Compiling and Optimizing Real-world Programs for STRAIGHT ISA

Toru Koizumi, Shu Sugita, Ryota Shioya, Junichiro Kadomoto, Hidetsugu Irie, Shuichi Sakai
Graduate School of Information Science and Technology, The University of Tokyo Tokyo, Japan
Email: {koizumi, sugita, kadomoto, irie, sakai}@mtl.t.u-tokyo.ac.jp, shioya@ci.i.u-tokyo.ac.jp

***Abstract*—The renaming unit of a superscalar processor is a very expensive module. It consumes large amounts of power and limits the front-end bandwidth. To overcome this problem, an instruction set architecture called STRAIGHT has been proposed. Owing to its unique manner of referencing operands, STRAIGHT does not cause false dependencies and allows out-of-order execution without register renaming. However, the compiler optimization techniques for STRAIGHT are still immature, and we found that the naive code generators currently available can generate inefficient code with additional instructions. In this paper, we propose two novel compiler optimization techniques and a novel calling convention for STRAIGHT to reduce the number of instructions. We compiled real-world programs with a compiler that implemented these techniques and measured their performance through simulation. The evaluation results show that the proposed methods reduced the number of executed instructions by 15% and improved the performance by 17%.**

***Index Terms*—instruction-level parallelism, optimizing compiler, callee-saved registers**

I. Introduction

As Amdahl's law suggests, the single-thread performance of a core for executing sequential portions of a program is essential, even in the age of multi/many-core processors. The current mainstream strategy for improving single-thread performance is to extract more instruction-level parallelism by increasing the processing width of a core and the size of the scheduling window [1]–[3]. As a result, even cores in mobile devices, such as the ARM Cortex A77, can execute more than ten instructions in a single cycle [4].

However, an increase in the processing width of a core can lead to a superlinear increase in the circuit area, resulting in a significant increase in power consumption. One of the units that causes such an increase is the renaming unit in the front-end of the core. The renaming unit renames register numbers using a table called a register map table (RMT)[1] to remove false dependencies between instructions. The RMT mainly consists of a multiport memory, with the number of ports proportional to the processing width. The circuit area rapidly increases as the processing width of the renaming unit increases [5], [6].

To overcome this problem, an instruction set architecture (ISA) called STRAIGHT has been proposed [7]. STRAIGHT has a register file where registers are sequentially allocated and refers to operands, not by register number, but by *distance* between instructions. Because of this new register architecture, STRAIGHT has no false dependencies and does not require register renaming [7]. As a result, the renaming logic itself can be omitted in the STRAIGHT processor; this removes the front-end bottleneck and improves the scalability of the reorder buffer (ROB).

In STRAIGHT, compiler optimization techniques are still immature, and we have found that the naive code generators currently available can produce inefficient code. In regular ISAs, optimization is traditionally well studied, and optimization techniques such as code generation and register allocation are mature. However, the optimization techniques for STRAIGHT remain unexplored. This is due to the following two reasons: 1. STRAIGHT refers to operands by inter-instruction distance, so existing optimization techniques for register allocation cannot be applied. 2. STRAIGHT has inherent constraints, owing to its distance referencing. In existing research, the authors have manually optimized small benchmarks, such as Dhrystone, to achieve performance comparable to the RISC-V ISA [7]. However, the properties of STRAIGHT for larger real-world programs are not clear. We implemented a STRAIGHT toolchain to compile real-world programs, evaluated the SPEC CPU benchmarks, and found that the existing compiler can significantly increase the number of instructions generated.

In this paper, we propose three new compiler optimization techniques for STRAIGHT. The first reduces the number of instructions generated by reordering the instructions, while satisfying the inherent constraints of STRAIGHT. This optimization corresponds to register-allocation optimization in general instruction sets. The second is called aggressive spill; it improves performance by deliberately spilling out values, even when sufficient registers are available[2]. The third is a novel function-calling convention that implements callee-saved registers (CSRs) in STRAIGHT. The method of implementing callee-saved registers in STRAIGHT has not been clear; however, we propose a method to implement it in this paper.

We implemented these optimization techniques in a compiler using the LLVM [9] compiler infrastructure and evaluated its performance using a simulator. We compiled the programs contained in the SPEC CPU 2006/2017 benchmark suites and

This work was partially supported by JSPS KAKENHI Grant Numbers JP19H04077, JP20H04153, JP20J22752.

[1]It is also known as a register alias table or register allocation table (RAT).

[2]This is an extended version of our content presented at a workshop [8]. We extended the proposed method and increased the number of evaluations.

used them in our evaluation. The evaluation results showed that the optimization techniques reduced the number of executed instructions by 15% and improved the performance by 17% on average.

II. Existing basic compiler algorithms for STRAIGHT

A. STRAIGHT instruction set architecture

STRAIGHT is an ISA that uses inter-instruction distances rather than register names to specify the source operands. In STRAIGHT, the source operand is specified in the form, "use the result of the n^{th}-previous instruction." In the assembly representation, square brackets are used to indicate this. For example, when referencing the result of an instruction executed two instructions previously, `[2]` is used as the operand. Each instruction implicitly reserves one destination register and uses it in a write-once manner. Owing to the instruction-length limitation, there is a maximum value defined in the ISA for the distance that can be used to specify the source operand. Therefore, the results of old instructions become unreferenceable after a certain number of instructions have been executed. Taking advantage of this feature, we can use a ring buffer for the register file. The destination registers can be allocated in order from the ring buffer. Even so, it is guaranteed that only registers that are out of life will be overwritten. Register renaming is no longer necessary because there are no false dependencies [7].

STRAIGHT uses a calling convention that uses relative distances from branch instructions used for function calls and returns (such as the `jal` (jump and link) and `ret` (return) instructions). Expressly, the instructions that generate the argument values are placed so that the instruction immediately before the function call instruction generates the first argument, and the instruction two previous generates the second argument, and so forth. In addition, the instruction to generate the return value is placed so that the instruction immediately before the return instruction generates the return value.

B. STRAIGHT-specific code generation

In the following, we explain the "distance fixing" algorithm [7], which is specific to the STRAIGHT compiler; that is, the parts where conventional RISC compiler algorithms cannot be applied. Conventional algorithms can be used for conversion from high-level programming languages to the compiler's intermediate representation, and for optimization in static single-assignment (SSA) form [10], [11].

When generating STRAIGHT code, the RISC code-generation algorithm can be applied to most parts; however, a different algorithm is required for register allocation. This is because STRAIGHT, unlike RISC, is an instruction set whose instruction specifies operands by distance. A new algorithm is needed for code generation when selecting an operand

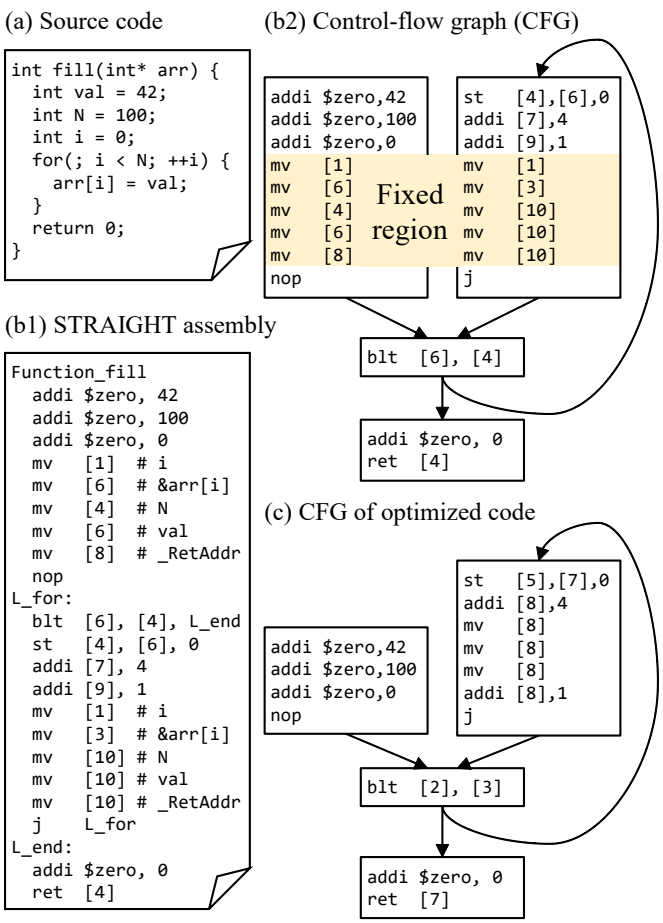

Fig. 1. Example of building a fixed region: The live variables are transferred in a common order to the area indicated in yellow at the end of the basic block (called the fixed region).

that depends on the execution path[3]. In RISC, this problem is solved by using the same register in all execution paths, while in STRAIGHT, the problem is solved by matching the distances for all execution paths. The following is an algorithm for generating STRAIGHT code from a static single-assignment format.

1) Enumerate values that are live at a merging point and add a phi function for each live variable.
2) For each value referenced by the phi function, add a `mv` (move or register-transfer) instruction just before the merging point. By doing this, the referenced value is generated at the same relative position in each path.

When instructions are added using this algorithm, the final instruction sequence is as shown in Fig. 1 (b).

[3]In RISC, this corresponds to the case of creating a phi function in a static single-assignment format, whereas in STRAIGHT, it includes the case where an instruction after a merging point refers to the result of an instruction before branching. This is because the distance between instructions generally differs when the execution paths differ.

III. Optimizing compiler techniques for STRAIGHT

In the following, we explain three proposals that contribute to performance. The first is an algorithm to reduce the redundant instructions added during "distance fixing," as described in Section II-B, by reordering the instruction order. The second is an optimization that reduces the number of executed instructions by aggressively spilling out some live values, even if there are enough registers. The third is a proposal for a calling convention that introduces callee-saved registers.

A. Distance adjustment to remove redundant instructions

1) Overview: Existing basic compiler algorithms add redundant `mv` instructions in many cases. For example, in Fig. 1 (b), the operands referenced by the `blt` (branch less than) instruction are placed four and six instructions before, regardless of the execution path, by being relayed via the `mv` instruction. The existing algorithm guarantees the compilation of arbitrary programs by creating a *fixed region*, which contains `mv` instructions to relay the values just before the merging point. However, we found that this method generated many redundant `mv` instructions. For example, `mv` instructions can be removed by reordering the instructions so that the instructions after merging can directly reference the operands before merging, as shown in Fig. 1 (c).

We find that the addition of the `mv` instruction is inevitable in the following two situations:

1) When holding a loop constant. In this case, at least one `mv` instruction per loop constant is required in the loop to refer to it by a static distance, regardless of the number of times to loop.
2) When the instruction order cannot be reordered because of restrictions on the execution order, such as a true dependency. In this case, some `mv` instructions are required because it is impossible to refer to the values directly without building a fixed region.

The algorithm to find the appropriate positions to add such `mv` instructions consists of the following two steps. Step 1 involves adding `mv` instructions to hold the loop constants and preprocessing for Step 2. Step 2 involves adding `mv` instructions to reorder the instruction order.

2) Step 1: Add `mv` instructions to hold loop constants: To refer to a loop constant by a static distance, regardless of the number of times the loop is performed, at least one `mv` instruction is required in the loop. When adding phi functions, as described in Section II-B, phi functions that define the same loop constant refer to each other and form a loop (Fig. 2 (a)). This reference-relationship loop can be eliminated by adding one `mv` instruction at any point in the loop (Fig. 2 (b)). In other words, this type of loop can be eliminated by adding `mv` instructions at the position corresponding to the feedback edge set problem. The graph that we should consider has vertices corresponding to phi functions and edges corresponding to reference relations.

We provide a detailed implementation of this method. We perform the following procedure for each feedback edge.

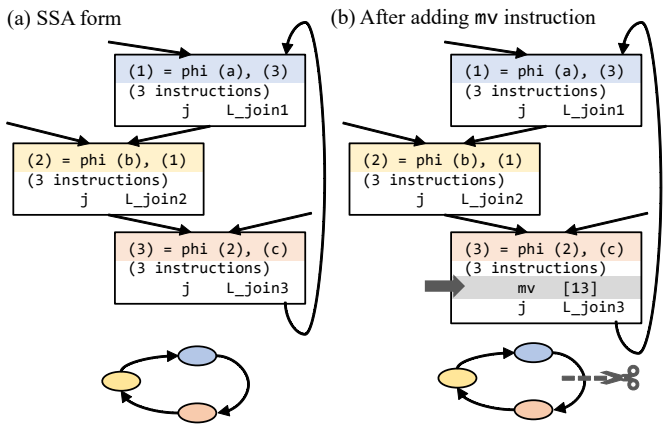

Fig. 2. Adding `mv` instructions to hold loop constants. The location of the `mv` instruction corresponds to the solution of the feedback edge set problem.

First, we add `mv` instructions that refer to a phi function corresponding to the destination of the feedback edge to the basic block where the phi function is located. Then, we change the reference of the phi function corresponding to the source of the feedback edge to the added `mv` instruction.

In terms of implementation, the graph to be constructed is a directed multigraph with vertices corresponding to the basic block. This modification is needed because Step 1 also serves as a pre-processing step for Step 2. To construct this graph, we add a directed edge from A to B, for each phi function in basic block A that references a phi function in basic block B.

After adding the `mv` instructions, the graph of the reference relationship of the phi function becomes a directed acyclic graph (DAG). Every DAG has at least one topological ordering. The following Step 2 is performed for each corresponding basic block in its topological order.

3) Step 2: Add `mv` instructions to eliminate dependency inversions: The values referenced after the merging point must be placed at the same relative distance regardless of the execution path. However, it is difficult to solve this problem directly. Thus, we solve the relaxed problem of placing values in the same order regardless of the execution path. If we solve this problem, it is straightforward to satisfy the original constraints by adding `nop` (no operation) instructions as appropriate.

In many cases, the constraint, "the values are generated in the same order in all execution paths," can be satisfied by swapping instructions. This is not possible when the order of instructions has constraints owing to a dependency, and the order constraints in multiple execution paths contradict each other. In such a case, adding the `mv` instructions needs to remove the dependency.

Whether this constraint is inconsistent can be solved by detecting a cycle in the directed graph with the dependencies as edges. By solving the feedback edge set problem of the directed graph, it is possible to derive where to add the `mv` instruction. However, the addition of a single `mv` instruction may solve multiple constraints simultaneously. Therefore, minimizing the number of `mv` instructions to be added may

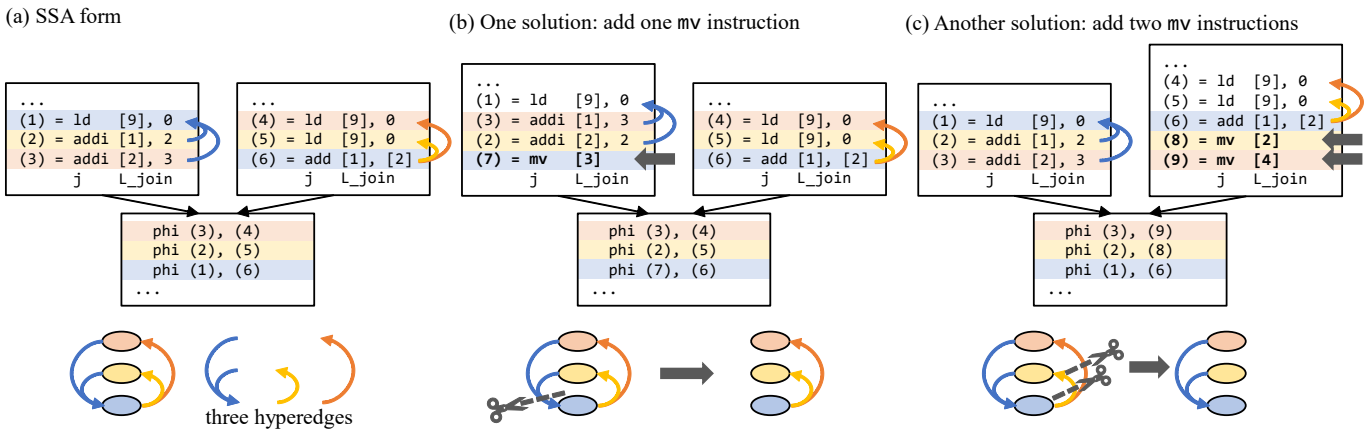

Fig. 3. Adding mv instructions to eliminate dependency inversions. (a) Two execution paths have conflicting execution-order constraints. The hypergraph has three hyperedges, which form loops. (b) Adding one mv instruction resolves the constraints. This corresponds to the best solution of the feedback hyperedge set problem. (c) Adding two mv instructions also resolves the constraints. This corresponds to another solution of the feedback hyperedge set problem.

not correspond to the best solution to the feedback edge set problem. The problem to be solved for minimizing it corresponds to the feedback hyperedge set problem of a hypergraph.

The method of constructing the hypergraph with vertices corresponding to the phi functions is as follows. First, for each execution-order constraint, a directed edge is created. The edge's destination corresponds to a phi function that references the instruction's result to be executed first in a path. The source of the edge corresponds to a phi function that references the instruction's result to be executed later in the path. Then, the directed edges with the same destination for each execution path are grouped and converted into a directed hyperedge of the following form, e.g., $\{\mathrm{src} : \mathrm{v}_1, \mathrm{v}_2, \mathrm{v}_3; \mathrm{dst} : \mathrm{v}_4\}$ (Fig. 3). A hyperedge in this hypergraph represents a constraint that is solved by adding a mv instruction.

The location in which to add the mv instructions is derived from the solution of the feedback hyperedge set problem. Where the mv instructions should be added corresponds to the feedback hyperedges (Fig. 3). We need to add a mv instruction whose source operand is the result of the instruction referenced by the phi function corresponding to the destination of a hyperedge. Then, we should change the reference of the phi function to the result of the added mv instruction.

After adding the mv instructions, the hypergraph of the execution-order constraints becomes a directed acyclic hypergraph. Because this graph has no cycles, we can calculate a topological ordering. Then, the execution order of each instruction that generates a result that references the phi function can be reordered to be in that topological order. After reordering the instructions, we appropriately insert nop instructions, starting from the end of the basic block. Instructions at the same relative distance will generate results referenced to the phi function in all execution paths.

B. *Aggressive spilling*

In STRAIGHT, the mv instruction is executed once per loop to hold a loop constant and it is needed for each loop constant. This is because the distance needs to be the same, regardless of the number of times the loop is performed. The distance adjustment optimization cannot eliminate this problem.

Even if the variable is a loop constant that is not referenced in the loop (hereinafter referred to as a non-referenced loop constant), it still needs to be relayed by executing a mv instruction. We found that when there is a reference across the loop, the referenced variable act as a non-referenced loop constant; the generated code requires many dynamic instructions to relay it.

We propose an optimization method that reduces the number of instructions to be executed by spilling out these variables. We call this optimization "aggressive spill" because it spills out, even if the number of logical registers is sufficient. Although this slightly increases the number of load/store instructions, it is worth doing because reducing the number of executed mv instructions has significant effect, if the loop is executed in sufficiently large numbers.

The procedure for spill-code insertion by this optimization can be treated in a unified manner with the procedure for spill-code insertion when there is a reference across function calls. The details are given in Appendix.

C. *STRAIGHT calling conventions with callee-saved registers*

The existing calling conventions of STRAIGHT lack callee-saved registers, and the lack of callee-saved registers increases the number of load/store instructions. Because the number of instructions in a function is generally unknown, references across function calls cannot specify their operands by distance. STRAIGHT does not have named registers because of the characteristics of the instruction set, so it is impossible to set registers as callee-saved. For this reason, the existing calling convention defines all registers as caller-saved [7]. This causes

(a) Source code

```
int f(int x) {
  return x + 42;
}
int g(int val) {
  ...
  int i = 0;
  int prod = 1;
  for(; i < 100; ++i) {
    prod *= f(i);
  }
  ...
}
```

(b) Existing calling conventions

```
Function_f
  add  [2], 42
  ret  [2]
Function_g
  ...
  addi $zero, 0 # i
  addi $zero, 1 # prod
  nop
L_for:
  addi $zero, 100
  blt  [4], [1], L_end
  stsp [5], 4 # i
  stsp [5], 8 # prod
  mv   [7]    # arg0 = i
  jal  Function_f
  ldsp 8      # prod
  ldsp 4      # i
  mul  [2], [4]
  j    L_for
L_end:
  ...
```

(c) Proposed calling conventions (# of CSRs is 3)

```
Function_f
  add  [5], 42
  mv   [5] # CSR2 relay
  mv   [5] # CSR1 relay
  mv   [5] # CSR0 relay
  ret  [5]
Function_g
  ...
  addi $zero, 0 # i
  addi $zero, 1 # prod
  nop
L_for:
  addi $zero, 100
  blt  [4], [1], L_end
  mv   [5] # arg0 = i
  nop      # CSR2
  mv   [7] # CSR1 = i
  mv   [7] # CSR0 = prod
  jal  Function_f
  mv   [3]
  mul  [3], [5]
  j    L_for
L_end:
  ...
```

Fig. 4. Calling conventions in existing research [7] and proposed calling conventions. CSR refers to callee-saved registers. In the proposed calling conventions, we add arguments for the CSRs after the first argument of the function and add return values for relaying the CSRs after the return value of the function.

large numbers of memory accesses when calling small leaf functions, which leads to performance degradation.

To address this problem, we propose a convention for introducing callee-saved registers in STRAIGHT. This convention uses the relative position to the branch instruction, similar to the existing calling conventions described in Section II-A. It uses the relative distance from the branch instruction to specify the callee-saved registers. Hence, callee-saved registers can be dealt with in the same manner as arguments and return values by adding extra arguments and return values to the function.

Fig. 4 shows how the code changes, as a result of a change in conventions. In this convention, the caller passes the values to be preserved across calls as extra arguments. In the callee, `mv` instructions are executed to relay the values passed as extra arguments directly to the corresponding return values. This allows the caller to receive them as return values across calls. Thus, it is possible to specify an operand across the function call by a static distance while delaying the spill decision until the execution of the callee.

The STRAIGHT compiler can use conventional compiler techniques to select values for callee-saving. However, in STRAIGHT, the number of instructions for placing arguments in the caller and relaying values in the callee increases by setting the callee-saved register. Therefore, to obtain the full effect of callee-saving, it is necessary to avoid adding redundant instructions. In particular, variables that are subject to aggressive spill optimization should not be subject to callee-saving.

IV. Evaluation

A. Methodology

The evaluation was performed by measuring the number of cycles required to execute the benchmark programs through simulations. The benchmark programs used for the evaluation were 401.bzip2, 605.mcf_s, 619.lbm_s, 657.xz_s, and CoreMark [12]. The first four are benchmark programs from real-world programs in SPEC CPU 2006 [13] and SPEC CPU 2017 [14] benchmark suites, all written in C. CoreMark is a synthetic benchmark and not a real-world program; however it was used for comparison with the previous study [7].

The benchmark programs were compiled using our optimizing STRAIGHT compiler. The base STRAIGHT compiler was acquired from the author of an existing study and was based on version 7 of LLVM [9]. We implemented the proposed optimizations on it.

For the C library, we used *musl* libc [15], a lightweight libc implementation. This was because the base STRAIGHT compiler was for bare metal and could not compile programs that required the C library. We ported musl libc for STRAIGHT. The changes we made for porting are described below.

In addition, RISC-V [19] was chosen as the existing architecture for comparison. The optimizing compiler for RISC-V was *llc* static compiler included in version 9 of LLVM. This is the closest to the LLVM version of the compiler for STRAIGHT that can correctly compile RISC-V. To make the comparison fair, we used musl with the above changes to compile the RISC-V binaries.

A cycle-accurate simulator, Onikiri2 [20], was used for the simulation. First, we confirmed that the overall program behavior was correct by using a userland-only instruction-set simulator. To compare the performance of the different binaries, the performance was measured using the same region. The regions were specified in the form of the number of times line A was executed to the number of times line B was executed. Because it would be too time-consuming to simulate all the benchmark programs, we selected a region around the function with the most instructions to be executed in RISC-V for each benchmark. The details of the measurement sections and their characteristics are listed in Table I.

Table II lists the parameters of the processors used in the simulations. We prepared several parameters with different front-end widths and reorder-buffer (ROB) capacities, which were considered to increase the scalability by STRAIGHT. We referred to the Apple Firestorm core [21] to determine the number of execution units and cache parameters. The RISC-V model has a two-cycle-longer front-end latency owing to the

TABLE I
THE MEASUREMENT SECTIONS AND THEIR CHARACTERISTICS

Benchmark	Measurement sections	Characteristics
`CoreMark`	10 iterations	Synthetic benchmark
`401.bzip2`	From the 1st execution of the `mainGtU` to its 100,000th execution	Burrows–Wheeler transform (string comparison and sort)
`605.mcf_s`	From the 1st execution of the `primal_bea_mpp` to its 5th execution	Thrashing access and quicksort
`619.lbm_s`	100,000 iterations of the loop in `LBM_performStreamCollideTRT`	Many long chains of floating-point number instructions
`657.xz_s`	From the 1st execution of the `sha_compress` to its 300th execution	Bit manipulations and loads/stores unfriendly to store set

TABLE II
THE PARAMETERS OF THE PROCESSOR USED IN THE SIMULATION

	`6-fetch`	`8-fetch`	`12-fetch`	`16-fetch`
Front-end width	6	8	12	16
Front-end latency	STRAIGHT: 5 cycles, RISC-V: 7 cycles (fetch + decode + rename + dispatch)			
Issue width	16			
Issue latency	4 cycles (issue + register read)			
Execution unit	Int×8, Float×4, Load×3, Store×2, iMul×2, iDiv×1, fDiv×1			
Reorder buffer	640	1024	2048	4096
Logical registers	STRAIGHT: Unified×127 RISC-V: Int×31, FP×32			
Physical registers	(Sufficient amount)			
Branch predictor	8-component TAGE [16], 130-bit history, 8 KiB			
Branch target predictors	Branch target buffer (BTB): 4-way, 8192 entries Return address stack (RAS): 16 entries			
Mem. dep. pred.	Store set [17], 9-bit producer ID, 4096 entries			
L1I cache	128 KiB, 8-way, 64B line, 3 cycles			
L1D cache	128 KiB, 8-way, 64B line, 3 cycles			
L2 cache	8 MiB, 16-way, 64B line, 12 cycles Stream prefetcher [18], distance 8, degree 2			
Main memory	80 cycles			

register renaming[4]. Exception recovery was assumed to be instantaneous; this is a favorable setting for RISC-V, unlike the existing study where took into account the time for ROB walking.

B. Porting musl for STRAIGHT

To compile real-world programs written in C, a C standard library is required. We chose musl libc as the C standard library; it is lightweight and almost written in C. Based on the 2c2477d commit hash, which was the latest at the time of development, we ported it to STRAIGHT from the implementation of the 64-bit version of RISC-V, which has almost the same application binary interface (ABI). The main changes made in the port are as follows:

a) Removing inline assembly instructions: Although most of musl is written in C, it contains some inline assembly instructions to perform atomic memory operations. Our STRAIGHT compiler does not support inline assembly, so we removed them and wrote the equivalent memory operations with the C language. Although it lost atomicity because of these modifications, it did not cause any problems because all the benchmarks we used were single-threaded programs.

[4]STRAIGHT has a shorter front-end pipeline because only a simple adder is needed to obtain a physical register number of a source operand [7]. The calculation is done in parallel with the other decoding processes; thus, removing the dedicated rename stage does not extend the critical path.

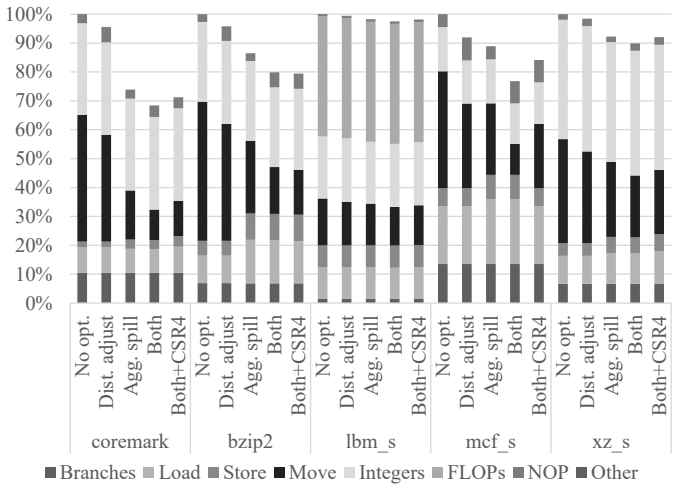

Fig. 5. Number of instructions for each instruction type. For each benchmark, the values are normalized to those of the case without optimizations.

b) Avoiding problems with quadruple-precision floating-point numbers: The implementation for RISC-V, which we used as a base for porting, assumed that the `long double` was a quadruple-precision floating-point number. This was a problem because the simulator we used could not simulate quadruple-precision floating-point operations. To address this problem, we rewrote the source code for the part where the `long double` is represented so that it works, even if it is a double-precision floating-point number. In this modification, we referred to architectures where the `long double` is a double-precision floating-point number. In addition, we excluded mathematical functions and complex functions for the `long double` from the compilation target. This modification did not affect the results because the benchmarks used in this study did not contain any code that used `long doubles`.

C. Results

1) Change in the number of instructions, owing to our proposal: Fig. 5 shows the extent to which our proposed methods reduce the number of instructions executed. For each benchmark, the values were normalized to those of the case without optimizations (using the basic compiler algorithm [7] shown in Section II-B). Instructions that were discarded because of speculative execution were not included.

In all benchmarks, except for `619.lbm_s`, our proposed optimizations significantly reduced the number of executions of `mv` instructions and contributed to a reduction in the total

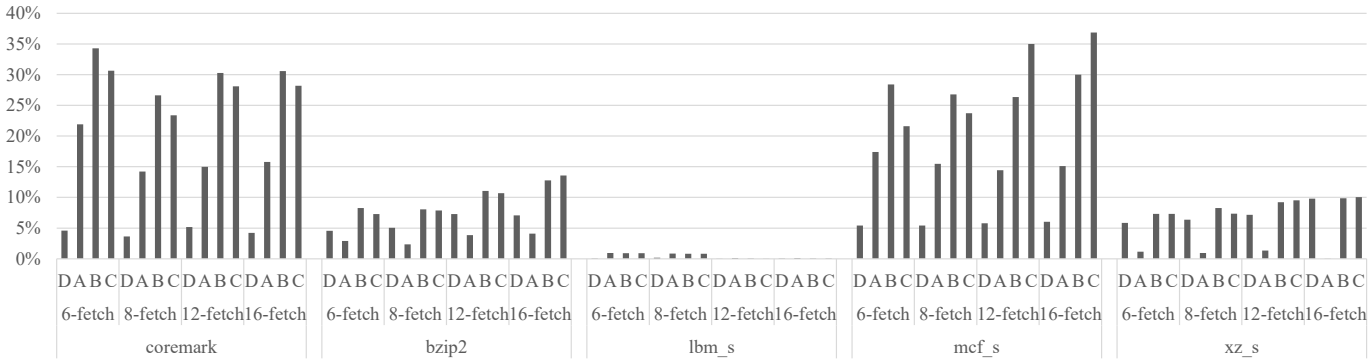

Fig. 6. Performance improvement owing to our proposal. 'D' corresponds to the distance-adjustment optimization, 'A' corresponds to the aggressive-spill optimization, 'B' corresponds to applying both optimizations, and 'C' corresponds to introducing four CSRs to 'B'. For each benchmark, the case without optimization is used as the 0% baseline.

number of executed instructions. There was almost no reduction in the number of instructions executed in `619.lbm_s` because most of the executions are executing a single large loop ($\sim$350 instructions). The loop contains few function calls or merging points, and the only non-referenced loop constant is the return address of the function containing this loop; therefore, we almost cannot reduce the `mv` instructions.

Although the aggressive-spill optimization increases the number of load/store instructions, it reduces the number of `mv` instructions by more than that, resulting in a significant reduction in the total number of instructions executed. On the other hand, the distance-adjustment optimization reduces the number of `mv` instructions without increasing the number of other instructions. Furthermore, these two optimizations are independent of each other, as they reduce completely different `mv` instructions.

The introduction of callee-saved registers has essentially increased the number of instructions because it requires the `mv` instruction to copy the value to the location defined by the calling conventions. In `401.bzip2` and `605.mcf_s`, which call many leaf functions with a small number of execution instructions, the introduction of callee-saved registers reduces the number of load/store instructions required for spilling, while the number of `mv` instructions increases. For `619.lbm_s` and `657.xz_s`, which have fewer leaf function calls, there is almost no change in the number of instructions. In `CoreMark`, where all the small functions are inline expanded, the number of instructions increased. This is because there are not enough live variables to pass as callee-saved registers, so the number of load/store instructions increases unnecessarily. This phenomenon also occurs in conventional RISC architectures. Thus, the increase in the number of load/store instructions executed by setting the callee-saved registers is not a problem unique to STRAIGHT.

2) Performance improvement, owing to our proposal: Fig. 6 shows the extent to our proposed methods improve the performance. The performance is the inverse of the number of cycles required for execution. For each benchmark, the case without optimization was used as the 0% baseline.

In all benchmarks, except for `619.lbm_s`, our proposed optimizations significantly improved the performance. Almost no performance improvement was observed for `619.lbm_s` because our optimizations hardly reduce the number of instructions to be executed. Depending on the benchmark, either the aggressive-spill optimization or the distance-adjustment optimization may contribute significantly to the performance. The two optimizations are independent of each other, not only in their effect on instruction-count reduction, but also in their contribution to performance improvement.

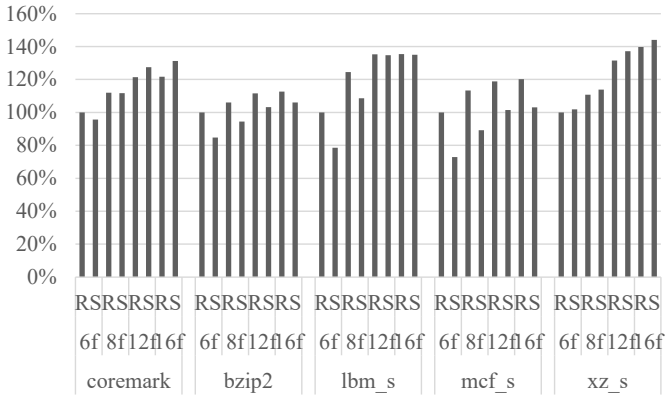

Fig. 7. Performance compared to RISC-V's `6-way`. 6f, 8f, 12f, and 16f respectively correspond to parameters `6-way`, `8-way`, `12-way`, and `16-way`. 'R' and 'S' correspond to RISC-V and STRAIGHT, respectively.

The introduction of callee-saved registers improved the performance of `401.bzip2` and `605.mcf_s`, which call many leaf functions with a small number of instructions to execute. For `619.lbm_s` and `657.xz_s`, which call fewer leaf functions, there is little performance improvement. In `CoreMark`, where all the small functions are inline expanded, the performance is even worse. Our preliminary evaluations show that `CoreMark` performs poorly on conventional RISC architectures because of the setting of the callee-saved registers; therefore, the performance degradation due to the setting of callee-saved registers is not unique to STRAIGHT.

3) Compared to RISC: Fig. 7 compares the performance of binaries compiled with our proposed optimizations and

four callee-saved registers with the performance of binaries compiled for RISC-V. The values are normalized to those of RISC-V's `6-fetch`.

In `CoreMark` and `657.xz_s`, the performance of STRAIGHT is higher than that of RISC-V. In these benchmarks, STRAIGHT has the advantage of a larger number of logical registers and faster recovery from branch misprediction, owing to the shorter front-end latency.

In `619.lbm_s`, when the fetch width is small, the performance is lower than that of RISC-V; however, when the fetch width is sufficient, the performance is equivalent to that of RISC-V. This is because STRAIGHT requires more instructions to execute the same program as RISC-V. If there is sufficient fetch width, and sufficient instructions can be supplied to the back-end, the performance will be identical, because the number of floating-point units will be the bottleneck.

For `401.bzip2` and `605.mcf_s`, the performance of RISC-V was higher than that of STRAIGHT. The main computation in these benchmarks is sorting, which involves many complex control flows with many merges and branches. Moreover, the execution order constraints are often inconsistent across multiple paths, and the addition of `mv` instructions is inevitable, even with the distance-adjustment optimization. This overhead of increasing the number of instructions is the cause of the inferior performance, compared to RISC-V.

In the geometric mean of the five benchmarks, our compiler achieved the following performance against RISC-V: 86.1% for the `6-fetch` model, 91.1% for the `8-fetch` model, 97.0% for the `12-fetch` model, and 97.8% for the `16-fetch` model. It is noteworthy that STRAIGHT, which has an increased number of instructions instead of register renaming, can achieve almost the same performance as RISC-V. The performance of STRAIGHT is within 2% of RISC-V's performance from evaluation, but it makes the 16-wide fetch much more feasible.

V. Related works

Various approaches have been proposed to reduce the memory-accesses overhead to save registers on function calls. Lang and Huguet [22] proposed a method to avoid unnecessary spills by adding a hardware mechanism to dynamically track register usage. Wall [23] and Chow [24] independently proposed an interprocedural register allocation using call graph analysis. However, many recent compilers have a fixed set of callee-saved registers because of the increased hardware cost and the difficulty of optimizing for indirect calls. The register allocation to reduce the cost of function calls is described in detail by Lueh and Gross [25]. Following this approach, we set a fixed number of callee-saved registers in the STRAIGHT compiler.

The write-once manner of register usage in STRAIGHT is similar to that of purely functional languages, so compiler techniques for these languages can be valuable in STRAIGHT. The proposed method using extra arguments for introducing callee-saved registers is similar to the approach in the continuation-passing style [26].

VI. Conclusion

STRAIGHT is an instruction set architecture that uses inter-instructional distance to specify operands so that false dependencies do not occur. Because the compiler, not the hardware, resolves false dependencies, the compiler is essential to achieve high performance. In this paper, we proposed two important optimizations and a novel calling conventions for STRAIGHT. The first was distance-adjustment optimization, which matched the inter-instructional distance in multiple execution paths by reordering the instruction order. Although the distance-adjustment optimization required solving the feedback edge set problem, it contributed to reducing the number of `mv` instructions without increasing the number of other instructions. The second was the aggressive spill optimization, which spills loop constants that are not referenced in the loop, even though the number of logical registers is not insufficient. While this optimization slightly increases the number of load/store instructions, it significantly reduces the number of `mv` instructions executed. The third is a method of setting the callee-saved registers suitable for STRAIGHT. Setting the callee-saved registers can reduce the number of load/store instructions for benchmarks that frequently call small leaf functions, although it increases the number of `mv` instructions for distance adjustment.

We implemented these optimizations and the method of setting the callee-saved registers in a STRAIGHT compiler using LLVM. We also developed a STRAIGHT toolchain to compile real-world programs. We compiled and evaluated real-world programs using the compiler. Our optimizations reduced the number of executed instructions by 18% and improved the performance by 16% on the average of five benchmarks. In addition, the introduction of callee-saved registers improved performance by up to 7% in benchmarks with many leaf function calls. Performance simulations with exactly the same scale processor revealed that the performance difference between STRAIGHT and RISC-V was only approximately 2%, despite the elimination of register renaming. Our compiler technology, which enables high-quality STRAIGHT code generation, encourages the development of highly scalable processors based on the STRAIGHT architecture.

Appendix

When there is a reference across function calls or loops in applying the aggressive-spill optimization, it is necessary to insert a restore instruction. We describe the details of the procedure to determine where to insert the restore instruction.

If a reference to a value r may cross a function call, it is necessary to insert a restore instruction. The restore instruction should be inserted immediately before an instruction referencing the value r that exists in a region where the value r may not be in the registers. To find the region, consider whether the value r may not be in the registers after the instruction is completed. After completing the producer instruction of

r, the value r is clearly in the registers. After completing a consumer instruction of r, the value r is also in the registers because it was referenced as an operand in the STRAIGHT's distance form. After completing a function call, the value r is not in the registers. Elsewhere, if the value r may not be in the registers after completing an instruction, it may not be in the register after completing the next instruction. By finding the least fixed-point set for the above relationship, we can determine whether there is a possibility that the value r is not in the register at the location where a particular instruction is completed.

To describe this formally, we introduce the notation of modal mu-calculus [27], [28]. To describe the algorithms, we define the following:

- U_r: Set of all instructions in the living section of r. In the following, this set is considered as the universal set.
- $\mathrm{Producer}_r$: Proposition that this instruction is the producer instruction of r.
- $\mathrm{Consumer}_r$: Proposition that this instruction is a consumer instruction of r.
- Call: Proposition that this instruction is a function-call instruction.

The least fixed-point set described above can be written as the following A_r:

$$A_r = [\![\mu X. \neg\mathrm{Producer}_r \wedge \neg\mathrm{Consumer}_r \wedge (\mathrm{Call} \vee \overline{\langle 1 \rangle} X)]\!].$$

The code obtained by solving this data-flow equation passes values through registers in any case where the values can be passed through registers.

When applying aggressive-spill optimization, we can use the greatest fixed-point set B_r, instead of the least fixed-point set A_r:

$$B_r = [\![\nu X. \neg\mathrm{Producer}_r \wedge \neg\mathrm{Consumer}_r \wedge (\mathrm{Call} \vee \overline{\langle 1 \rangle} X)]\!].$$

The code obtained by solving this data-flow equation passes values through registers, only if the passing path does not contain loops in which the value is not referenced.

REFERENCES

[1] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM POWER9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017.

[2] D. Kanter, "Intel's Sunny Cove sits on an icy lake," *Microprocessor Report*, pp. 1–4, 2019.

[3] D. Suggs and D. Bouvier, "The path to "Zen 2"," https://www.slideshare.net/AMD/the-path-to-zen-2, 2019.

[4] L. Gwennap, "Cortex-A77 improves IPC," *Microprocessor Report*, pp. 1–4, 2019.

[5] R. Shioya, K. Horio, M. Goshima, and S. Sakai, "Register cache system not for latency reduction purpose," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 301–312.

[6] R. Shioya and H. Ando, "Energy efficiency improvement of renamed trace cache through the reduction of dependent path length," in *2014 IEEE 32nd International Conference on Computer Design*, 2014, pp. 416–423.

[7] H. Irie, T. Koizumi, A. Fukuda, S. Akaki, S. Nakae, Y. Bessho, R. Shioya, T. Notsu, K. Yoda, T. Ishihara, and S. Sakai, "STRAIGHT: Hazardless processor architecture without register renaming," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 121–133.

[8] T. Koizumi, S. Nakae, A. Fukuda, H. Irie, and S. Sakai, "Reduction of instruction increase overhead by STRAIGHT compiler," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, 2018, pp. 92–98.

[9] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[10] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1988, pp. 12–27.

[11] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1988, pp. 1–11.

[12] "CoreMark," https://www.eembc.org/coremark/.

[13] "Standard performance evaluation corporation cpu2006 benchmark suite," http://www.spec.org/cpu2006/.

[14] "Standard performance evaluation corporation cpu2017 benchmark suite," http://www.spec.org/cpu2017/.

[15] "musl libc," https://musl.libc.org.

[16] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-Level Parallelism*, vol. 8, no. 2, pp. 1–23, 2006.

[17] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Annual International Symposium on Computer Architecture*, 1998, pp. 142–153.

[18] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 63–74.

[19] "RISC-V: The free and open risc instruction set architecture," https://riscv.org/.

[20] "Processor simulator onikiri 2," https://github.com/onikiri/onikiri2.

[21] A. Jani, "Apple ships its first PC processor," *Microprocessor Report*, pp. 1–5, 2021.

[22] T. Lang and M. Huguet, "Reduced register saving/restoring in single-window register files," *ACM SIGARCH Computer Architecture News*, vol. 14, no. 3, pp. 17–26, 1986.

[23] D. W. Wall, "Global register allocation at link time," *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 264–275, 1986.

[24] F. C. Chow, "Minimizing register usage penalty at procedure calls," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, 1988, pp. 85–94.

[25] G.-Y. Lueh and T. Gross, "Call-cost directed register allocation," *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 296–307, 1997.

[26] A. W. Appel and Z. Shao, "Callee-save registers in continuation-passing style," *LISP and Symbolic Computation*, vol. 5, pp. 191–221, 1992.

[27] D. Kozen, "Results on the propositional mu-calculus," *Theoretical Computer Science*, vol. 27, no. 3, pp. 333–354, 1983.

[28] B. Steffen, "Data flow analysis as model checking," in *International Symposium on Theoretical Aspects of Computer Software*, 1991, pp. 346–364.