STRAIGHT: Hazardless Processor Architecture Without Register Renaming

Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya
The University of Tokyo
Tokyo, Japan
irie, koizumi, a.fukuda, akaki, nakae, bessho@mtl.t.u-tokyo.ac.jp, shioya@ci.i.u-tokyo.ac.jp

Takahiro Notsu, Katsuhiro Yoda, Teruo Ishihara
FUJITSU LABORATORIES LTD.
Kawasaki, Japan
notsu.takahiro, yoda.katsuhiro, ishihara@jp.fujitsu.com

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

***Abstract*—The single-thread performance of a processor improves the capability of the entire system by reducing the critical path latency of programs. Typically, conventional superscalar processors improve this performance by introducing out-of-order (OoO) execution with register renaming. However, it is also known to increase the complexity and affect the power efficiency. This paper realizes a novel computer architecture called "STRAIGHT" to resolve this dilemma. The key feature is a unique instruction format in which the source operand is given based on the distance from the producer instruction. By leveraging this format, register renaming is completely removed from the pipeline. This paper presents the practical Instruction Set Architecture (ISA) design, the novel efficient OoO microarchitecture, and the compilation algorithm for the STRAIGHT machine code. Because the ISA has sequential execution semantics, as in general CPUs, and is provided with a compiler, programming for the architecture is as easy as that of conventional CPUs. A compiler, an assembler, a linker, and a cycle-accurate simulator are developed to measure the performance. Moreover, an RTL description of STRAIGHT is developed to estimate the power reduction. The evaluation using standard benchmarks shows that the performance of STRAIGHT is 18.8% better than the conventional superscalar processor of the same issue-width and instruction window size. This improvement is achieved by STRAIGHT's rapid miss-recovery. Compilation technology for resolving the possible overhead of the ISA is also revealed. The RTL power analysis shows that the architecture reduces the power consumption by removing the power for renaming. The revealed performance and efficiencies support that STRAIGHT is a novel viable alternative for designing general purpose OoO processors.**

***Index Terms*—microprocessor, instruction-level-parallelism, out-of-order execution, register renaming, computer architecture, compiler, power efficiency**

I. Introduction

In response to the trends in semiconductor technology, processor architecture has been continually evolving to achieve higher performance, more functions, and higher power efficiency. Currently, for the wide range of purpose from embedded processors to server processors, the heterogeneous multi-core architecture [1] is adopted, which typically implements various types and various scales of cores into a chip; the most efficient cores among them are in charge for each application depending on its characteristics. This strategy trades off energy efficiency against lower-utilized cores, reflecting the recent dilemma that even though the number of transistors can be increased, they cannot be switched simultaneously [2]. In this scenario, the CPU is expected to effectively execute the programs that are not parallelized or cannot be parallelized by the programmer. Because they are often critical paths that require the longest execution time among the ongoing tasks, speeding up a single-thread execution is essential to improve the total system performance.

Currently, a big superscalar core is inevitably accepted as the most powerful and the only feasible architecture to gain this performance. Provided with an out-of-order (OoO) mechanism and a number of predictors, the sophisticated superscalar core can exploit the underlying instruction-level parallelism (ILP) in a thread relatively well. However, a further increase in its performance is challenging because the power of indirect operation increases with the scale of the core, which is critical in today's limited power budgets. Therefore, the recent improvement in single-thread performance is relatively modest compared to those of GPUs and TLP technologies, which demonstrate a performance increase that is proportional to the increase in the number of transistors employed [3] [4].

This paper realizes a novel OoO execution architecture that reduces the amount of indirect operations per instruction. The key idea is that the unique instruction set architecture (ISA) is adopted to skip register renaming. The ISA guarantees that each logical register will be written only once and then discarded in a fixed period. As the paper elaborates later, this rule renders register renaming unnecessary, thus resulting in a simple OoO execution structure as well as a scalable instruction window for exploiting much larger ILP with simple

This research was partially supported by JSPS KAKENHI Grant Number 25730028, and 16H05855. Also, this research was partially supported by VLSI Design and Education Center(VDEC), The University of Tokyo with the collaboration with Synopsys Corporation and Mentor Graphics Corporation.

hardware. The architecture is called "STRAIGHT" because it executes each instruction directly without renaming its operands. The architecture actually requires the ISA to be changed; however, the code is easily translated from the static-single-assignment (SSA) form [5] [6] intermediate language, which is recently dominant for compiler infrastructures such that the viability is not compromised.

A cycle-accurate simulation environment and a register-transfer level (RTL) description of STRAIGHT are developed to the evaluation. Also, those environment of OoO RISC-V are developed as a superscalar counterpart. The evaluation result demonstrates that STRAIGHT improves the single-thread performance and reduces the power consumption, both of which are derived from the simpler hardware organization that is enabled by the ISA.

The concept of the ISA to eliminate register renaming is presented herein [7]; however, the method to realize the microarchitecture and the compiler of the architecture that can execute any kind of general-purpose programs has not been revealed. By presenting the essential hardware and software technologies, the contribution of this paper are as follows:

- The practical STRAIGHT ISA is built to write application programs. The ISA feature is that the source operands are given by the distance from the producer instruction.
- The microarchitecture that receives STRAIGHT ISA is realized. We show that this approach completely removes register renaming from the OoO core. This means that the major hotspots and critical paths are eliminated from the front-end pipeline. Furthermore, the simplified architecture achieves the rapid miss-recovery as well as the further reorder buffer (ROB) scalability.
- The compilation algorithm that generates STRAIGHT machine code from the LLVM [8] intermediate representation (IR) is developed. Padding with simple register move (`RMOV`) instructions, we show that most operands can be converted to the statically determined distances and the remainder can also be represented using the `SPADD` instruction.
- A cycle-accurate simulator that faithfully models the pipeline stages of STRAIGHT is developed for the performance evaluation. The compiled code of standard benchmarks (Dhrystone, Coremark) is used. The result shows that STRAIGHT achieves 18.8% better performance than modern OoO superscalar processors while requiring a simpler hardware. STRAIGHT's lower miss-penalty enhances the performance. The restriction of STRAIGHT ISA can deteriorate the performance with increasing redundant `RMOV` instructions; however, most of them are eliminated algorithmically by the compiler.
- The power efficiency is confirmed through RTL power analysis. In-house RTL descriptions of STRAIGHT is prepared. Comparative evaluation reveals that STRAIGHT removes the power of register renaming and improves the entire efficiency of the processor.

The remainder of this paper is organized as follows: Section II describes the motivation and the basic concept of the STRAIGHT architecture. Section III presents the details of both the ISA and the microarchitecture. Section IV presents the compiler algorithm to generate the STRAIGHT machine code. Section V shows the evaluation settings, followed by section VI, which presents the evaluation results. Section VII discusses the related works, and the paper is concluded in section VIII.

II. Motivation

A. Advantages and Disadvantages of Register Renaming

To enhance the single-thread performance, register renaming is critical in exploiting the additional ILP by solving the false dependency hazard dynamically. In the front-end pipeline, register renaming is applied to every instruction. It converts all operand identifiers from logical register numbers to physical register numbers that indicate the addresses in the internal register file. This operation is implemented by multiple references to the table composed of RAM or CAM that is called the register mapping table (RMT) or the register alias table (RAT).

In the RAM-based RMT, which has as many rows as the number of logical registers, the RMT stores the corresponding physical register number using the logical register number as an index. The mechanism also has a circular FIFO called free-list that holds all the physical register numbers that do not correspond to any logical register at that time. The operation for each instruction is as follows: i) The RMT is accessed with source register numbers, and source physical register numbers are obtained. The RMT is accessed simultaneously with the destination register number, and the previous dedicated physical register number is obtained for the recovery and retire operation. ii) A physical register number is provided from the head of the free-list, and sent to the next pipeline stage as the destination physical register number. The RMT is simultaneously updated by writing this physical register number. Through these operations, the code is now able to exploit much greater parallelism.

However, register renaming is reported to be one of the hotspots in terms of both power density and power consumption [9] [10] [11] [12]. It lowers the efficiency of the OoO CPU cores, with the result that a large fraction of energy is consumed by RMT accesses that are not essential for the execution. Each instruction requires three reads and one write in a cycle, and the required number of read/write ports is multiplied by the number of fetch width, which renders the RMT one of the most multiported tables in the processor. This multiported access also affects the clock frequency because unless the RMT updates are completed, the renaming of the next fetch group cannot start [13].

Moreover, the scalability of the instruction window whose size determines the amount of exploitable ILP is also disturbed by register renaming. The instruction window size is directly related to the ROB, in which all in-flight instructions from the dispatch stage to the retire stage are queued. However, the size of ROB proportionally increases the branch miss-penalty.

When a mispredicted branch instruction is detected, the RMT must be restored by walking the ROB from the tail (or from the head, depending on the implementation) to the corresponding branch instruction. The penalty is reported as several tens of cycles with the 256-entry ROB on average, which considerably affects the performance [14].

To be exact, a CAM-based RMT can avoid this penalty at the expense of a large amount of RMT checkpoints. However, it still disturbs the ROB scalability because the CAM-based RMT cannot increase the number of physical registers. If the number of physical registers are not sufficient for the number of ROB entries, the stall due to the shortage of free physical registers increases. Although register renaming provides an important contribution to the acceleration of single-thread execution, it also introduces design limitations, that causes the slow progress in core performance improvement.

B. STRAIGHT Architecture Concept

Instead of managing the physical registers dynamically, the STRAIGHT concept has been presented to solve the false dependency hazard by the compiler, which eliminates register overwrites [7]. Because the number of registers is finite, each register is freed after a fixed period such that the program length is not limited. To write the looped code without overwrites, a technique is introduced. Register numbers are indicated relatively by the dynamic instruction distance. In the architecture, fetched instructions can be directly dispatched to the scheduler. Not managed by any mapping table, the register file can be built as a simple key-value store that is easy to scale.

However, realizing the computer for this novel instruction format involves a number of challenges. Because of the unique operand format, the practical ISA that can write the real application has to be newly constructed. Especially, its viability significantly owes to the code generation, nevertheless, to develop the compiler for this instruction format is a novel challenge. Moreover, in previous STRAIGHT concept, the scalability has been prioritized and the large hardware that is provided with thousands of registers are assumed, which limits its use to the desktop or sever processors.

Hereafter in this paper, we realize and evaluate STRAIGHT processor by resolving the challenges above. Also We design STRAIGHT microarchitecture as small as conventional mobile processors, which increases the opportunities to be utilized.

III. Efficient STRAIGHT Processor Core

A. Specifying STRAIGHT ISA

The developed STRAIGHT instruction set is a collection of simple operations similar to the typical RISC architectures. As in conventional OoO superscalar architectures, it provides a sequential execution model and precise interrupts to the programmer.

Figure 1(a) shows a simple example of the STRAIGHT code. Each instruction performs an ordinary operation, but its operands are given by the distance from the instruction that produces the source value. For example, "`[1]`" of instruction

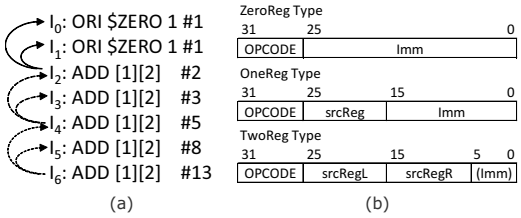

Fig. 1. Example of STRAIGHT code and bit-field formats

I_2 means that the instruction uses the result value of the previous instruction, and "`[2]`" means that the other operand is the result value of the second previous instruction. Therefore, this code calculates a Fibonacci series as long as the "`ADD [1] [2]`" instruction is repeated. It is noteworthy that an operand is represented as the distance in a control flow. Therefore, it differs from the distance in the static order when a code contains nonsequential program counter (PC) transitions such as jump instructions.

Based on the above concept, we defined the bit-field format of the STRAIGHT instruction set architecture as shown in figure 1(b). Because an instruction does not specify the destination register identifier, each identifier for the source operands can use a larger field. A source operand field can span up to 10 bits, which means that the results of the last $2^{10} - 1 = 1023$ instructions can be referenced. Here, "`[0]`" is decoded as a zero register. Store (`ST`) instructions need not make the register output, but to make the distance calculation simple, each instruction occupies one destination register. If it is referred, store value is returned in the current specification.

This unique representation contains some characteristics of the dataflow, and satisfies the following restrictions corresponding to the STRAIGHT concept. First, it guarantees the write-once usage of each register. A register is identified by the fetch order of the instruction, with the result that any two instructions never share the same destination. Next, because the operand is identified by the distance in the control flow path, the lifetime of each register is determined by the maximum distance (e.g., 1023 instructions). The value will be never referenced after the succeeding 1023 instructions are executed.

STRAIGHT requires an additional architectural register, "stack pointer (SP)." This is the only overwritable register, and it guarantees that any complicated algorithm can be written at the least using SP as a memory pointer. We find that adding one instruction, "`SPADD`" to the instruction set is sufficient to SP-related operations, which reads and modifies the SP by adding the given immediate value as well as writing this result to its destination register. The succeeding load and store instructions can use the SP value by indicating the distance from the `SPADD` instruction. As described in the compiler section below, the `SPADD` is utilized to compile the complicated control flows such as function calls.

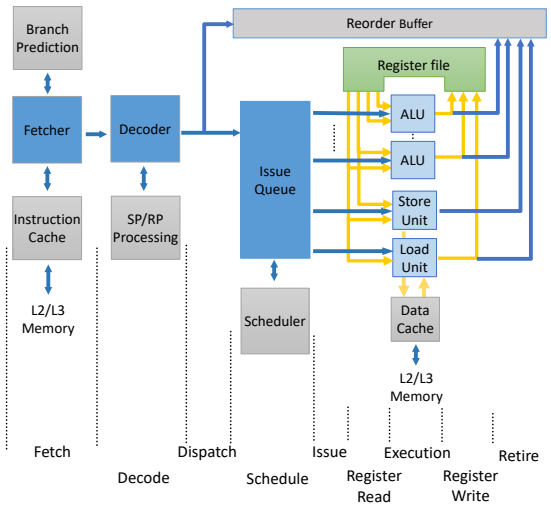

Fig. 2. Pipeline organization of STRAIGHT

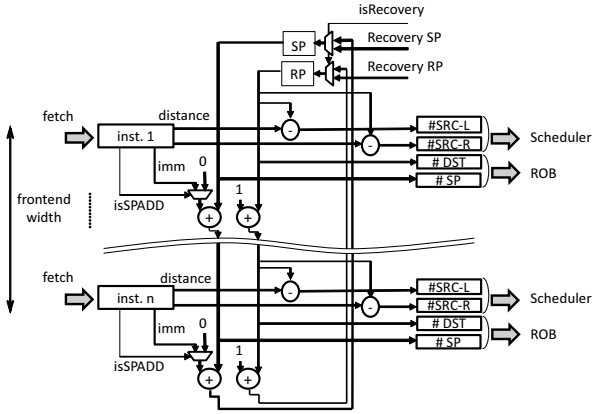

Fig. 3. The mechanism of operand determination in the front-end pipeline

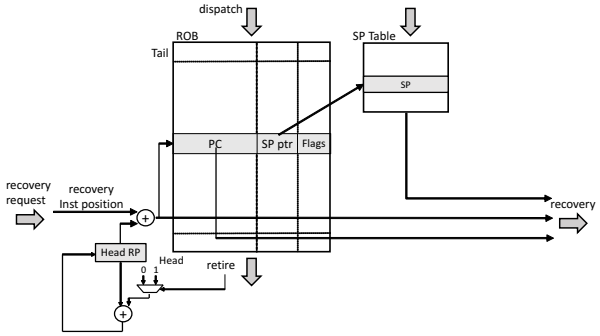

Fig. 4. The mechanism of retire/recovery operation

B. Realizing Scalable OoO Execution

The pipeline organization of STRAIGHT is designed as shown in figure 2. As the operations of the instruction set are similar to those of conventional RISC architectures, most of the pipelines are also composed of conventional mechanisms. The major differences are in the following mechanisms: the mechanism in the front-end pipeline that determines the operand register number and the retire/recovery mechanism, which is related to the ROB.

First, the detailed mechanism of the operand determination is shown in figure 3. This operation corresponds to the RMT accesses for conventional OoO superscalar processors. To obtain the physical source register numbers from the source distances given by an instruction, a special register, register pointer (RP), is introduced in the hardware. Incremented for each instruction, the RP value provides the destination register number, and the source register numbers are obtained by subtracting the distance from RP. As shown in figure 3, multiple instructions in a fetch group can be operated in parallel because the RP value for each instruction is rigidly determined regardless of the operation of the preceding instruction. Therefore, the fetch width can extend as far as it is effective. The RP value returns to zero when it exceeds its maximum value *MAX_RP*.

Figure 3 also shows the mechanism related to SP. SP is the only overwritable register and is operated by only the `SPADD` instruction. As we designed the `SPADD` instructions to require only an immediate operand, the instruction can update the SP in order at the decode stage. Subsequently, the `SPADD` instruction writes the copied SP value to its destination register, which can be performed in the OoO manner. Guaranteeing multiple `SPADD`s in a fetch group requires the cascaded `SPADD` calculations in a cycle, which possibly affect the clock frequency. The number of `SPADD` instructions in a fetch group can be restricted by stalling and the performance effect of this stall is negligible because the `SPADD` interval is very long (two per function call, at the most).

Next, the retire/recovery mechanism is shown in figure 4. Composed of a circular FIFO structure, the ROB maintains the information about each in-flight instruction. One entry maintains the part of the architecture state that will be changed by the corresponding instruction. In STRAIGHT, the RP and the PC of the instruction and the SP value at its decode time are sufficient for an entry. Unlike the conventional architecture, the retire operation only removes the completed instructions in order from the head of the ROB unless the instruction causes the exception or miss-recovery.

To reduce the hardware, each entry of the ROB does not have to maintain the relevant RP. Only if the RP copy corresponding to the ROB head is maintained, can the rest be derived from the entry position. In addition, storing the SP value for each ROB entry is also redundant because the SP updates are very rare. Providing a small table and maintaining pointers for it in the ROB saves significant ROB capacity.

The recovery operation from the branch misprediction or other speculation is extremely simple compared to typical superscalar architectures. Only one ROB entry read is sufficient. By using the destination register number of the oldest instruction in the discarded path as a key, the SP and PC (that is obtained by the execution result for branch misprediction recovery) for the restart are obtained from the ROB. That

register number is also used for RP restoration. No ROB updates are required except the tail pointer movement. This operation is sufficient to ensure that the microarchitectural changes generated by the mispredicted instructions will be overwritten eventually by the restarted correct path execution. Although not mandatory, to save the power consumed by invalid instructions, a partial flush of the scheduler, pipelined functional units, and the load-store queue can be performed using the corresponding RP value.

MAX_RP physical registers are sufficient to avoid unexpected physical register aliasing because the register number never exceeds that value. Here, one should consider deriving the value of *MAX_RP* to save the register values from the unexpected overwrites. When instruction k that uses k as the destination register number is about to retire, the value of register $(k -$ the maximum distance$)$ must still be maintained. Therefore, the corresponding number must not appear in RP before the retirement of instruction k. Nevertheless, at that time the youngest in-flight instruction receives the register number $(k +$ the number of ROB entries$)$. Thus, *MAX_RP* is given by $($the maximum distance $+$ the number of ROB entries$)$.

As shown, STRAIGHT improves the efficiency of the OoO execution by removing register renaming. Moreover, because it eliminates the ROB walking penalty, STRAIGHT enables the instruction window to be further increased. By eliminating hotspots caused by register renaming, STRAIGHT can operate at the same or higher frequency compared to the conventional OoO superscalar processor with the same issue width and the same number of functional units. Nevertheless, the architecture does not prevent configuring the small efficient core. The number of the physical register required is determined by the ROB size and the maximum distance, therefore smaller core can be configured by shrinking those values.

IV. STRAIGHT Compiler Technology

A. Compilation Flow of STRAIGHT Compiler

The STRAIGHT compiler generates the STRAIGHT assembly where every source register is expressed as a distance. In this section, we explain that any program is converted into the STRAIGHT assembly correctly. We adopted LLVM IR as the input of the STRAIGHT compiler. The advantage of adopting it is that LLVM IR is an SSA-formed IR. Every destination register in the SSA-formed IR is not overwritten statically and this manner is similar to the register management of STRAIGHT.

In the STRAIGHT architecture, every distance is calculated by the number of in-order instructions between the producer and consumer on the execution path. However, if any control flow merges in the control statements including the if- and while- statements, the distance in each path can differ. In this case, the STRAIGHT compiler adds instructions such that the distance is fixed, as described below.

B. Calling Convention

Basically, the STRAIGHT architecture stores arguments and returns values in registers. Instructions that generate those values are arranged in a fixed order defined by the function, as shown in Figure 5. In this manner, any caller can pass and receive the variables. For example, Figure 6 shows how arguments and return values are passed. In this example, the calling convention is to place the producer of argument arg1 just prior to the jump-and-link (`JAL`) instruction and place the producer of arg0 just before it. As long as callers satisfy this convention, distances between an instruction in the callee to producers of the argument are fixed regardless of which `JAL` instruction is invoked. In the example, the instruction `ADD [4] [3]` in the callee always refers `arg0` and `arg1`.

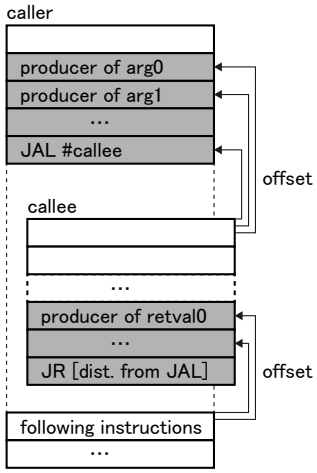

Fig. 5. Instruction Arrangement in Calling Convention

The return address is passed to the callee by the `JAL` instruction that writes its `PC+4` to its destination register. The jump register (`JR`) instruction in the callee refers the JAL instruction by the distance, which realizes the return operation. Variable arguments such as `printf` in C language requires are passed via stack frame. Calling conventions for System calls are defined in a similar manner. Functions can be defined to return one or multiple values by defining the distances between the `JR` instruction and producers of return values. For example, the `ADDi [3] 1` instruction following the `JR` instruction calculates `retval0` plus 1.

If the producer instructions cannot be arranged as the convention, the register move (`RMOV`) instructions are used instead. `RMOV` copies its source register value into its destination register, such that it can arrange the order of the produced values.

Alive variables are stored in the stack frame using the SP before the function call. The SP is incremented or decremented by the `SPADD` instruction. `SPADD` instructions are generated at the entrance and the exit of the function. `LD` and `ST` instructions can access the stack frame by referring the relevant `SPADD` instruction because `SPADD` writes the updated value of SP into its destination register.

C. Code Generation

1) Operation Translation: Figure 7 shows the compilation flow. The compilation process consists of three steps. First,

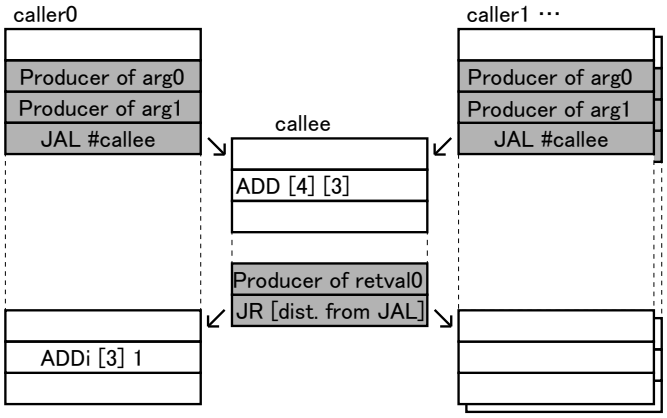

Fig. 6. Flows of Basic Blocks in Calling

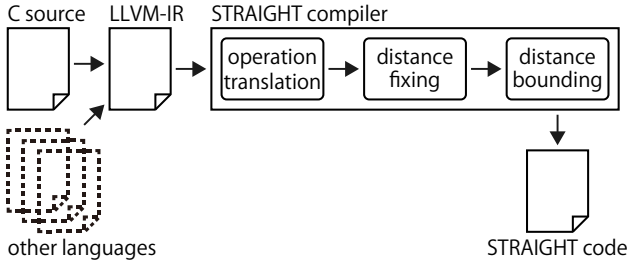

Fig. 7. Flow of Compilation

the STRAIGHT compiler translates operations in LLVM IR into those in STRAIGHT as the compiler for conventional architectures. Some instructions are added or deleted in this process because operations in LLVM IR and STRAIGHT do not always correspond exactly.

2) Distance Fixing on Merging Flow: Here, the STRAIGHT compiler adjusts distances from a consumer instruction to producer instructions to the same length regardless of the control flow. Any control flow is classified as either branching or merging. Figure 8(a) shows that the distances to the producer instruction are always fixed in the case of branch. However, distances can differ in the case of merge, as shown in Figure 8(b). In this case, the STRAIGHT compiler adds `RMOV`s at the tail of each merging basic block to fix the distance basically, as shown in Figure 8(c).

These `RMOV`s are added by using the following algorithm. First, the STRAIGHT compiler obtains the information on all possible producers and corresponding merging basic blocks from PHI instructions in LLVM IR. A PHI instruction is appeared when a operand has multiple producer candidates. A distance also varies when there are multiple paths from one producer instruction to its consumer. The information of such producers and merging basic blocks is obtained by liveness analysis as well.

`RMOV`s are added at the tail of such merging basic blocks to fix the distance regardless of the control flow. This process is repeated as many times as the number of live variables. `RMOV`s are stacked up on the tail of merging basic blocks

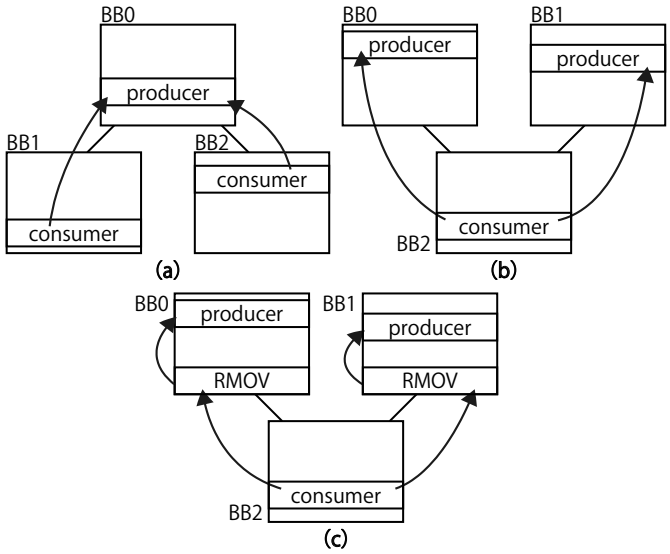

Fig. 8. Distances on control flows

not to change the distances that have been already fixed. `NOP`s are also added to eliminate the distance differences that are caused when there are fall-through paths. Although these `RMOV`s appear to be redundant, the optimization described later can reduce them.

Figure 9 shows an example of the fixed distances of the counter variable in the loop statement. The counter variable is initialized in the *BB0* (`ADDi [0] 0`) and incremented before the conditional statement in the *BB2* (`ADDi [4] 1`). The `SLT` at the head of the *BB1* take the counter variable as its source operand. There are two paths to this operand: from the *BB0* and the *BB2*. The operand of `SLTi` is fixed as `[2]` by adding `RMOV`s.

3) Distance Bounding: The STRAIGHT architecture has the maximum distance of source registers. Therefore, the STRAIGHT compiler adds `RMOV`s that relay the values when any distance exceeds the maximum. Each `RMOV` is added in the possible maximum distance from the consumer avoiding the added `RMOV`s. The STRAIGHT compiler repeats this process until every distance is equal to or less than the maximum.

D. Optimization specific to STRAIGHT

Thus far, the method that convert any program to the STRAIGHT code that satisfies the requirement of the architecture is described. Furthermore, in addition to the basic STRAIGHT code generation, the compiler can optimize the code from the aspect of its specific register management.

We designed the redundancy elimination that reduces `RMOV`s of which example is shown in the figure 10. (a), (b), and (c) are lists of STRAIGHT code that are compiled from the source code above with the different optimization level. Here, "*%%*" in the list is a pseudo code which generates the indicated variables.

In the basic compilation algorithm, `RMOV`s are generated proportionally to the number of live variables that are read

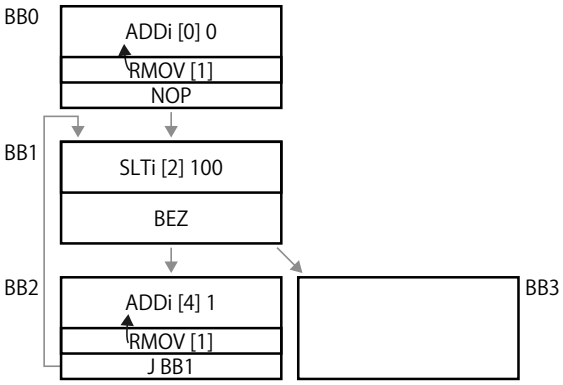

Fig. 9. Distance fixing of loop-statement

through the merging flows. For example, the loop body in the figure 10(a) contains four RMOVs out of ten instructions. The instructions only for fixing the distance occupy most of the loops, which causes the performance degradation. However, the distances can be fixed without adding RMOVs by rearranging the producer instructions in most cases. In the example of the figure10(b), two ADDi instructions can place instead of RMOVs because their result are not used within the iteration. Now those instructions generate values and adjust distances at the same time. The number of RMOVs is reduced to two out of eight instructions.

The figure 10(c) shows the other method to remove the RMOV instruction. In the example, the return address "_RETADDR" is stored in the stack frame and loaded after the loop completion. In fact, the variable is never used during the loop execution, therefore the corresponding RMOV only relays the variable. Storing such variables in the stack frame can reduces all RMOVs that only relay the variables. Now the loop contains only one RMOV out of seven instructions. Generally, in cases where the variables are not read in the near future, this method is effective.

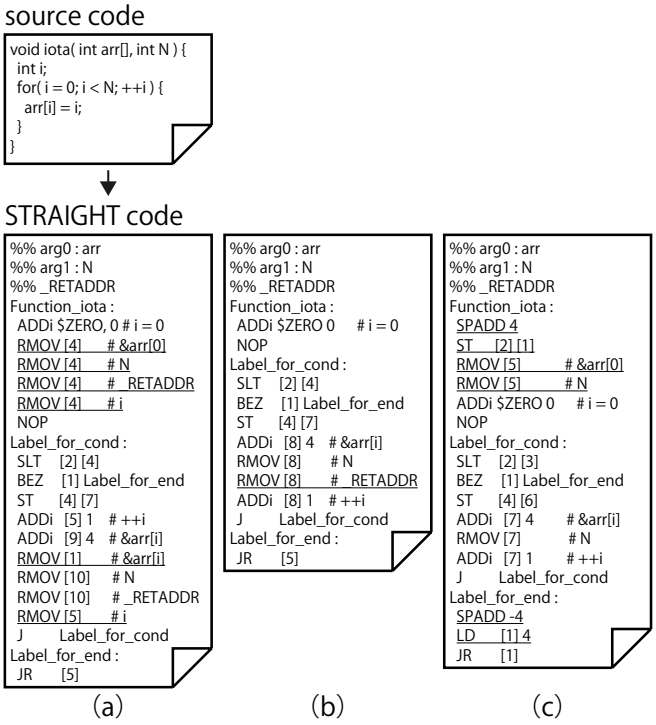

Fig. 10. STRAIGHT code and optimization

V. Evaluation Method

A. Simulated Models

STRAIGHT is a novel architecture that has different instruction sets and execution pipelines. To confirm its feasibility for executing application programs, we developed an in-house cycle-accurate simulation environment. A compiler, an assembler, a linker, and a cycle-accurate simulator are developed. The simulator faithfully models all pipeline stages of STRAIGHT, involving OoO scheduling, branch prediction, memory dependency prediction, a load-store queue (LSQ) for memory disambiguation, cache hit/miss prediction, the scheduler replay, a stream prefetcher for data caches, and the mechanisms for the misprediction recovery.

We also compared STRAIGHT to the conventional superscalar to reveal whether this architecture improves the performance and efficiency. An in-house cycle-accurate simulator of RISC-V (RV32IM) is also developed as a superscalar counterpart. The RAM based ROB is assumed in the evaluation. The front-end is stalled if the walking has not been completed when the first group of re-fetched instructions reaches to the rename stage. The ROB-walking width is same as the speed of frontend-width. Because the back-end pipeline of STRAIGHT is similar to conventional superscalar processors, both simulators can share common codes for the most part. Therefore, this superscalar model is also provided with the same state-of-art ILP technologies as described in above. As the equalization to RV32IM, we set STRAIGHT as a 32bit architecture and disabled the floating-point instructions and modules in the evaluation.

The evaluated processor models and their parameters are shown in Table I. As shown, the sizes of each module are set to the same value between the "SS" and "STRAIGHT" models to clarify the comparison, which represent the superscalar and STRAIGHT, respectively. We configure two classes named "4way" and "2way" to confirm the behavior of the different typical design scales. "4way" models the high-end CPU cores for desktop PCs and servers in which the OoO execution fully shows its abilities. "2way" models smaller OoO cores for the emerging mobile devices. The maximum distance is set to 31 in STRAIGHT-*4way* and STRAIGHT-*2way* models. This value is determined only for equalizing the number of the ROB entries and the number of the physical registers to SS parameters. The architecture's optimal parameter can be different, however the parameter is set to clarify the comparison as described above. As the instruction set is different from each other, the performance is measured by the execution cycles to complete the programs that are compiled from the same

TABLE I
EVALUATED MODELS

	2-way		4-way	
	SS	STRAIGHT	SS	STRAIGHT
ISA	RV32IM	STRAIGHT	RV32IM	STRAIGHT
Fetch Width	2		6	
Front-end latency	8	6	8	6
ROB Capacity	64		224	
Scheduler	2 way, 16 entries		4 way, 96 entries	
Register File	96		256	
LSQ	LD 48 / ST 48		LD 72 / ST 56	
Exec Unit	ALU 2, MUL 1, DIV 1, BC 2, Mem 2		ALU 4, MUL 2, DIV 1, BC 4, Mem 4	
Commit Width	3		4	
Branch Predictor	Gshare, Global History 10 bits, 32K entries			
L1I Cache	32 KiB, 4 way, 64 B line, 4 cycle hit latency			
L1D Cache	32 KiB, 4 way, 64 B line, 4 cycle hit latency			
L2 Cache	256 KiB, 4 way, 64 B line, 12 cycle hit latency			
L3 Cache	N/A		2 MiB, 4 way, 64 B line, 42 cycle hit latency	
Main Memory	200 cycle latency			

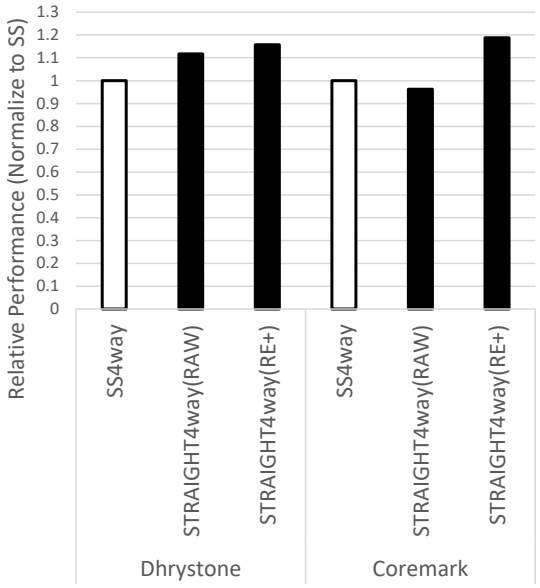

Fig. 11. Performance comparison of STRAIGHT and SS (4way)

source code.

Two standard benchmarks are used, Dhrystone 2.1 and CoreMark. Both are representative general-purpose integer benchmarks, which are suited to evaluate the novel processor's fundamental characteristics. The STRAIGHT code is generated by our STRAIGHT compiler that takes the LLVM IR code obtained by using clang 3.8 with `-O2` and `--target=mips-pc_linux-gnu -S -emit-llvm` options as an input. The target option is only for indicating the 32-bit architecture. The specific compiler front-end for STRAIGHT does not exist yet; however, currently the front-end for mips is sufficient to output an intermediate code for the STRAIGHT compiler back-end. For both benchmarks, two STRAIGHT binaries are prepared; STRAIGHT_RAW that is generated by the basic algorithm described in Section IV-A to IV-C and STRAIGHT_RE+ that is generated by adding the redundancy elimination described in IV-D to STRAIGHT_RAW.

For the comparison, clang/LLVM is also used for generating the RISC-V code. The back-end for LLVM by lowRISC is used. The code is generated with the `-O2` and `-march=rv32im --target=riscv32` option.

The cycles to complete the fixed number of iterations of the benchmark program is measured: 9000 times for Dhrystone, and 9 times for CoreMark. The performances are shown by using the inverse of the execution cycles.

B. RTL Power Analysis

To reveal the improvement of the power efficiency, we performed a power analysis in the register-transfer level (RTL). The RTL description of STRAIGHT that faithfully models the OoO execution, speculation, and recovery mechanisms is developed as well as an in-house RTL description of RV32I that involves the state-of-the-art superscalar technology. Similar to our simulator, both descriptions use the common code as much as possible for the comparison. In fact, the only differences are the modules related to front-end stages and ROB. The organizations of the RTL processors are almost the same as that of STRAIGHT-*2way* and SS-*2way* in the table I; however, the functions for integer multiply/divide instructions and general system calls are omitted.

We confirmed that both RTL processors can execute a test code correctly. The power consumed during the test code execution is analyzed using the Joules RTL Power Solution of Cadence. For both processors, the same recent advanced technology node is assumed. The baseline clock frequencies is set to be comparable to that of modern mobile processors. Several clock frequencies are set for the evaluation, in which the RTL processors are synthesized with the relevant time constraint for each.

VI. Evaluation

A. Performance Comparison to the Conventional Superscalar

Figure 11 shows the performance of STRAIGHT-*4way* and SS-*4way*. Each bar shows the relative performance that is normalized to SS-*4way*. The white bars represent the performances of the superscalar processor and the black bars represent those of STRAIGHT. STRAIGHT on Dhrystone and STRAIGHT-*4way*_RE+ on CoreMark show the better performance than SS-*4way*, which are by 15.7% on Dhrystone and 18.8% on CoreMark, respectively.

On the other hand, the possible overhead of STRAIGHT ISA is shown in STRAIGHT-*4way*_RAW on CoreMark that degrades the performance by 4% from the baseline SS. The reason is that a number of `RMOV` instructions are added by the basic STRAIGHT compiler because CoreMark tends to have the larger number of alive values through the execution than

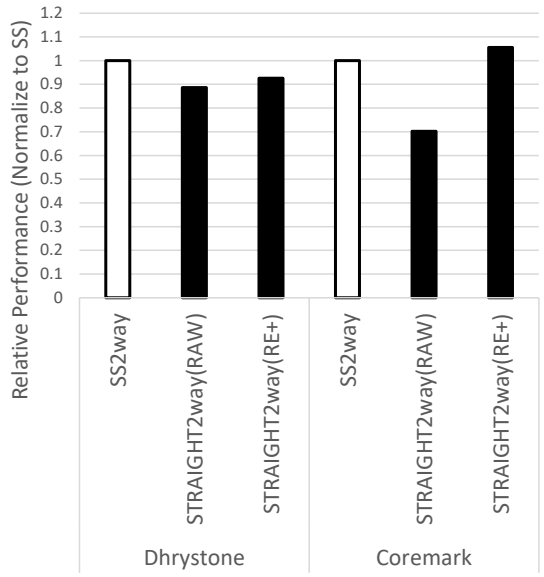

Fig. 12. Performance comparison of STRAIGHT and SS (2way)

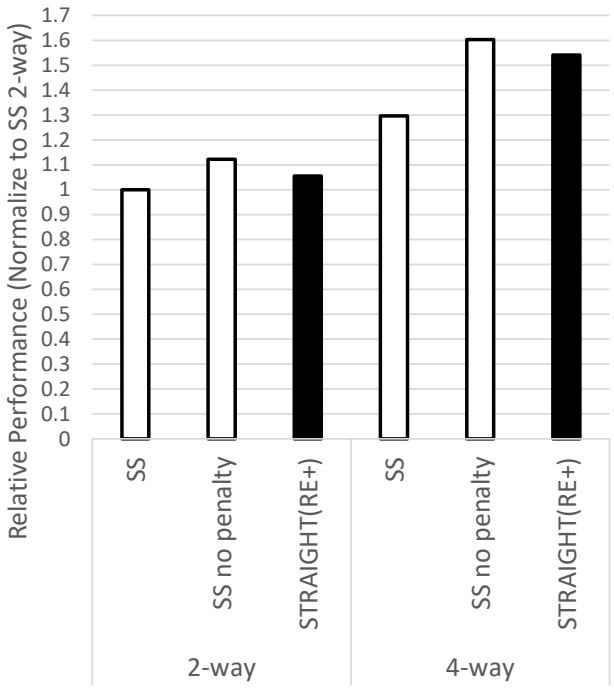

Fig. 13. The effect of the misprediction penalty

Dhrystone. However, the graph also shows that the overhead can be reduced by the redundancy elimination algorithm, as STRAIGHT-*4way*_RE+ shows the best performance.

Figure 12 shows the performance comparison of STRAIGHT-*2way* and SS-*2way* in the same manner. The relative performance degradation of STRAIGHT-*2way*_RAW is larger than that of the four-way configuration. This is as expected; in the current model, each `RMOV` instruction behaves as one ALU instruction. The impact of increased `RMOV` instruction becomes relatively large in the smaller configuration. However, in this small OoO core configuration, STRAIGHT-*2way*_RE+ also shows a comparable or better performance to its superscalar counterpart; it degrades the performance by 7.4% from SS-*2way* on Dhrystone but improves by 5.5% on CoreMark. The redundant elimination algorithm is also effective for the smaller core.

As both architectures are configured to have the same pipeline width, instruction window sizes, and predictors, the significant reasons for performance difference are as follows: i) ISA characteristics in STRAIGHT that eliminates register overwrites by adding `RMOV` instructions or `LD`/`ST` instructions. ii) STRAIGHT has fewer misprediction penalties because of its recovery mechanism and shorter front-end pipeline. The former can degrade the performance of STRAIGHT while the latter improves the performance.

The impact of the rapid recovery of STRAIGHT is shown in Figure 13. The graph shows the performance of SS and STRAIGHT_RE+ as well as the performance when the misprediction penalty of SS is idealized to zero. Both *2way* and *4way* performances on CoreMark are shown.

As shown in the graph, the effect of misprediction penalty is significant for the superscalar. The effect is around 20% that is similar amount to the reported in [14] on the integer programs when RAM based RMT and ROB walking are configured. Although the RMT technologies for reducing this penalty are exist, they increase the RMT power further. As the strong point of STRAIGHT, it reduces this penalty with simple hardware such that the performance and efficiency are both improved.

Branch prediction technologies can also reduce the recovery penalty. To reveal this impact, we implemented TAGE predictor (8-component CBP-TAGE) [15] to our simulator. Figure 14 shows the relative performance of CoreMark when TAGE is used instead of the conventional gshare predictor. As expected, relative performances of STRAIGHT is reduced for both *2-way* and *4-way* because the performance of the baseline SS is relatively improved by the reduced recovery penalty. However, STRAIGHT-*4way* shows 10% better performance even in this configuration. The basic characteristic is kept; STRAIGHT shows comparable or better performance.

Next, figure 15 shows the number of executed instructions on CoreMark to contrast code characteristics. Each bar is normalized to the total instruction count of SS. The graph also shows the fraction of instruction types for jump or branch, ALU, LD/ST, RMOV, NOP, and others. The graph shows that STRAIGHT_RAW requires more code than SS to complete the same program. The increased instructions are shown to be primarily `RMOV`s to adjust the distance. This is the possible impact of the naive STRAIGHT ISA code in exchange for the simpler hardware.

STRAIGHT_RE+ reduces the instruction count drastically, which shows the impact of the compiler technology for STRAIGHT. The count of increased `RMOV`s are now reduced to about 20% of the baseline SS. As the performance evaluation shows, the performance of STRAIGHT_RE+ is better than SS in the same issue-width for the most case. This means

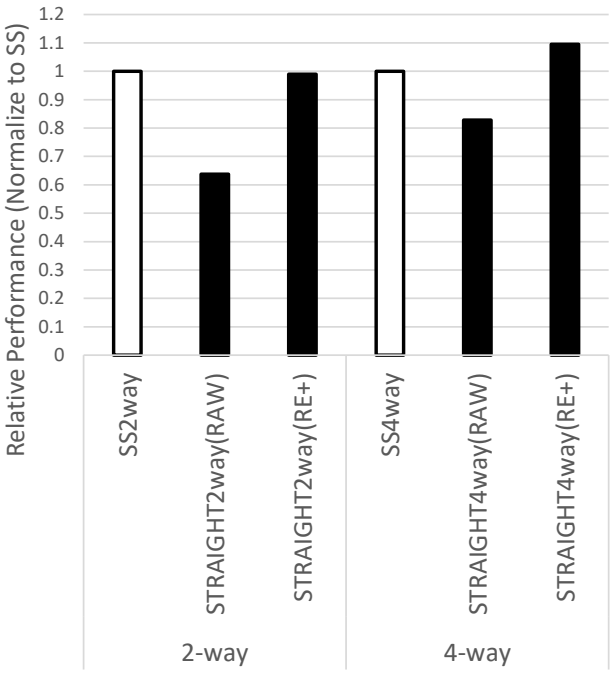

Fig. 14. Performance comparison of STRAIGHT and SS with TAGE branch predictor

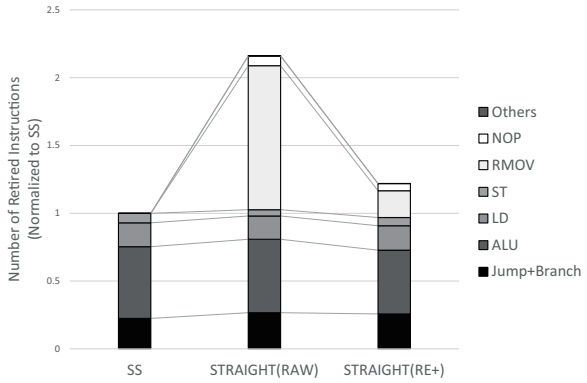

Fig. 15. Fraction of the Retired Instruction Type

that the increased 20% instructions can be executed in parallel, and they can utilize the empty issue slots that are reserved for the maximum ILP.

B. The Sensitivity of STRAIGHT design parameters

Figure 16 shows the source distance distribution in the cumulative graph. The distribution of source operand distances is measured for all retired instructions. The code that is generated with the uppermost distance limitation (1023) is used for this measurement. The actual maximum distance of the generated code is under 127 for each benchmark. The graph shows that most of the distances between producer instructions and consumer instructions are within 32. The graph also shows that almost 40% of the source operands in Dhrystone and 30% of those in CoreMark are the result of the previous instruction.

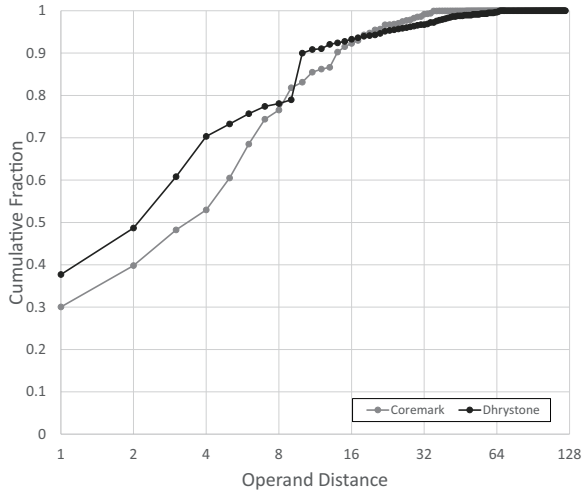

Fig. 16. Cumulative Fraction of the Source Distance

The result indicates that the short operand field is sufficient to represent the source distance. The distance limitation of STRAIGHT ISA is not severe unless it is set to below 31. This trend helps to reduce the STRAIGHT hardware resources for physical registers as well as to shrink the instruction length. We confirmed this by simulation with CoreMark. The performance degradation is only around 1% when the maximum distance is reduced from 1023 to 31.

C. Power Reduction

Figure 17 shows the result of the RTL power analysis. The relative powers of SS and STRAIGHT for various frequencies from the baseline to 4.0 times of the baseline are shown. In the graph the powers for the rename logic, the register file, and "other modules" are shown normalized to the corresponding power of SS when operated in the baseline frequency. "Other modules" involves the rest of the core, but caches, the buses, and the branch predictor are not included. The White bars indicate SS and the black bars indicate STRAIGHT. For "rename logic" of STRAIGHT, the power of the circuit for operand determination (fig 3) is shown as the counterpart.

As clearly shown in the graph, the power corresponding register renaming is almost removed in STRAIGHT. The power efficiency of STRAIGHT is supported because this power is known to one of the major factors of recent processors' power dissipation. The effect increases as the frequency. For the reference, the proportion of the renaming power is 5.7% to the other modules in this analysis. Because the scale of the analyzed processor is small, the proportion will increase when the wider front-end width is configured or check point is introduced to reduce the miss-recovery.

The power of the register file and other modules show the slight increase in STRAIGHT. The amount is under 18% for the register file and under 5% for other logics. This is the effect of the increased instructions per cycle (IPC) of STRAIGHT.

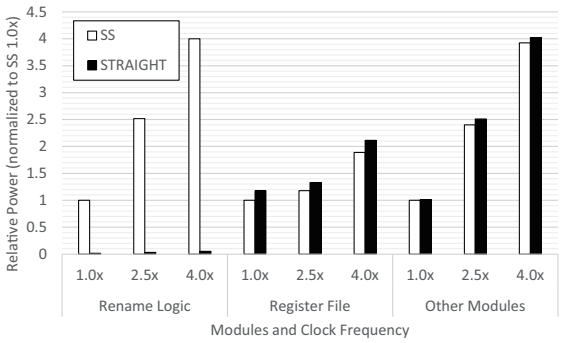

Fig. 17. Relative Power of SS and STRAIGHT with various clock frequency

For the aspect of energy, the performance improvement cancels this increase by the reduced execution time.

VII. Related Works

The single-thread performance is intensively researched by various approaches because of its significance. While the superscalar architecture achieves the sophisticated OoO execution with the support of register renaming, scheduling, deep pipelining, smart cache managements, and various predictors, its complexity and power consumption per instruction are major drawbacks.

The limitation and optimization of the superscalar pipeline depth are comprehensively explored around the turn of the century [16] [17] [18], which enforces researchers to explore novel ILP architectures as well as effective microarchitectures. Scalable ILP architectures such as clustered architectures are examples of the representative approaches [13] [19] [20]. Their basic idea is to separate the wide-issue core into several execution blocks to maintain the critical loops short regardless of the entire processor's scale.

The single-thread program often lacks parallelism to fully utilize such wide-issue cores. Speculative multithreading [21] [22] technologies are presented to supply many instruction streams to the core(s) by speculatively separating a single-threaded program into multiple threads. Slipstream processors [23] and Runahead execution [24] are similar but more drastic technologies that utilize helper threads only for the training of predictors and caches.

For the recent heterogeneous multicore processors, the technology for "OoO performance with in-order power" is desired for the total chip performance improvement. Kumar *et al*. [25] showed the potential of heterogeneous multicore architectures to improve the efficiency of a single-thread execution by switching the dedicated core dynamically. Different scale cores that share the same ISA are implemented in a chip, and an adequate core is selected depending on the ILP amount of the thread. It is known that the adequate configuration for the program changes in less than a thousand instructions [26]. Composite cores [27] or FXA [28] introduces both OoO and in-order mechanisms into a core to enable rapid switching. Mirage Core [12] virtually increases the number of OoO cores by transferring the scheduled instructions to the smaller in-order (OinO) cores.

As discussed, register renaming increases the OoO core's power consumption as well as it affect the clock frequency by introducing the critical-loop into the front-end pipeline. Safi *et al*. [10] proposes the two-stage pipelined renaming logic to reduce both the power and frequency overheads. Vajapeyam *et al*. [29] proposed the renamed trace cache (RTC) to cache renamed operands. Shioya *et al*. [11] introduced the distance representation into RTC to extend the caching target. Although conventional RMT is required for the RTC miss, both approaches have the advantage of compatibility.

TILE architectures target both the simple hardware and wider execution of single-thread performance. By introducing a specific ISA, the architecture maps the dataflow graph almost directly onto its ALU networks [30] [31]. A hybrid approach has been introduced to increase its viability [32].

The concept of STRAIGHT [7] reduces the required hardware and improves the performance and power efficiency by leveraging the compiler support. Unlike VLIWs [33], which also leverages compiler technologies, its scheduling is performed by the hardware, which enables dynamic and speculative ILP execution.

The instruction format of STRAIGHT has partially similar characteristics to instruction representations of dataflow architectures [34] [35] [36] in those an operand is represented as a connection between a producer instruction and a consumer instruction. Unlike RISC architectures, a register number (name) is not essential for both dataflow architectures and STRAIGHT; and a value in a register is automatically discarded in a short period. In STRAIGHT, this property leads to the simple hardware structure because temporal values can be maintained in a queue structure instead of a map structure.

The mechanism to pass the live variables across code blocks often becomes a challenge in such designs because the producer-consumer relation is not so simple in that case. STRAIGHT's approach is to statically prepare code that equalizes the relative distance between a consumer and producers so that the operand fetch behavior is fixed regardless of execution paths. The use of such relative distance is also seen in hybrid dataflow architecture SINAN [37] that passes arguments according to the relative position in the data segment to improve the inter-block communication. Contrary to typical dataflow architectures where a producer instruction specifies its destinations, the direction is inverse in STRAIGHT, that is, a consumer instruction refers to the source at its execution. This design can avoid the fanout problem but requires a broadcast mechanism in the scheduler.

Same as RISC architectures, the execution model of STRAIGHT inherits control flow guided by PC, which enables to exploit speculations for pumping instructions into the deep fast pipeline without waiting for actual execution. The two different approach, dataflow and control flow, are bound at the instruction level by RP that is incremented by the control flow and is used as a data pointer. Contrast to the typical dataflow architectures which require hierarchical structure of

inter-block and intra-block, this instruction level unification makes STRAIGHT a simple flat architecture.

VIII. Conclusion

Herein, we presented the practical STRAIGHT architecture by describing its ISA specification, microarchitecture organization, and compiler algorithm. By indicating source operands based on the distance from the producer instruction, the ISA guarantees that each register will be written once and then discarded in a fixed period. These characteristics enable the design of a novel microarchitecture that eliminates register renaming while maintaining a flexible OoO execution. The compiler algorithm that generates code for this novel ISA is first revealed. The key idea is to arrange the alive variables in a fixed order regardless of the control flow variation. The performance is evaluated with a cycle-accurate simulator; the power consumption is evaluated by the RTL power analysis. The same sized superscalar counterpart is also developed for the comparison.

The evaluation result shows that STRAIGHT achieves a higher performance with the simpler hardware. It shows 18.8% better performance than its superscalar counterpart. The low misprediction penalty delivered by STRAIGHT's simple hardware supports the superiority. The unique ISA of STRAIGHT possibly deteriorates the performance with the naive compiler, however, we also revealed that our redundancy elimination resolves the overhead. STRAIGHT shows better performance than the superscalar processor in the small configuration as well, which demonstrates its suitability for mobile platforms. The RTL power analysis shows that STRAIGHT reduces the power consumption by removing the power for renaming. Therefore the architecture achieves the performance improvement with the lower energy by the simple hardware. The results support that the architecture is a novel viable alternative for designing general purpose OoO processors.

Acknowledgment

Authors would like to thank Akifumi Fujita for the first implementation of our RTL description, and also thank Makoto Sahoda for the first implementation of the STRAIGHT simulator.

References

[1] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, 2008.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, 2012.

[3] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, 2010.

[4] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," in *Int. Symp. on Computer Architecture*, 2012.

[5] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global Value Numbers and Redundant Computations," in *Symp. on Principles of Programming Languages*, 1988.

[6] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting Equality of Variables in Programs," in *Symp. on Principles of Programming Languages*, 1988.

[7] H. Irie, D. Fujiwara, K. Majima, and T. Yoshinaga, "STRAIGHT: Realizing a Lightweight Large Instruction Window by using Eventually Consistent Distributed Registers," in *Int. Workshop on Challenges on Massively Parallel Processors*, 2012.

[8] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *Int. Symp. on Code Generation and Optimization*, 2004.

[9] A. Moshovos, "Checkpointing alternatives for high-performance, power-aware processors," in *Int. Symp. on Low Power Electronics and Design*, 2003.

[10] E. Safi, A. Moshovos, and A. Veneris, "Two-stage, pipelined register renaming," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 19, 2011.

[11] R. Shioya and H. Ando, "Energy efficiency improvement of renamed trace cache through the reduction of dependent path length," in *Int. Conference on Computer Design*, 2014.

[12] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Mirage cores: The illusion of many out-of-order cores using in-order hardware," in *Int. Symp. on Microarchitecture*, 2017.

[13] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," in *Int. Symp. on Computer Architecture*, 1997.

[14] S. Petit, R. Ubal, J. Sahuquillo, and P. Lopez, "Efficient register renaming and recovery for high-performance processors," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 22, 2014.

[15] A. Seznec, "A new case for the tage branch predictor," in *Int. Symp. on Microarchitecture*, 2011.

[16] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *Int. Symp. on Computer Architecture*, 2000.

[17] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays," in *Int. Symp. on Computer Architecture*, 2002.

[18] A. Hartstein and T. R. Puzak, "Optimum power/performance pipeline depth," in *Int. Symp. on Microarchitecture*, 2003.

[19] R. Canal, J. M. Parcerisa, and A. Gonzalez, "Dynamic Cluster Assignment Mechanisms," in *Int. Symp. on High-Performance Computer Architecture*, 2000.

[20] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster Dynamically-Scheduled, Superscalar Processors," in *Int. Symp. on Microarchitecture*, 2000.

[21] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," in *Int. Symp. on Computer Architecture*, 1995.

[22] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Trans. on Computers*, vol. 48, 1999.

[23] Z. Purser, K. Sundaramoorhy, and E. Rotenberg, "A Study of Slipstream Processors," in *Int. Symp. on Microarchitecture*, 2000.

[24] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, 2003.

[25] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: the potential for processor power reduction," in *Int. Symp. on Microarchitecture*, 2003.

[26] H. H. Najaf-abadi and E. Rotenberg, "Architectural contesting," in *Int. Symp. on High Performance Computer Architecture*, 2009.

[27] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Int. Symp. on Microarchitecture*, 2012.

[28] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Int. Symp. on Microarchitecture*, 2014.

[29] S. Vajapeyam and T. Mitra, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," in *Int. Symp. on Computer Architecture*, 1997.

[30] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," in *Int. Symp. on Microarchitecture*, 2001.

[31] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *Int. Symp. on Computer Architecture*, 2004.

[32] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Int. Symp. on Computer Architecture*, 2015.

[33] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. Computers*, vol. 33, 1984.
[34] J. Dennis and D. Misunas, "A preliminary architecture for a basic data-flow processor," in *Annual Symp. on Computer Architecture*, 1975.
[35] J. Dennis and G. Gao, "An efficient pipelined dataflow processor architecture," in *Conf. on Supercomputing*, 1988.
[36] Arvind and R. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Trans. on Computers*, vol. 39, 1990.
[37] S. Onder and R. Gupta, "Sinan - a forwarding multithreaded architecture," in *Int. Conf. on High-Performance Computing*, 1995.