A Sound and Complete Algorithm for Code Generation in Distance-Based ISA

Shu Sugita
The University of Tokyo
Tokyo, Japan
sugita@mtl.t.u-tokyo.ac.jp

Toru Koizumi
The University of Tokyo
Tokyo, Japan
koizumi@mtl.t.u-tokyo.ac.jp

Ryota Shioya
The University of Tokyo
Tokyo, Japan
shioya@ci.i.u-tokyo.ac.jp

Hidetsugu Irie
The University of Tokyo
Tokyo, Japan
irie@mtl.t.u-tokyo.ac.jp

Shuichi Sakai
The University of Tokyo
Tokyo, Japan
sakai@mtl.t.u-tokyo.ac.jp

Abstract

The single-thread performance of a processor core is essential even in the multicore era. However, increasing the processing width of a core to improve the single-thread performance leads to a super-linear increase in power consumption. To overcome this power consumption issue, an instruction set architecture for general-purpose processors, called STRAIGHT, has been proposed. STRAIGHT adopts a distance-based ISA, in which source operands are specified by the distance between instructions. In STRAIGHT, it is necessary to satisfy constraints on the distance used as operands to generate executable code. However, it is not yet clear how to generate code that satisfies these constraints in the general case. In this paper, we propose three compiling techniques for STRAIGHT code generation and prove that our techniques can reliably generate code that satisfies the distance constraints. We implemented the proposed method on a compiler and evaluated benchmark programs compiled with it through simulation. The evaluation results showed that the proposed method works in all cases, including conditions where the number of registers is small and existing methods fail to generate code.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Superscalar architectures.*

Keywords: compiler, code generation, instruction set architecture, instruction-level parallelism

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02…$15.00
https://doi.org/10.1145/3578360.3580263

ACM Reference Format:
Shu Sugita, Toru Koizumi, Ryota Shioya, Hidetsugu Irie, and Shuichi Sakai. 2023. A Sound and Complete Algorithm for Code Generation in Distance-Based ISA. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578360.3580263

1 Introduction

The single-thread performance for executing sequential portions of a program is a crucial factor for a processor core, even in the multicore era [19, 21]. This is because the speedup from a parallel execution is limited by sequential portions of the program, as Amdahl's law suggests. As a result, modern processors extract more instruction-level parallelism to improve their single-thread performance by increasing the processing width of a core and the size of the scheduling window [12, 31].

However, increasing the processing width can lead to a super-linear increase in power consumption. One of the units that cause such an increase is a renaming unit. This unit has a table called a register map table (RMT) to remove false dependencies between instructions. The RMT needs the circuit area and power consumption in proportion to the square of the processing width. Thus, the RMT causes the super-linear increase in circuit area and power consumption [32, 33].

To address this power consumption issue, an instruction set architecture (ISA) that does not require register renaming, called STRAIGHT, has been proposed [23]. STRAIGHT has a register file like a ring buffer in which the results of each instruction are sequentially written. Each instruction refers to the source operands by the distance between instructions, as shown in Figure 1. This value used to specify operands is called *reference distance* or simply *distance*. Owing to this instruction format, STRAIGHT has no false dependencies even without register renaming. As a result, STRAIGHT processors do not require a renaming unit, which increases power consumption super-linearly with processing width, thus allowing for larger processing width than conventional RISC processors [28].

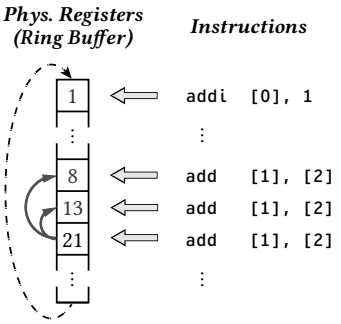


Figure 1. Register usage and assembly codes in STRAIGHT architecture. In assembly representation, square brackets are used to indicate inter-instruction distances (`[0]` is zero register). Each destination register is allocated consecutively as a ring buffer.

STRAIGHT compilers must generate code that satisfies distance-related constraints, but an algorithm that can generate such codes in general cases is not yet clear. Since STRAIGHT uses the distance to specify source operands, STRAIGHT code has two distance constraints: 1) The reference distance is statically determined regardless of the execution path. 2) The reference distance must be less than an upper limit specified in the ISA specification. Existing STRAIGHT compilers resolve these constraints by using an ad hoc algorithm on the premise that there are sufficient registers [23, 26]. However, we found that without this premise, these compilers cannot generate executable code that correctly satisfies the constraints.

In this paper, we propose a code generation algorithm for STRAIGHT that resolves the distance constraints in the general case. Our proposed algorithm consists of the following three processes. (1) The first is a novel spilling technique. Our technique appropriately limits the number of live variables and ensures that the subsequent processes can generate code that satisfies the distance constraints without additional spill code. (2) The second is a novel technique in SSA destruction, which converts a program from an intermediate representation to a distance-based form. Our SSA destruction technique reliably and efficiently replaces ϕ-functions with move instructions by combining existing methods. (3) The third is a novel technique in move insertion, which resolves the distance constraints. This process generally generates codes that satisfy distance constraints with fewer move instructions.

We implemented the proposed algorithm on an existing STRAIGHT compiler [4] and evaluated benchmark programs compiled with it through simulation. The evaluation results show that the proposed method can generate code even under conditions that the existing compiler cannot. Moreover, the results show that our techniques reduce the number of move instructions for satisfying the distance constraints.

2 STRAIGHT Architecture

2.1 STRAIGHT Instruction Set Architecture

STRAIGHT is an ISA that specifies the source operands by the distance between instructions, not by the logical register numbers [22, 23]. In STRAIGHT, each instruction specifies its source operand by pointing to the producer instruction, as "use the result of the n-th previous instruction." This value n used to specify an operand is called *distance*. The distance must be less than an upper limit specified in the ISA specification. This upper limit is called *maximum reference distance*, which is an architecture-specific parameter of STRAIGHT.

Utilizing the maximum reference distance, STRAIGHT removes register renaming, which causes a super-linear increase in power consumption with a processing width of a processor core. Due to the constraint on the maximum reference distance, registers are no longer referenced after a certain time in STRAIGHT. STRAIGHT has a ring buffer, where the result of instructions are written sequentially to these unreferenced registers (Figure 1). Since STRAIGHT assures to overwrite only out-of-life registers, it does not cause any false dependencies between instructions. The position of the source operand on the ring buffer can be determined by simply adding the distance to its head point, eliminating the complex logic required for conventional register renaming.

The instruction representation of STRAIGHT is similar to the static single-assignment (SSA) form [5, 16, 30] in that there is no overwriting. The SSA form is a program representation in which all variables are assigned precisely once. Because this representation is compatible with the write-once manner of register usage in STRAIGHT, STRAIGHT compilers [4, 23, 26] uses the SSA form as an intermediate representation.

2.2 Distance Constraints in Code Generation

In STRAIGHT, it is necessary to use different code generation techniques than in conventional RISCs due to the distance-related constraints. Conventional RISCs use a logical register number to specify the source operands. On the other hand, STRAIGHT uses a distance between instructions, which is a relative value. This value can be easily changed when inserting or relocating instructions to satisfy the constraint. For this reason, it is challenging to determine the placement of each instruction individually.

A STRAIGHT compiler needs to generate machine codes to satisfy the following two distance constraints:

1. The source operands of each instruction must exist at the same distance for all execution paths.
2. The distance of each source operand must be no longer than the maximum reference distance.

To resolve these distance constraints, a STRAIGHT compiler inserts move instructions to copy variables and `nop` instructions to adjust distances. These instructions, which are not directly related to the execution of the program, usually

do not affect performance. However, an excessive insertion of them may have a negative impact. Therefore, a STRAIGHT compiler aims to convert programs to the STRAIGHT form with as few additional instructions as possible. We describe existing methods for resolving distance constraints in Section 3.

2.3 Distance-Based Calling Convention

STRAIGHT has distance-based calling conventions. In conventional RISCs, arguments and return values are passed between functions by placing them on specific logical registers. However, STRAIGHT cannot use similar conventions due to the lack of named general-purpose registers.

STRAIGHT uses a relative distance from a function call and return (`jal` and `ret`) to pass arguments and return values. Concretely, STRAIGHT uses a calling convention where the previous instruction of a function call produces the first argument, the second previous instruction produces the second argument, and so on. The producers of return values are also placed similarly to those of arguments.

Saving contexts across a function call also needs a distance-based convention for STRAIGHT. Since a distance across a function call is generally undetermined, it is impossible to reference a variable across a function call by distance. Consequently, all general-purpose registers in STRAIGHT are caller-preserved. However, the lack of callee-saved registers leads to performance degradation in frequent calls of leaf procedures. For this reason, a distance-based callee-save convention is proposed [26]. This method adds additional arguments and returns values to each function, which are used as callee-saved values.

3 Challenges of SSA-Based Compiler for STRAIGHT

In constructing a compiler for STRAIGHT ISA, there are the following three challenges:

1. Because the condition under which all variables can be assigned to available registers without spilling is more complex than in conventional RISC, there is no clear policy for the insertion of spill code.
2. There is no efficient and certain technique for matching reference distances of different execution paths within the maximum reference distance.
3. Even when the condition that a program can be translated into STRAIGHT code without further spilling is met, it is difficult to keep all the reference distances within the maximum reference distance.

These challenges are related to each of the three compiling processes in STRAIGHT: spill code generation, SSA destruction, and move insertion. This section summarizes existing compiler techniques and their limitations that we found for each process.

3.1 Spill Code Generation

Spill code generation is the process of spilling variables so that all live variables can be assigned to available registers [8, 10, 13]. In this process, an excessive insertion of spill codes diminishes the execution speed. Thus, spill code generation aims to minimize the execution of spill codes while ensuring that all live variables can be assigned to available registers without further spilling.

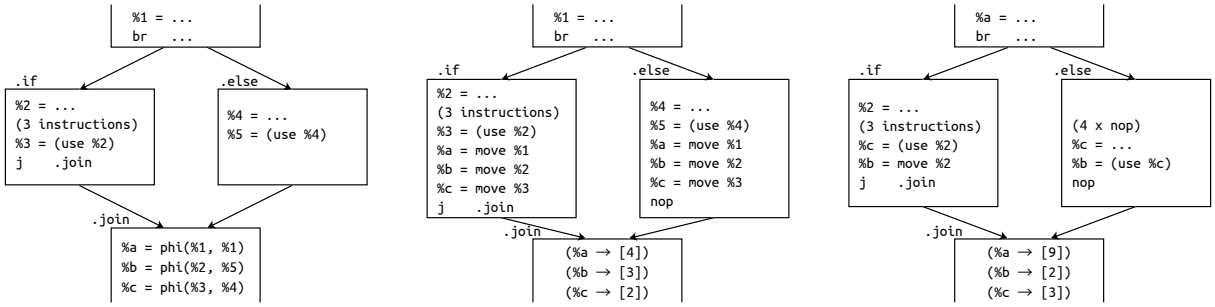
However, minimizing the number of spill codes is a challenging task for a STRAIGHT compiler. This is because the condition in which all live variables can be assigned is more intricate in STRAIGHT than in conventional RISCs. In conventional RISCs, the number of allocatable variables corresponds exactly to the number of logical registers because it is sufficient for each variable to exist in one register. On the contrary, in STRAIGHT, the number of allocatable variables does not always correspond to the maximum reference distance. This is for two reasons: (1) A single value often exists simultaneously at multiple referable distances to satisfy the distance constraints. (2) A store instruction consumes a register and increases other reference distances. Thus, more registers can be consumed than the number of live variables at each program point. As a result, we cannot easily check whether a program needs spill code even if the number of variables is less than that of referable registers at every program point. In fact, we found that the existing STRAIGHT compiler [4] did not spill; thus, it failed to compile a program when it needed spill code to resolve the distance constraints.

3.2 SSA Destruction

SSA destruction is the process of converting a program from the SSA form to an executable form by eliminating ϕ-functions [9, 18]. The ϕ-function is a statement used to merge different variables of the predecessor blocks. Since ϕ-functions are not implemented as machine code, a compiler needs to replace them with move instructions during SSA destruction.

In STRAIGHT, ϕ-functions are used as markers to merge references, whose distances differ with the execution path. A conventional RISC compiler eliminates ϕ-functions by assigning different variables to the same register. In contrast, a STRAIGHT compiler eliminates ϕ-functions by placing the variables at the same distance from the join block. The reference distance may differ for different execution paths, even when referencing the same variable. For this reason, the STRAIGHT compiler adds ϕ-functions for all live variables to each join block before code generation as shown in Figure 2 (a).

Distance fixing [23] is a naïve method of SSA destruction in STRAIGHT. This method eliminates ϕ-functions to copy live variables in a common order at the end of each predecessor block (Figure 2 (b)). The region where these copy instructions exist is called a fixed region, where the positions



(a) SSA program **(b)** Result by the naïve method **(c)** Result by the reordering method

Figure 2. SSA destruction in STRAIGHT. In this figure, square brackets are used to indicate distances from the join point.

of instructions are fixed in subsequent processes. Thus, this process ensures that each variable can be referenced by a static distance regardless of an execution path.

Because the distance fixing inserts many redundant instructions, an optimizing technique by reordering instructions is proposed [26]. This technique is hereafter called the reordering method. The reordering method determines the positions of inserting move instructions to consider dependencies between variables. It minimizes the number of move instructions by solving a feedback arc set problem [35]. Meanwhile, it has a disadvantage that the maximum reference distance constraint may not be met because each instruction is adjusted to a distant position by inserting nop instructions (Figure 2 (c)).

However, these two existing methods do not balance certainty and efficiency. The naïve method can reliably translate from SSA form but requires many redundant move instructions. On the contrary, the reordering method is more efficient but less certain since it cannot generate executable code in case the reference distance of a ϕ-function exceeds the maximum reference distance.

3.3 Move Insertion

Move insertion is the process of splitting a live range of each variable [15]. In STRAIGHT, instructions that can reference a variable are limited to be within a maximum reference distance from that variable. Thus, splitting a wide live range is directly related to the maximum reference distance constraint. For this reason, move insertion is an essential process in STRAIGHT to generate executable code.

Distance bounding [23] is the only move insertion technique that has ever been proposed. This method inserts move instructions to split distances in the following way:

Step 1) Find a reference whose distance exceeds the maximum reference distance. If not found, end the process.

Step 2) Add a move instruction at the maximum possible distance from the consumer for relaying the reference found in Step 1. Then, go back to Step 1.

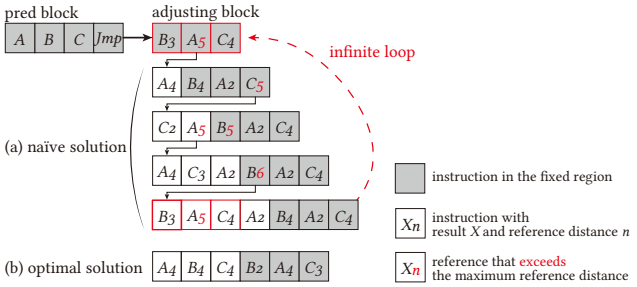


Figure 3. Example of move insertion where the maximum reference distance is 4. (a) a naïve solution that causes an infinite loop (b) an optimal solution that results in 3 move instructions

However, the distance bounding may insert many redundant instructions. Moreover, we found that it does not work in the worst cases, as shown in Figure 3. The inserted move instruction extends the distance of references across it. Hence, it is challenging to satisfy the constraint on the maximum reference distance for all instructions in an ad hoc manner.

4 Proposed Method

4.1 Overview

To address the challenges described in Section 3, we propose a novel compiler algorithm comprising of the following three techniques:

1. Spilling technique that ensures that there exists a way to insert moves to generate executable codes.
2. SSA destruction technique that minimizes the number of move instructions at various maximum reference distances
3. Move insertion technique that ensures to resolve the maximum reference distance constraint after applying our spilling technique

These processes are executed in the compilation flow shown in Algorithm 1.

Algorithm 1: Proposed Compilation Flow

```
Definition : f is a function in SSA form
             l is max referable length
Require    : f has no loop (for simplicity).
1  Function StraightCodeGen(f, l):
2      SpillCodeGen(f, l)                          ▶ Section 4.2
3      foreach join point p in the execution order of f do
4          SSADestruction(p, l)                    ▶ Section 4.3
5          foreach unfixed path u before p do
6              MoveInsertion(u, l)                 ▶ Section 4.4
7              u.setFixed()
8      foreach unfixed path u do
9          MoveInsertion(u, l)                     // for calling conv.
10 Function MoveInsertion(u, l):
11     RelayingVariable(u, l)                      ▶ Section 4.4.1
12     SortingVariable(u, l)                       ▶ Section 4.4.2
```

This division of the compilation processes in STRAIGHT is general to distance-based ISAs, not specific to STRAIGHT ISA. This is for the following two reasons:

1. Since inserting spill code affects other reference distances, it is difficult to perform spilling simultaneously with the other two processes.
2. The maximum reference distance constraint can only be resolved once the reference distance is statically determined by SSA destruction.

In the case of a distance-based ISA where store instructions do not affect other reference distances, the spilling process could be executed concurrently with the other processes. However, such a distance-based ISA requires a more complex hardware implementation than STRAIGHT ISA.

Our proposed algorithm uses SSA form as the compiler representation because this representation is compatible with the write-once manner of register usage in STRAIGHT. Our techniques can be applied to STRAIGHT compilers that use other compiler representations, but a process of conversion to a single-assignment representation is additionally required for code generation.

4.2 Spill Code Generation

The existing STRAIGHT compilers cannot generate code when its input program cannot satisfy the distance constraints without spill codes. In such cases, the compiler needs to spill variables to satisfy the constraints.

To deal with this situation, we propose a spilling technique for STRAIGHT to satisfy the distance constraints. First, we consider a condition for satisfying the distance constraints without spill codes. Then, we propose a spilling technique to minimize spill codes using the condition that we obtain.

Before a detailed explanation, we introduce some terms and notations that help to describe our method.

- The maximum reference distance is denoted by l.
- The two terms *variable* and *place* are used as follows.
  - A *variable* represents the unique value in an SSA-form program.
  - A *place* represents the unique position of a register occupied by an instruction in a distance-based form.
- The sequence of results of instructions within the maximum reference distance is called a *context*.
  - The value located at the oldest place within a context is designated as *oldest value*.
  - Within a context, a value that is not subsequently referenced is called a *dead value*, and the place of a dead value is called a *dead place*[1].
- A program is said to be *assignable* if it can be translated into STRAIGHT form by inserting only move instructions.

To make a program assignable, it must satisfy the following two properties:

1. At each program point, all live variables are still referable when the next instruction is executed.
2. At each join point, the place of each live value can be matched in every predecessor block.

To show the conditions to satisfy the above properties, we first prove Lemma 1.

Lemma 1. *If at least one dead place exists in the context, live values can be arbitrarily reordered by inserting only move instructions.*

Proof. With a dead place e, any two live values x, y can be swapped with $2l$ move instructions in the following process.

1. Let x be the older value among x and y.
2. Insert first l move instructions:
   - If e is the oldest place, insert a move instruction to copy x. The result of the inserted instruction is denoted by x_e.
   - Otherwise, insert move instructions to copy the value at the oldest place.
3. Insert last l move instructions:
   - If x is located at the oldest place, insert a move instruction to copy y.
   - If y is located at the oldest place, insert a move instruction to copy x_e.
   - Otherwise, insert move instructions to copy the value at the oldest place.

Repeating this swap procedure enables reordering live values in an arbitrary order. □

From Lemma 1, we obtain Theorem 2, which shows the sufficient condition to make a program assignable.

[1]Note that the dead place is not always synonymous with the place of a dead variable. This is because when a live variable has multiple occurrences within a context, only the most recent value will be referenced by subsequent instructions.

Theorem 2. *A program is assignable if the number of live variables is* less than l *at any program point.*

Proof. When the number of live variables is less than l at every program point, at least one dead place exists in context. Hence, using Lemma 1, we can show the following:

1. All live variables are still referable when the next instruction is executed by reordering the context so that the oldest place is a dead place.
2. The position of each live value can be matched in every predecessor block by reordering the corresponding variables in a common order.

However, a program with at most n live variables is not always assignable. When there are only n ϕ-functions at a join block, the predecessors of the join block are not assignable because a jump instruction occupies a register. □

Using Theorem 2, we can show that spill code generation of distance-based ISA can be implemented with conventional allocation techniques [10, 13, 18, 29, 34] by limiting the number of live variables to $l - 1$ at any program point. Although inserting a load/store instruction might increase the reference distance of following instructions, it does not affect the condition of Theorem 2 because the place occupied by that instruction becomes dead immediately after its execution.

4.3 SSA Destruction

Two SSA destruction methods are proposed for STRAIGHT compilers, as described in Section 3.2. However, there is no method that balances certainty and code quality. The naïve method can reliably eliminate ϕ-functions but often inserts redundant move instructions. The reordering method inserts fewer move instructions but does not reliably generate STRAIGHT codes.

Our proposed technique is a technique that combines two existing methods by incrementally changing which method to use for each ϕ-function. This technique provides reliable and efficient SSA destruction for any conditions of maximum reference distance.

Our technique decides which of the existing methods to apply for each ϕ-function in the following way:

Step 1) Let R be the set of all ϕ-functions and let S be an empty set.
Step 2) Apply the reordering method to R and the naïve method to S.
Step 3) If there are no ϕ-functions whose reference distance exceeds the maximum reference distance, end the process.
Step 4) Otherwise, remove the ϕ-function that has the most distant references from R and add it to S. After that, restore the order of instructions and go to Step 2.

Even in the worst case, this algorithm ensures obtaining the same result as the naïve method. Thus, the certainty of SSA destruction is guaranteed. Moreover, our method can reduce the number of move instructions by applying the reordering method as much as possible.

4.4 Move Insertion

STRAIGHT compiler inserts move instructions to satisfy distance constraints. However, existing methods are ad hoc and thus insert many redundant move instructions. Furthermore, code generation fails in the worst case due to a failure to satisfy the constraints. This is because adding move instructions to resolve the constraint may prevent satisfying the constraint as a whole by extending the distance of other references.

Our proposed method generates efficient code in the general case through two processes: *relaying variables* and *sorting variables*. This two-phase method is based on our observation that the purpose of move insertion can be divided into two:

1. For relaying variables that cannot be referenced directly due to their long live range.
2. For sorting variables to a specific order that is determined by the result of SSA destruction or calling conventions.

We describe the method of relaying variables in Section 4.4.1 and sorting variables in Section 4.4.2.

4.4.1 Relaying Variables. Relaying variables is the process of inserting move instructions to relay a variable with a long live range due to the constraint on the maximum reference distance. The insert position of such relay instructions should be as far from the producer as possible. This is because the live range can be covered with fewer instructions if the reference distance of relaying instructions is longer.

Our proposed method inserts move instructions so that the variable is always relayed at the maximum distance. The detailed algorithm for relaying variables is as follows.

- For each instruction I in execution order
  1. Insert a move instruction of the oldest variable immediately before I until the oldest variable is dead.
  2. Fix the position of the instruction I.

Since each inserted move instruction always targets the oldest variable, it references at the maximum reference distance. Therefore, this method is optimal in the number of additional instructions to relay variables.

4.4.2 Sorting Variables. Sorting variables is the process of inserting move instructions so that live variables match a specific order at the end of the path. This order is defined by the result of SSA destruction or calling conventions. In this case, it is sometimes more efficient to insert move instructions whose reference distance is not the maximum reference distance. For this reason, the existing method often inserts many redundant instructions.

We design an algorithm that can reliably sort variables with an optimal number of instructions in the general case.

Before going into detail, we first set up the problem to solve. We introduce the following definitions:

- The integer n denotes the number of variables to sort.
  - The i-th variable is denoted by v_i.
- Define the position of variables τ *before* sorting as follows.
  - Let $a_1, \ldots, a_l$ be the context before sorting.
  - Define τ by setting $\tau(i) = \arg\max_j \{v_i = a_j\}$[2].
- Define the position of variables π *after* sorting as follows.
  - Let $b_1, \ldots, b_m$ be the sequence of variables to be satisfied after sorting, where $n \le m < l$.
  - Define π by setting $\pi(i) = \arg\min_j \{v_i = b_j\}$[3].

First, we consider a lower bound on the number of move instructions required for sorting. When a consumer refers to a producer at a distance x, at least $\lceil x/l \rceil - 1$ move instructions are required between them. Now, suppose that sorting is completed by insertion of d move instructions. Thus, the distance between the producer and the consumer of x_i is $\pi(i) - \tau(i) + l + d$. Let $f(\tau, \pi, d)$ be the summation of the minimum number of required move instructions for all variables:

$$f(\tau, \pi, d) = \sum_i \left\lceil \frac{\pi(i) - \tau(i) + d}{l} \right\rceil. \tag{1}$$

If $f(\tau, \pi, d) > d$, sorting by inserting d move instructions is obviously impossible. Now, we call d^* the smallest non-negative integer d satisfying $f(\tau, \pi, d) \le d$. The integer d^* is a lower bound for the number of move instructions required for sorting.

Algorithm 2 shows our proposed algorithm for sorting variables with d^* move instructions. To show that this algorithm always works, we prove Theorem 6 through Lemmas 3, 4, and 5.

Lemma 3. *Let d^* be the minimum non-negative integer d satisfying $f(\tau, \pi, d) \le d$. Then*

$$f(\tau, \pi, d^*) = d^*. \tag{2}$$

Proof. Let $g(d) = f(\tau, \pi, d) - d$. $g(d)$ has three properties:

1. $g(0) \ge 0$.
2. For any non-negative integer d, $g(d+l) - g(d) = n - l < 0$.
3. For any non-negative integer d, $g(d+1) - g(d) = -1$ if $g(d+1) - g(d) < 0$.

Using these properties, we can obtain the conclusion. □

Lemma 4. *If $d^* > 0$, then there exists an i in $\{1, 2, \ldots, n\}$ such that*

$$\left\lceil \frac{\pi(i) - \tau(i) + l + d^*}{l} \right\rceil \neq \left\lceil \frac{\pi(i) + d^* - 1}{l} \right\rceil. \tag{3}$$

[2]Choose the largest index because it is optimal to reference the closest instruction among those that have the same variable.

[3]Choose the smallest index because the later instructions can copy from the preceded instruction that produces the same variable.

Algorithm 2: Move insertion for sorting variable

Definition : u is the path to insert move instructions
l is the max referable length

1 **Function** `SortingVariable`(u, l):
2 $\quad \tau \leftarrow$ the position of variables before sorting
3 $\quad \pi \leftarrow$ the position of variables after sorting
4 $\quad d^* \leftarrow$ minimum non-negative integer d satisfied $f(\tau, \pi, d) \le d$
5 $\quad$ `RecursiveInsertion`(u, τ, π, d^*)
6 **Function** `RecursiveInsertion`(u, τ, π, d^*):
7 $\quad$ **if** $d^* > 0$ **then**
8 $\quad\quad S \leftarrow \left\{ i \,\middle|\, \left\lceil \frac{\pi(i) - \tau(i) + d^* + l}{l} \right\rceil \neq \left\lceil \frac{\pi(i) + d^* - 1}{l} \right\rceil \right\}$ $(\neq \emptyset)$
9 $\quad\quad j \leftarrow \arg\min_{i \in S} \{\tau(i)\}$
10 $\quad\quad$ Insert a `move` of x_j after the end of u
11 $\quad\quad$ Update the context τ to τ' by
$$\tau'(i) \leftarrow \begin{cases} l & (i = j) \\ \tau(i) - 1 \ (> 1) & (i \neq j) \end{cases}$$
12 $\quad\quad$ `RecursiveInsertion`($u, \tau', \pi, d^* - 1$)

Proof. Suppose that there does not exist an i satisfying equation 3. Hence,

$$f(\tau, \pi, d^*) = \sum_i \left\{ \left\lceil \frac{\pi(i) + d^* - 1}{l} \right\rceil - 1 \right\}. \tag{4}$$

Now, define r by $r = d^* \bmod l$. In the case where $r = 0$, $f(\tau, \pi, d^*) < d^*$. This contradicts the result of Lemma 3.

Next, consider the case in which $r \neq 0$. Let T be the set

$$T = \left\{ i \,\middle|\, \left\lceil \frac{\pi(i) + d^* - 1}{l} \right\rceil \neq \left\lceil \frac{d^*}{l} \right\rceil \right\}. \tag{5}$$

Then, by using $|T| < r \le d^*$, we see that

$$f(\tau, \pi, |T|) \le f(\tau, \pi, r) = |T|. \tag{6}$$

This contradicts that d^* is the minimum non-negative integer d satisfying $f(\tau, \pi, d) \le d$. □

Lemma 5. *In Algorithm 2, `RecursiveInsertion`(u, τ, π, d^*) is always called with the condition that d^* is the minimum non-negative integer d satisfying $f(\tau, \pi, d) \le d$.*

Proof. It clearly satisfies the condition when it is first called; thus, we prove that $d^* - 1$ is the minimum non-negative integer d satisfying $f(\tau', \pi, d) \le d$ in a recursive call.

Consider the difference of $f(\tau, \pi, d^*)$ and $f(\tau', \pi, d^* - 1)$. Since $j \in S$ (lines 8–9 in Algorithm 2),

$$\begin{aligned} &f(\tau, \pi, d^*) - f(\tau', \pi, d^* - 1) \\ &\quad = \left\lceil \frac{\pi(j) - \tau(j) + d^* + l}{l} \right\rceil - \left\lceil \frac{\pi(j) + d^* - l}{l} \right\rceil = 1. \end{aligned} \tag{7}$$

Combining this with $f(\tau, \pi, d^*) \le d^*$, we have $f(\tau', \pi, d^* - 1) \le d^* - 1$. Therefore, $d^* - 1$ is clearly the minimum non-negative integer that satisfies the condition. □

Theorem 6. *Algorithm 2 is (1) sound, (2) complete, and (3) optimal for the number of inserted move instructions.*

Proof. (1) The termination condition of the recursive call $d^* = 0$ means that no additional instructions are required for sorting. Therefore, the result of this algorithm satisfies the constraint on the maximum reference distance. (2) Lemma 4 and 5 together show that this algorithm always works. (3) This sorting algorithm is optimal for the number of instructions since d^* is the tight lower bound. □

5 Evaluation

5.1 Methodology

We implemented the three proposed compiler techniques on the existing compiler [4]. First, we evaluated that our proposed spilling technique allows our compiler to generate executable code in all cases by measuring the change in the number of instructions. Then, we evaluated each of the other proposed techniques by measuring the following values.

1. Reduction rate of move instructions by our proposed SSA destruction
2. Reduction rate of move instructions by our proposed move insertion

The evaluation was performed by measuring the number of instructions required to execute benchmark programs compiled with our compiler through simulations. The benchmark programs used for the evaluation were CoreMark [1], 401.bzip2 from SPEC CPU 2006 [20], and 605.mcf_s, 619.lbm_s, 657.xz_s from SPEC CPU 2017 [11]. These benchmarks are the same as those used in [26] and all written in C language.

We used a STRAIGHT simulation environment [4] available on GitHub for the simulations. This environment includes a basic compiler, a C library, and a simulator for STRAIGHT. The basic compiler is a STRAIGHT compiler implementation on the version 12 of LLVM [27]. The C library is a STRAIGHT port of musl libc [2], which is a lightweight libc implementation. The simulator is based on a cycle-accurate simulator, Onikiri2 [3].

5.2 Results

5.2.1 Change in Number of Instructions for Different Maximum Reference Distances. Figure 4 shows the change in the number of executed instructions for the maximum reference distance when using the proposed method. This figure plots the total number of instructions and the number of move/load/store instructions which varies mainly with the maximum reference distance. The vertical dashed lines indicate the lower bound[4] on the maximum reference distance where the proposed method can compile each benchmark. This line is used similarly in the other figures.

We obtained the execution results under all the conditions that each benchmark can be theoretically compiled. Our proposed method inserts only the minimum amount of spill codes required to generate code. That is to say, an increase in load or store instructions indicates that it is the result under a condition that the existing compiler cannot compile. Hence, the proposed method can generate code even under conditions that the existing methods cannot.

We can see in this figure that the peak of the line about move instructions is not at the shortest maximum reference distance. The more spills, the fewer variables have a live range across a join point. This results in a decrease in move instructions caused by SSA destruction. For this reason, the peak point is not always an endpoint.

5.2.2 Reduction Rate of Move Instructions by Our Proposed SSA Destruction. Figure 5 shows the reduction rate in move instructions when using our proposed SSA destruction compared to when using the naïve SSA destruction. Our proposed method did not fail in SSA destruction, and we obtained the result with various maximum reference distances.

In the evaluation of this paper, the upper limit of the maximum reference distance was set to 127 due to the limitation of the instruction word length. However, Figure 5 implies that the performance may be further improved by using a larger value to the maximum reference distance because an increasing trend was seen even at around 127 for 657.xz_s.

For all benchmarks except CoreMark, as the maximum reference distance increased, the reduction rate of the executed move instructions decreased in some portions. As the maximum reference distance increases, the reduction rate seems to continue to increase because the number of ϕ-functions to which the reordering method can be applied increases. However, the evaluation result implies that is not always true. This can be explained by the fact that changes in the order of instructions by the reordering at a join point method may adversely affect the efficiency in SSA destruction at other join points.

For 619.lbm_s, our proposed method could not reduce the number of executed move instructions. This was due to the characteristics of 619.lbm_s. The function mainly executed in this benchmark, `LBM_performStreamCollideTRT`, has a large number of instructions within each basic block. For this reason, the reordering method failed to adjust the position of each ϕ-function to within the maximum reference distance. Thus, the result obtained by our proposed method was the same as those obtained by the naïve method and did not reduce the number of move instructions.

[4] In the calling convention we used, a function can be compiled if the number of maximum reference distances is greater than the number of its arguments by at least one. This is because the stack pointer has to be updated immediately after the function call while keeping the values of the arguments and the return address. It is possible to generate code under shorter maximum reference distance conditions by modifying the calling convention, but that is beyond the scope of this paper.

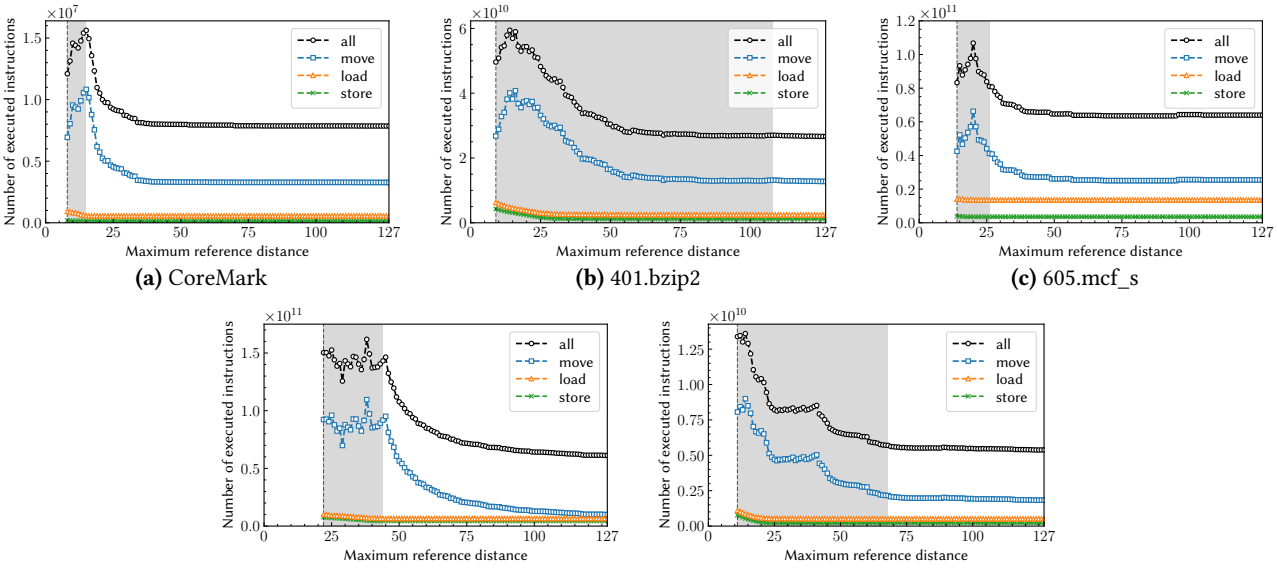


Figure 4. Number of executed move/load/store instructions for each benchmark. In this figure, the vertical dashed line indicates the lower bound on the maximum reference distance where the proposed algorithms can compile each benchmark. The shaded portion in each figure indicates conditions where the existing method could not generate code without additional spills.

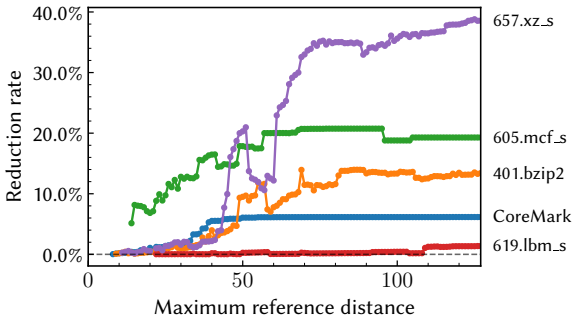


Figure 5. Reduction rate in move instructions when using our proposed SSA destruction compared with when using the naïve SSA destruction.

5.2.3 Reduction Rate of Move Instructions by Our Proposed Move Insertion. Figure 6 shows the reduction rate in executed move instructions when comparing the naïve and proposed move insertion methods. In this comparison, the SSA destruction method was unified to the naïve one to compare only the efficiency of move insertion. The hatched areas in this figure indicate the conditions where the naïve move insertion could not generate executable code. The naïve method of move insertion can lead to an infinite loop. For this reason, if compilation did not finish after a certain time, we concluded that the naïve method could not compile a program.

For 401.bzip2 and 657.xz_s, the naïve move insertion failed to generate code under some conditions with shorter maximum reference distances. This result shows the limitations of the ad hoc approach and demonstrates the effectiveness of the proposed algorithm.

Except for some conditions of 619.lbm_s, the number of move instructions in the proposed method was fewer than that in the naïve method. Particularly, in CoreMark, the number of move instructions was reduced by up to 12.5%. CoreMark is a benchmark in which load/store instructions increased significantly as the maximum reference distance decreased. In such a case, the number of live variables at each program point is almost the same as the number of the maximum reference distance; thus, it is difficult to resolve the distance constraints in the existing method. Hence, the reduction of the move instruction in our proposed method had a significant effect on CoreMark. Additionally, this property of a large increase in load/store of CoreMark is also common to 401.bzip2. Thus, it is expected that 401.bzip2 could also have a significant effect, although there are no data because it could not be compiled with the existing method.

For 619.lbm_s, we observed some conditions in which the proposed method executed more move instructions than the naïve method. Since the proposed method divides the process into two steps and optimally inserts move instructions in each step, it does not always achieve the overall optimal result. However, the increase in the number of instructions from existing methods is very small, if any.

6 Related Works

Our proposed method determines the position of instructions after spilling. This is similar to conventional register allocation techniques that completely decouple spilling

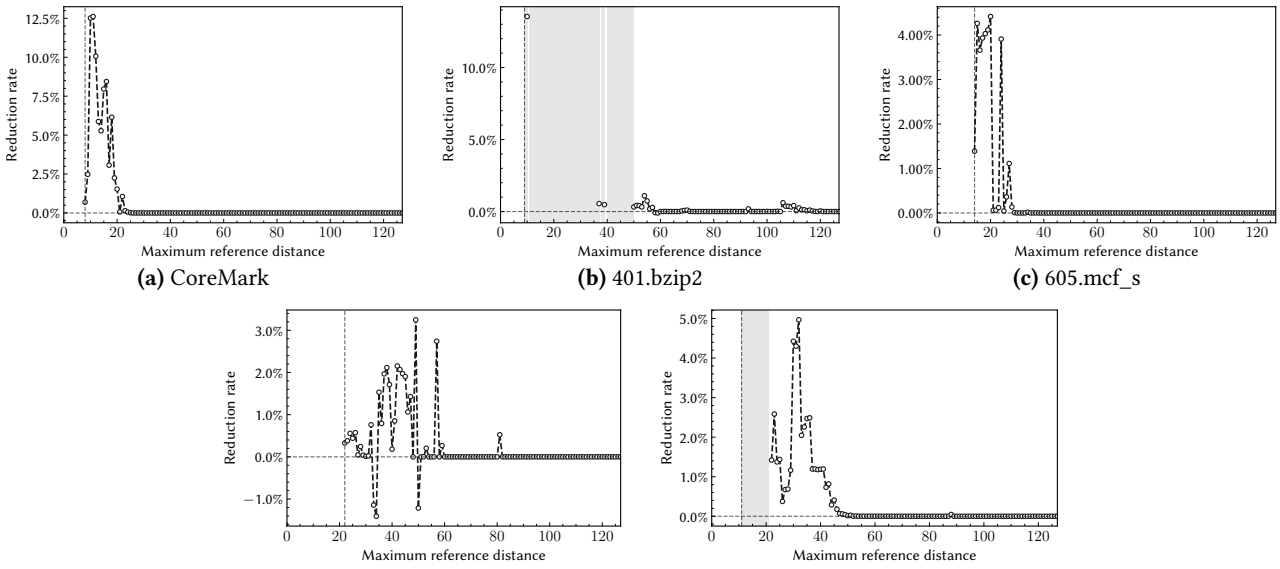


Figure 6. Reduction rate in move instructions when using our proposed move insertion compared to when using the naïve move insertion. In this figure, hatched areas indicate the conditions where the naïve move insertion could not satisfy the distance constraints.

and register assignment [7, 14, 25]. In conventional RISCs, some methods of register allocation that perform spilling and register assignment in the same framework are also proposed [13, 17]. On the contrary, such methods are not suitable for STRAIGHT. This is because inserting spill codes affects the distance of source operands of other instructions, reducing the efficiency of move insertion.

The instruction representation of STRAIGHT has been pointed out to have similarities with functional programming [26]. This is from the same observation as for the SSA form [6, 24], which has the same write-once manner as STRAIGHT. Consequently, methods used in functional programming may be useful for code generation and compiler optimization in STRAIGHT. For instance, the callee-save convention in STRAIGHT [26] uses the same approach as the continuation-passing style [7].

7 Conclusion

STRAIGHT is an instruction set architecture that uses the distance between instructions to specify the source operands. Owing to this unique instruction format, STRAIGHT has no false dependencies between instructions and thus does not require register renaming, which can cause a significant increase in power consumption. However, a STRAIGHT compiler must generate code that satisfies distance-related constraints. Existing methods attempt to satisfy these constraints in an ad hoc manner and thus may fail to generate executable code.

In this paper, we proposed three compiling techniques for STRAIGHT code generation. We proved that these techniques generate executable code under the theoretical shortest condition of the maximum reference distance where a benchmark can be compiled. The first was about spill code generation. This method limits the number of live variables so that the distance constraint can be satisfied with a minimum number of spill codes. The second was about SSA destruction. This method reliably and efficiently converts programs to the instruction format of STRAIGHT. The third was about move insertion. This method resolves the distance constraint by inserting the optimal number of move instructions for each of the two types of move instructions.

We implemented our proposed algorithm on an existing STRAIGHT compiler. We compiled and evaluated benchmark programs with our compiler under various conditions of maximum reference distance. The evaluation results showed that the proposed method could generate code even under conditions that existing methods could not. Moreover, our method reduced the number of move instructions for satisfying the distance constraints. The proposed method allows compilation at various maximum reference distances and evaluation of STRAIGHT processors in various configurations. Consequently, our result is a significant step forward for the future design of processors based on STRAIGHT ISA.

Acknowledgments

This work was partially supported by Premo Inc., JST SPRING Grant Number JPMJSP2108, and JSPS KAKENHI Grant Numbers JP19H04077, JP20H04153, JP20J22752.

References

[1] 2022. CoreMark. The embedded microprocessor benchmark consortium benchmark suite. http://www.coremark.org.

[2] 2022. musl libc. https://musl.libc.org/

[3] 2022. Onikiri2. https://github.com/onikiri/onikiri2

[4] 2022. straight-dev/env. https://github.com/straight-dev/env/

[5] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL'88)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/73560.73561

[6] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Not.* 33, 4 (apr 1998), 17–20. https://doi.org/10.1145/278283.278285

[7] Andrew W. Appel and Zhong Shao. 1992. Callee-Save Registers in Continuation-Passing Style. *Lisp Symb. Comput.* 5, 3 (sep 1992), 191–221. https://doi.org/10.1007/BF01807505

[8] David Bernstein, Martin C. Golumbic, Yishay Mansour, Ron Y. Pinter, Dina Q. Goldin, Hugo Krawczyk, and Itai Nahshon. 1989. Spill Code Minimization Techniques for Optimizing Compliers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI'89)*. Association for Computing Machinery, New York, NY, USA, 258–263. https://doi.org/10.1145/73141.74841

[9] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. 1998. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience* 28, 8 (1998), 859–881. https://doi.org/10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1994. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 428–455. https://doi.org/10.1145/177492.177575

[11] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[12] Thomas Burd, Wilson Li, James Pistole, Srividhya Venkataraman, Michael McCabe, Timothy Johnson, James Vinh, Thomas Yiu, Mark Wasio, Hon-Hin Wong, Daryl Lieu, Jonathan White, Benjamin Munger, Joshua Lindner, Javin Olson, Steven Bakke, Jeshuah Sniderman, Carson Henrion, Russell Schreiber, Eric Busta, Brett Johnson, Tim Jackson, Aron Miller, Ryan Miller, Matthew Pickett, Aaron Horiuchi, Josef Dvorak, Sabeesh Balagangadharan, Sajeesh Ammikkallingal, and Pankaj Kumar. 2022. Zen3: The AMD 2nd-Generation 7nm x86-64 Microprocessor Core. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 1–3. https://doi.org/10.1109/ISSCC42614.2022.9731678

[13] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[14] Quentin Colombet, Florian Brandner, and Alain Darte. 2015. Studying optimal spilling in the light of SSA. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4, Article 47 (jan 2015), 26 pages. https://doi.org/10.1145/2685392

[15] Keith D. Cooper and L. Taylor Simpson. 1998. Live Range Splitting in a Graph Coloring Register Allocator. In *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*. Springer-Verlag, Berlin, Heidelberg, 174–187. https://doi.org/10.1007/BFb0026430

[16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[17] Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3 (may 1996), 300–324. https://doi.org/10.1145/229542.229546

[18] Sebastian Hack, Daniel Grund, and Gerhard Goos. 2006. Register Allocation for Programs in SSA-Form. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. Springer-Verlag, Berlin, Heidelberg, 247–262. https://doi.org/10.1007/11688839_20

[19] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. https://doi.org/10.1145/3282307

[20] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (sep 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[21] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 07 (jul 2008), 33–38. https://doi.org/10.1109/MC.2008.209

[22] Hidetsugu Irie, Daisuke Fujiwara, Kazuki Majima, and Tsutomu Yoshinaga. 2012. STRAIGHT: Realizing a Lightweight Large Instruction Window by Using Eventually Consistent Distributed Registers. In *Proceedings of the 2012 Third International Conference on Networking and Computing (ICNC'12)*. IEEE Computer Society, USA, 336–342. https://doi.org/10.1109/ICNC.2012.66

[23] Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya, Takahiro Notsu, Katsuhiro Yoda, Teruo Ishihara, and Shuichi Sakai. 2018. STRAIGHT: Hazardless Processor Architecture without Register Renaming. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Fukuoka, Japan). IEEE Press, 121–133. https://doi.org/10.1109/MICRO.2018.00019

[24] Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. *SIGPLAN Not.* 30, 3 (mar 1995), 13–22. https://doi.org/10.1145/202530.202532

[25] David Ryan Koes and Seth Copen Goldstein. 2009. Register Allocation Deconstructed. In *Proceedings of Th 12th International Workshop on Software and Compilers for Embedded Systems* (Nice, France) *(SCOPES '09)*. Association for Computing Machinery, New York, NY, USA, 210–30. https://doi.org/10.1145/1543820.1543824

[26] Toru Koizumi, Shu Sugita, Ryota Shioya, Junichiro Kadomoto, Hidetsugu Irie, and Shuichi Sakai. 2021. Compiling and Optimizing Real-world Programs for STRAIGHT ISA. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE Computer Society, Los Alamitos, CA, USA, 400–408. https://doi.org/10.1109/ICCD53106.2021.00070

[27] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE Computer Society, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[28] Satoshi Mitsuno, Junichiro Kadomoto, Toru Koizumi, Ryota Shioya, Hidetsugu Irie, and Shuichi Sakai. 2020. A High-Performance Out-of-Order Soft Processor Without Register Renaming. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, Los Alamitos, CA, USA, 73–78. https://doi.org/10.1109/FPL50879.2020.00022

[29] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (sep 1999), 895–913. https://doi.org/10.1145/330249.330250

[30] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL'88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[31] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. 2022. Intel Alder Lake CPU Architectures. *IEEE Micro* 42, 3 (2022), 13–19. https://doi.org/10.1109/MM.2022.3164338

[32] Ryota Shioya and Hideki Ando. 2014. Energy efficiency improvement of renamed trace cache through the reduction of dependent path length. In *2014 32nd IEEE International Conference on Computer Design (ICCD)*. IEEE Computer Society, Los Alamitos, CA, USA, 416–423. https://doi.org/10.1109/ICCD.2014.6974714

[33] Ryota Shioya, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. 2010. Register Cache System Not for Latency Reduction Purpose. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE Computer Society, USA, 301–312. https://doi.org/10.1109/MICRO.2010.43

[34] Christian Wimmer and Michael Franz. 2010. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO'10)*. Association for Computing Machinery, New York, NY, USA, 170–179. https://doi.org/10.1145/1772954.1772979

[35] D. H. Younger. 1963. Minimum Feedback Arc Sets for a Directed Graph. *IEEE Transactions on Circuit Theory* 10, 2 (June 1963), 238–245. https://doi.org/10.1109/TCT.1963.1082116

Received 2022-11-10; accepted 2022-12-19