseteq CDRND$ **then**
      if $isAvail \vee (isDownSafe \wedge isReal)$ **then**
        add N_f **to** *insert_dst*
        $isAvail := true$
  if $isAvail \vee (isDownSafe \wedge isReal)$ **then**
    return ($isAvail$, $isReal$)
  else return (*false*, *false*)
function Local(n, q)
  if n = **s then return** (*false*, *false*)
  if $answer[n] \neq \perp$ **then return** $answer[n]$
  if $visited[n] > 1$ **then return** (*false*, *false*)
  if $query[n] = q$ **then return** (*true*, *false*)
  $query[n] := q$
  $visited[n]$++
  for i=$instSize(n)$ **to** 0
    let $inst := getInstruction(n,i)$
    if $mayAlias(q, inst)$ **then**
      $answer[n] := (false, false)$
      return (*false*, *false*)
    if *isSameVal(q, inst)* **then**
      $answer[n] := (true, true)$
      return (*true*, *true*)
    if $isDefVal(q, inst) \wedge visited[n] = 1$ **then**
      $updateQuery(q, inst)$
  let $rlt := Propagate(n, q)$
  $answer[n] := rlt$
  return rlt

A.2 The Algorithm of Propagation of a Query about Speculation

Algorithm 2 Propagation of a query about speculation

function PropagateSpeculativeQuery(e, m, n)
  for all $p \in pred(n)$
    let $(canHoist_p, Node_p) := Local\_Spec(e,m,p)$
    if $canHoist_p$ is *false* **then add** p **to** N_p
  let $canHoist := \prod_{p \in pred(n)} canHoist_p$
  let $Node := \bigcup_{p \in pred(n)} Node_p$
  if (both *true* and *false* are in $canHoist_p$) $\wedge$ ($N_p \subseteq$ CDRND) $\wedge$ (n is down-safe) **then**
    $Node := Node \cup N_p$
    $canHoist := true$
  if $canHoist$ **then return** (*true*, $Node$)
  else return (*false*, $\emptyset$)
function Local_Spec(e, m, n)
  if n = **s then return** (*false*, $\emptyset$)
  if $answer[n] \neq \perp$ **then return** $answer[n]$
  if $query[n]$ = q **then return** (*true*, $\emptyset$)
  $query[n] := (e, m)$
  let $rlt := \perp$
  if $containsMayAlias(e,n) \vee containsDefVal(e,n)$ **then**
    if n is a divergent branch node **then**
      if m is control-dependent on n **then**
        $rlt := (true, \{n\})$
      else $rlt := (false, \emptyset)$
    else $rlt := (false, \emptyset)$
  else
    if n is a divergent branch node **then**
      if m is control-dependent on n **then**
        $m := n$
        let $(canHoist, Node) := PropagateSpeculativeQuery(e, m, n)$
        if $canHoist$ is *false* **then**
          $(canHoist, Node) := (true, \{n\})$
        $rlt := (canHoist, Node)$
      else $rlt := PropagateSpeculativeQuery(e, m, n)$
    else if n is a non-divergent branch node **then**
      if n is down-safe **then**
        if m is control-dependent on n **then**
          $m := n$
          $rlt := PropagateSpeculativeQuery(e, m, n)$
        else $rlt := PropagateSpeculativeQuery(e, m, n)$
      else $rlt := (false, \emptyset)$
    else $rlt := PropagateSpeculativeQuery(e, m, n)$
  $answer[n] := rlt$
  return rlt

Junji Fukuhara received his B.S. and M.S. degrees from Tokyo University of Science in 2018 and 2020, respectively. He is currently in the second year of the Ph.D. course at Tokyo University of Science.

Munehiro Takimoto is a professor in the Department of Information and Sciences from Tokyo University of Science. His research interests include theory and practice of programming languages, and the various things derived from them, which include mobile agent systems and their applications. He received his PhD, MS, and BA in Engineering from Keio University. He is a member of ACM, IEEE Computer Society, IPSJ, JSSST, and IEICE.