IPSJ Transactions on System LSI Design Methodology Vol. 3 57–68 (Feb. 2010)

Regular Paper

Custom Instruction Generation for Configurable Processors with Limited Numbers of Operands ⋆1

Kenshu Seto†1 and Masahiro Fujita†2

This paper presents a novel framework to generating efficient custom instructions for common configurable processors with limited numbers of I/O ports in the register files and fixed-length instruction formats, such as RISCs. Unlike previous approaches which generate a single custom instruction from each subgraph, our approach generates a sequence of multiple custom instructions from each subgraph by applying high-level synthesis techniques such as scheduling and binding to the subgraphs. Because of this feature, our approach can provide both of the following two advantages simultaneously: (1) generation of effective custom instructions from Multiple Inputs Multiple Outputs (MIMO) subgraphs without any change in the configurable processor hardware and the instruction format, and (2) resource sharing among custom instructions. We performed synthesis, placement and routing of the automatically generated Custom Functional Units (CFUs) on an FPGA. Experimental results showed that our approach could generate custom instructions with significant speedups of 28% on average compared to a state-of-the-art framework of custom instruction generation for configurable processors with limited numbers of I/O ports in the register file and fixed-length instruction formats.

1. Introduction

Instruction-set extensible processors (or configurable processors) are becoming popular as basic processing elements for complex SoCs. Examples of such processors include Xtensa [20], MIPS with CorExtend [16], Altera Nios II [1], Toshiba MeP [22] and all of the examples are RISC-type processors. Application-specific custom instructions can be added to these processors to accelerate complex C/C++ applications in low design cost with flexibility to design changes. These custom instructions are implemented in the extra hardware called custom functional units (CFUs).

†1 Tokyo City University
†2 The University of Tokyo
⋆1 This paper is the journal version of the paper 18) with extended experimental results.

To develop efficient configurable processors quickly for various application programs, design tools that can automatically generate high-quality custom instructions and CFUs are necessary. To satisfy such requirements, a number of techniques for automatic custom instruction generation have been published, and some successful commercial tools exist such as XPRES [20] and CORXpert [12]. These techniques typically enumerate subgraphs from a given application program as candidates for custom instructions [5),6),10),19)]. A set of subgraphs that maximizes speedups are selected for custom instructions.

Pan, et al. reported that Multiple Inputs, Multiple Outputs (MIMO) subgraphs with more than 2 inputs and 1 output can achieve significant speedup compared to subgraphs with up to 2 inputs and 1 output [23]. By relaxing I/O constraints, more subgraphs become candidates for custom instructions and hence more speedups can be generally obtained. Configurable processors are, however, usually RISC processors and the numbers of operands for each custom instruction are limited because of the restrictions on (1) the number of register file I/O ports and (2) the length of instruction formats. Existing approaches [5),6),10)] for custom instruction generation had difficulties to realize general MIMO subgraphs as custom instructions for such processors, since these approaches had to select subgraphs that satisfied too restrictive I/O constraints, typically, 2 inputs and 1 output.

To tackle the problem, this paper proposes a novel approach that generates effective custom instructions from MIMO subgraphs for configurable processors with limited numbers of operands per custom instruction, such as RISC processors. Differently from previous work which generates a single custom instruction from each enumerated subgraph, our work generates a sequence of multiple custom instructions by applying high-level synthesis techniques such as scheduling and binding to the subgraphs. Because of this novelty, our approach, for the first time, provides both of following advantages at the same time:

(1) Our approach generates effective custom instructions from MIMO subgraphs without any change in the common RISC-type configurable processor architecture in terms of the processor hardware or the instruction format.
(2) Our approach enables resource sharing among custom instructions, so resource usage of CFUs can be significantly reduced.

Our approach can be used in conjunction with the recent approach that increases the limited data bandwidth by the use of local memories in CFUs [7].

This paper is organized as follows. Section 2 describes related work. Section 3 proposes the target architecture of our approach. Section 4 shows the overall flow and an illustrative example of our framework. In Section 5, experimental results and discussions are presented and Section 6 concludes this paper.

2. Related Work

One of the main issues that are addressed in this paper is the generation of custom instructions from general MIMO subgraphs. As explained in Section 1, custom instructions with larger numbers of operands usually provide better speedups. Some advanced configurable processor generation environments such as ASIP Meister [3], LISA [12] and XPRES Compiler [20] support generation of VLIW-based configurable processors which can execute MIMO custom instructions without difficulty. RISC-based configurable processors such as Xtensa 7 [20], MIPS with CorExtend [16], Altera Nios II [1], Toshiba MeP [22], however, allow custom instructions to access limited numbers of operands (for example, 2 inputs and 1 output), so those RISC-based processors have difficulties in executing MIMO custom instructions. Recently, several work have been published to tackle this problem.

Sun, et al. added special registers outside a configurable processor in order to make custom instructions read/write extra operands from/to the special registers [19]. Data transfer operations such as MOV instructions, however, are necessary between the processor register file and the special registers, so the execution cycles are wasted by the data transfer operations. Differently from the approach, the custom instructions generated by our approach perform both computation and data transfer simultaneously, so that the impact of the data transfer overhead is minimized.

Cong, et al. added shadow registers to an base processor to increase the number of inputs for custom instructions [11]. These shadow registers are written at the write-back stage of the base processor pipeline. An extra control bit is needed in the instruction format to specify whether the shadow registers are written or not for each instruction. Their work successfully increases the number of inputs operands for custom instructions. Their method, however, does not address the method to increase the number of output operands, so that it cannot handle MIMO subgraphs. In addition, their method cannot achieve data forwarding of the operands stored in the shadow registers, so that the speedup would be degraded because of the pipeline hazard.

Pozzi, et al. proposed an approach to generate custom instructions from MIMO subgraphs with unlimited numbers of I/Os for configurable processors with limited numbers of I/O ports in the register file, and achieved significant performance improvement [17]. Their approach introduces pipeline registers into the subgraphs and execute each subgraph in multiple cycles in order to satisfy the register file I/O port constraints. More specifically, their work enumerates all the scheduling of MIMO subgraphs under register file I/O port constraints, and obtains the scheduling results with the minimum latency and the minimum number of pipeline registers. Our work is similar to theirs, however, their work did not address the important issues to encode increased number of operands in the fixed-length instruction formats and requires significant changes in the base processor pipeline. These issues are addressed in this paper.

Jayaseelan, et al. proposed an architectural framework to generate custom instructions from subgraphs with up to four input and one output operands by exploiting forwarded operands in the base processor pipeline [14]. To maximally use the forwarded operands in custom instructions and reduce extra MOV instructions, their work uses ILP-based instruction scheduling. Their approach could successfully generate custom instructions that improved the speedup for RISC-type processors without any change to the RISC-type instruction format. Their approach, however, requires a simple change in pipeline control logic to provide forwarded operands correctly to custom functional units (CFUs) in case of cache misses. In addition, their approach cannot generate custom instructions from MIMO subgraphs.

Another issue that is tackled by our approach is resource sharing among custom instructions. This issue is very important especially when many custom instructions or custom instructions that use expensive resources are generated. Brisk, et al. proposed a method to share functional units among subgraphs and reduced the hardware area significantly [8]. Zuluaga, et al. builds on the work [8] to

take clock period constraints into consideration while resource sharing [24]. The previous approaches do not address the important issue of increasing limited data-bandwidth, however, our approach can perform both resource sharing and enhancement of data-bandwidth of custom instructions simultaneously.

One of the main features of our work is that our approach typically generates a sequence of multiple custom instructions from each subgraph, while all previous approaches, as far as the authors know, generates a single custom instruction from each subgraph. Because of this feature, our work brings the following advantages simultaneously: (1) our approach requires no modification to the existing RISC-type configurable processors and instruction formats in order to generate custom instructions from MIMO subgraphs, and (2) our approach can share resources among custom instructions, so the resource usage in the CFUs can be significantly reduced. These advantages will be demonstrated by the experimental results presented in Section 5.

3. Target Architecture

Before describing our method to generate custom instructions, we first explain the target processor architecture on which our method is based.

3.1 Instruction Encoding

Our approach can use common RISC-type instruction formats without any change in order to implement custom instructions. An example of such formats is shown in **Fig. 1**. It is a typical RISC instruction format with three register operands that is actually used in Nios II [2]. The first field (OPCode) is a 6-bit op-code field and has a fixed value for custom instructions. The next field n (CustomInstKind) specifies which custom instruction to execute. Since the field has 8 bits, we can add at most 256 different custom instructions. When we need to add more than 256 custom instructions, we can use an external configuration register that works as the extended CustomInstKind field that supplementarily specifies those extra custom instructions. To use the extra custom instructions, configurable processors need to place appropriate values in the configuration register. Rs and Rt represent the indices of source registers and Rd represents the index of a destination register. The 14th, 15th, 16th bits are used to control the accesses to the operands. As shown in the format, a custom instruction can read at most two operands and write at most one operand from/to the register file in the configurable processor. The custom instructions in Nios II are represented by the following assembly format:

```
custom n, Rd, Rs, Rt
```

This assembly format will be used in Fig. 5 (h) that is explained in Section 4.2.

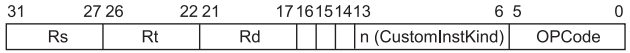

Fig. 1 Instruction format for custom instructions.

3.2 Configurable Processor

Figure 2 illustrates an example configurable processor architecture for our custom instruction generation. In this paper, we assume that the base configurable processor has a 5-stage, single-issue, in-order pipeline. As shown in Fig. 2, the hardware for custom instructions is attached to the pipeline of the configurable processor. If the fetched instruction is a custom instruction, it is decoded not only by the decoder in the configurable processor, but also by the Custom Instruction Decoder (CID). Our approach for custom instruction generation does not require any change in the base configurable processor, as will be illustrated in Section 4.2.

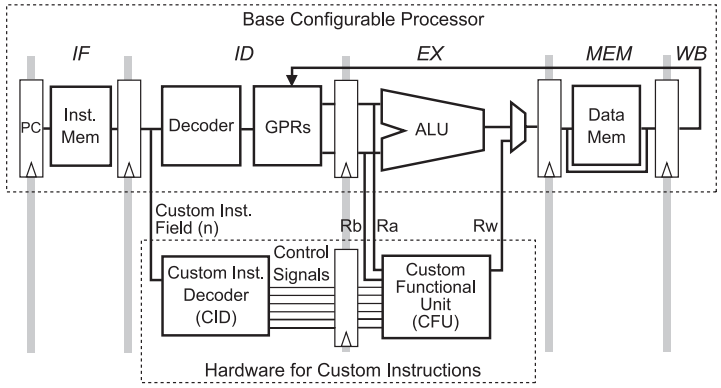

Fig. 2 Target architecture.

3.3 Hardware for Custom Instructions

As shown in Fig. 2, the hardware for custom instructions consists of the Custom Instruction Decoder (CID) and Custom Functional Unit (CFU). All the custom instructions generated from a given application program or even from a set of application programs can be implemented by a single CID and a single CFU. The control signals decoded from the custom instruction field (n in Fig. 1) by the CID are sent to the CFU via the pipeline register. As shown in Fig. 2, the CFU can read at most two operands Ra, Rb, and write at most one operand Rw from/to the register file of the base processor in exactly the same way as ALU instructions in the base processor. These operands are specified in the instruction encoding of Fig. 1 as Rs, Rt and Rd respectively. Although not explicitly encoded in the instruction format in Fig. 1, the custom instructions generated by our work access external special registers in the CFU implicitly, as explained in Section 4.2.

4. Our Approach

In this section, we present our method that can generate effective custom instructions for RISC-type configurable processors with limited operands in custom instructions from general MIMO patterns and that enables resource sharing at the same time by utilizing high-level synthesis. As previously mentioned, our approach uses the target architecture shown in Fig. 2.

4.1 Overall Flow

Figure 3 shows the overview of our approach. The inputs to the flow are (1) application program that are to be accelerated with custom instructions, (2) input data for the application program, (3) a component library in which the information on area, delay of each function unit is described, and (4) constraints for custom instructions, for example, on the number of source and destination operands, the clock period, the number of multipliers or the number of custom instructions, etc. The outputs of the flow are (1) a synthesizable RTL description for a Custom Functional Unit (CFU) and a Custom Instruction Decoder (CID) and (2) an assembly code with custom instructions.

As shown in Fig. 3, the application program is parsed by the compiler frontend so that optimized intermediate representation (IR) of the application program, namely, Control Flow Graphs (CFGs) and Basic Blocks (BBs) are generated.

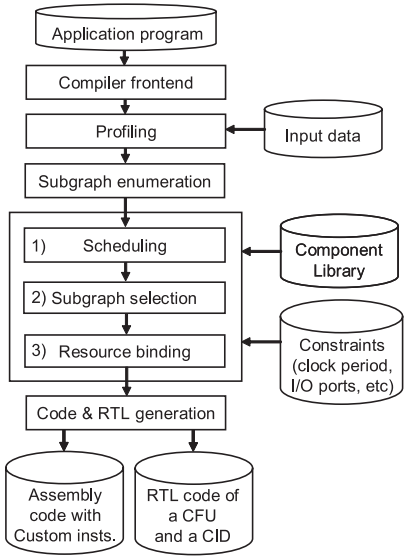

Fig. 3 Overview of our approach.

With input data, we perform profiling of the application program to obtain information such as the execution count of each BB. Using the profiling information, we find critical BBs and then generate data-flow graphs (DFGs) for the BBs. Here, the critical BB is the BB whose execution time occupies the large part of the total execution time. Note that the resulting custom instructions by our approach may change significantly when the input data change, because critical BBs may change by using different input data. We implemented the flow in Fig. 3 on top of LLVM compiler infrastructure [21], and used the profiling function in LLVM to obtain the execution count of each BB with the input data.

From the bottleneck DFGs, we enumerate subgraphs. Our approach can generate custom instructions not only from connected subgraphs but also disconnected subgraphs. These subgraphs may have any numbers of I/Os. We use an efficient ILP-based subgraph identification algorithm [4] without imposing any I/O constraints for subgraphs. After the subgraphs are identified, the following three steps (scheduling, subgraph selection and resource binding) are performed.

4.1.1 Scheduling

We schedule all the enumerated subgraphs under I/O constraints. For the scheduling algorithm, we can use any scheduling algorithms in high-level synthesis [13] as long as they can satisfy the imposed I/O constraints. In this paper, we use list scheduling.

After scheduling, we obtain the scheduling latency (or the number of execution cycles) $H(G)$ of each bottleneck subgraph G. Our approach generates a single-cycle custom instruction from each control step of the scheduled subgraphs. Thus, G is executed by a sequence of $H(G)$ custom instructions. When a subgraph G is executed in $S(G)$ cycles without custom instructions, then the gain by using custom instructions in terms of the reduction in the number of execution cycles for G is given by

$$Gain(G) = \frac{S(G)}{H(G)} \quad (1)$$

4.1.2 Subgraph Selection

From the scheduled subgraphs, we select a set of subgraphs for custom instruction generation so that the total gain is maximized while the number of custom instructions does not exceed the maximum allowed number of custom instructions (For example, 256). Our approach uses a simple greedy algorithm as shown in **Fig. 4** which is similar to the previous work [9]. In Fig. 4, procedure $FindBestGain()$ returns the subgraph with the best gain from the set of subgraphs Γ using formula (1), and procedure $NumberOfCustomInsts()$ returns the number of custom instructions generated from a set of subgraphs.

4.1.3 Resource Binding

After subgraph selection, we bind operations and variables in the selected subgraphs to I/O ports, functional units and registers. In the binding, we can choose to perform resource sharing or not to for resources except I/O ports of the register file. We always have to share I/O ports. When we choose not to share resources, unique functional unit or register is allocated for each operation or variable in the selected subgraphs, and each of the operation or variable is bound to the corresponding unique functional unit or register. When we share resources, we can use any binding algorithms in high-level synthesis [13]. In this paper, we use left edge algorithm [15] that minimizes the numbers of I/O ports, functional units and registers.

Algorithm Subgraph Selection

```
Γ: Set of scheduled subgraphs
Γ′: Selected subset of Γ
N_max: Maximum number of custom instructions
O(G): Set of subgraphs that overlap with a subgraph G
G_best: A subgraph that is returned by FindBestGain() procedure
Γ′ ← φ
while (Γ ≠ φ) {
    G_best ← FindBestGain(Γ)
    if (NumberOfCustomInsts(Γ′ ∪ {G_best}) ≤ N_max) {
        Γ′ ← Γ′ ∪ {G_best}
        Γ ← Γ − O(G_best)
    }
    Γ ← Γ − {G_best}
}
return Γ′
```

Fig. 4 Algorithm used in subgraph selection.

After binding, we have all necessary information to generate hardware for custom instructions, namely, a CFU and a CID. In addition to the generation of RTL for the CFU and the CID, parts of software (assembly code) are replaced by the sequences of custom instructions.

4.2 An Illustrative Example

Figure 5 illustrates our approach with a simple example. Figure 5 (a) is an example of bottleneck basic blocks (BBs). It is translated to the data flow graph (DFG) shown in Fig. 5 (b) where t1, ..., t7 represent temporary variables. In this paper, we use the subgraph identification algorithm [4] which identifies subgraphs with unlimited numbers of I/Os from the DFG by excluding memory operations as in Fig. 5 (c). An example G of such subgraphs is shown in Fig. 5 (d) and it is a MIMO subgraph with 3 inputs and 2 outputs. For simplicity, we explain our approach by using a connected subgraph, however, our approach can handle disconnected subgraphs without difficulty and these disconnected subgraphs are executed in parallel with the proposed custom instructions.

The filled rectangle nodes I1, I2 and I3 represent input variables (corresponding

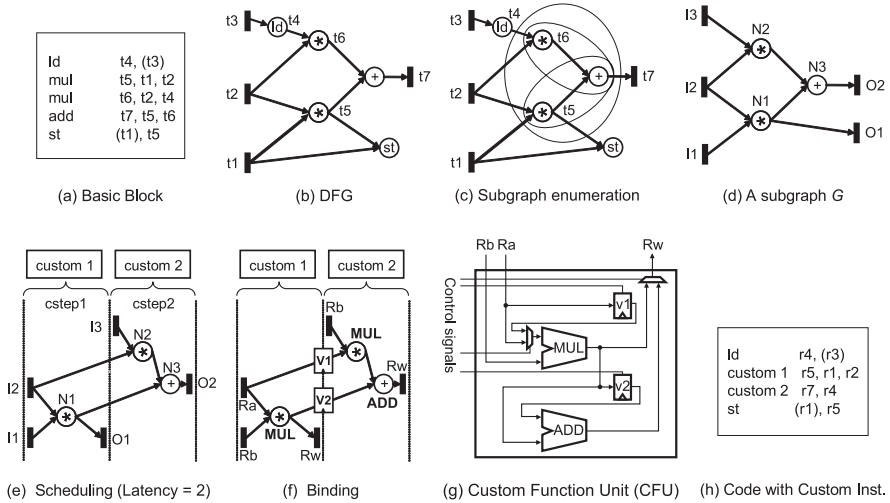

Fig. 5 Our approach illustrated with an example.

to register file read operations) and O1 and O2 represent output variables (corresponding to register file write operations). It contains three internal operations N1, N2 and N3.

In this paper, we schedule the MIMO subgraph under the I/O constraints of 2 inputs and 1 output per clock cycle as shown in Fig. 5 (e), and make a single-cycle custom instruction from each control step (cstep) of the scheduled subgraph. We assumed that the sequence of multiplication and addition operations can be executed in the target clock period. Our approach can generate custom instructions from multi-cycle operations as long as they are executed by pipelined functional units. In this example, two custom instructions (`custom 1` and `custom 2`) are made from the two control steps (cstep1 and cstep2) respectively. As a result, the MIMO subgraph is executed in 2 cycles by the sequence of the two custom instructions. Since the subgraph is executed in 3 cycles by the base instructions of the configurable processor, the gain is $Gain(G) = \frac{3}{2} = 1.5$.

For simplicity in illustration, assume that only one subgraph shown in Fig. 5 (e) is selected for custom instructions. Figure 5 (f) shows a binding result for the selected subgraph. Each input variable is bound to a register file read access (either Ra or Rb) and each output variable is bound to a register file write access (Rw). Temporary variables that are live across different csteps are bound to the special registers in the CFU. For example, the temporary variable t2 and t5 are bound to the special registers v1 and v2 respectively. In general, such special registers can be shared among different temporary variables that are written by different custom instructions. In the case of Fig. 5 (f), this does not happen, since t2 and t5 have overlapping lifetime. The multiplication operations (N1 and N2) are bound to the same multiplier (MUL) since they are executed in different control steps. Because of the sharing of MUL, an input multiplexer is required at the input of MUL, as shown in Fig. 5 (g). The addition operation (N3) are bound to an adder (ADD).

Figure 5 (g) is the resulting CFU by our work. As stated previously, the control signals are generated by decoding the custom instruction specifier n in Fig. 1 using the CID, and the control signals are used as the control inputs for the multiplexers and write-enable signals for the special registers in the CFU.

Figure 5 (h) is the resulting assembly code with the custom instructions. The operations in the selected subgraph are replaced with the sequence of custom instructions. After the replacement, register allocation is performed for the temporary variables in the code generation phase of the compiler. In Fig. 5 (h), t1, t2, t3, t4, t5, r7 are allocated to the registers r1, r2, r3, r4, r5, r7 in the base processor, respectively. t2, t5 are also assigned to the external registers v1 and v2, and t6 is reduced to a bus wire because of the chaining.

In the assembly code, `custom 1` implicitly moves r2 to v1 and writes v2, and `custom 2` implicitly read v1 and v2. These special registers v1 and v2 are not explicitly encoded in the custom instruction format, however, and are implicitly specified in the custom instruction specifier n in Fig. 1, so that our approach can use the RISC-type instruction format in Fig. 1 while increasing the data bandwidth of custom instructions. The reordering of custom instructions by compiler optimizations must be prohibited, since that may change the execution results.

4.3 Execution of Custom Instructions

Figure 6 (a) shows the execution of the generated assembly code with the custom instructions in the processor pipeline. Custom instructions generated by our

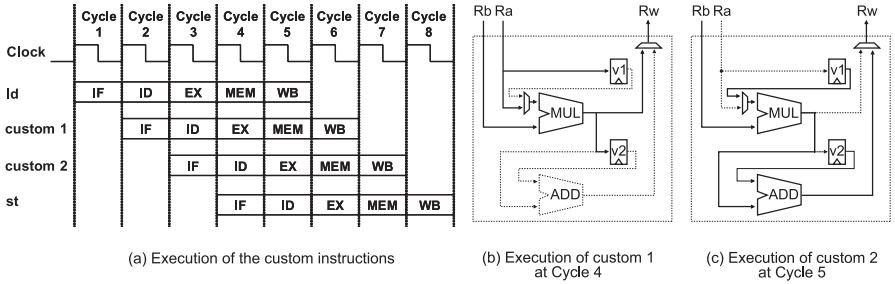

Fig. 6 Execution of custom instructions in processor pipeline.

approach can use the existing forwarding unit in the configurable processor, so that the pipeline hazards are minimized as in the same way as the base instructions of the configurable processor. As shown in Figs. 5 (h) and 6 (a), `custom 2`, for example, read the register r4 that are written by the ld instruction before the write-back (WB) stage of the ld instruction without stalling the pipeline by using the data forwarding unit.

Figures 6 (b) and (c) show how `custom 1` and `custom 2` are executed by the CFU. As shown in Figs. 6 (b) and (c), the custom instructions generated by our approach (`custom 1`) not only perform computation such as multiplication, but also perform data transfer between the register file in the configurable processor and the special register (v1) in the CFU simultaneously. In the execution cycle of `custom 2`, the special registers v1 and v2 are read immediately after the write access to v1 and v2 at the execution cycle of `custom 1`. Since the execution of `custom 1` and `custom 2` do not occur in the same cycle, the multiplier (MUL) is chosen to be shared.

There are two caveats in the generated custom instructions: (1) When a cache miss occurs while custom instructions execute, the processor pipeline is stalled in the same way as base instructions stall pipeline in case of cache misses, and (2) any pair of interrupt routines, A and B, that may be executed concurrently cannot use custom instructions that share external registers in the CFU to avoid corruption of data in the external registers.

5. Experimental Results

In this section, we evaluate our approach in terms of performance and resource usage. In particular, we compare the evaluation results by three approaches: (1) previous work by Jayaseelan, et al. [14], (2) our work without resource sharing, and (3) our work with resource sharing. In our work, we can choose to perform resource sharing or not. In the evaluation of performance, we use not only the execution cycle, but also the clock frequency (or equivalently, critical path delay) of the configurable processor with custom instructions, since both of them are important in performance evaluation.

5.1 Experimental Setup

We implemented our approach in Fig. 3 in C++ on top of an existing compiler framework. We estimated the execution cycles of basic blocks (BBs) and programs statically by the numbers of instructions weighted by the execution cycles of each instruction. We defined critical BBs as the ones whose numbers of execution cycles were more than 0.5% of the total number of execution cycles. We used the subgraph identification algorithm [4] without imposing the I/O constraints for subgraphs. We excluded memory access operations, floating-point operations and division operations from subgraphs. For the identified subgraphs, we performed list scheduling [13] with the resource constraints of up to 2 inputs and 1 output on I/O accesses. After list scheduling, we selected subgraphs using the algorithm in Fig. 4, and carried out left edge binding of I/O ports for the selected subgraphs followed by the generation of the Verilog-HDL file of the Custom Functional Unit (CFU). We did not generate the Verilog-HDL file of the Custom Instruction Decoder (CID), since it is unlikely to be on the critical path of the configurable processor system and the resource usages of the CIDs are most likely much smaller than those of CFUs.

We used Quartus II 8.1 Subscription edition [1] to perform synthesis, placement and routing of the CFUs and to obtain the information on critical path delays and resource usages of the CFUs. We used Altera Stratix II (EP2S180F1508C3) for a target device. We chose 6.5 ns as the target clock period for the configurable processor system, since the critical path delay of the configurable processor Altera Nios II/f with 16 KB I-Cache, 16 KB D-Cache, and an SDRAM controller was

Table 1 Benchmark programs.

Name	Description	insts.	TBB	BB	%
rawcaudio	ADPCM audio encoder	110	17	7	99.9
rawdaudio	ADPCM audio decoder	100	18	9	99.9
dct	Forward 8×8 DCT	258	10	2	99.2
idct	Inverse 8×8 DCT	338	27	18	99.1
gouraud	Gouraud shading	51	4	1	100
fft	Complex FFT	148	10	4	99.5
viterbi	Viterbi trellis decoder	188	4	1	99.3
sha	Secure hash algorithm	1,493	28	4	99.7
djpeg	JPEG image decoder	244,554	3,186	12	90.9

6.5 ns. The resource usage of the configurable processor system was 4,463 ALUTs (Adaptive LUTs), 4,259 registers and 8 8-bit multipliers.

In this experiment, we used the following formula to compute the speedups by custom instructions.

$$Speedup = \frac{Cycle_{nocust} \times CP_{nocust}}{Cycle_{cust} \times CP_{cust}} \quad (2)$$

where $Cycle_{nocust}$ and $Cycle_{cust}$ are the numbers of execution cycles when a benchmark is executed without and with custom instructions, respectively, and CP_{nocust} and CP_{cust} are the critical path delays of the configurable processor system without and with custom instructions, respectively. As mentioned above, we chose CP_{nocust} to be 6.5 ns. We denote the critical path delay of a CFU as CP_{cfu}, and we used the following formula to obtain CP_{cust}:

$$CP_{cust} = max\{CP_{nocust}, CP_{cfu}\} \quad (3)$$

Table 1 shows benchmark programs we used in this experiments. We picked up the benchmarks from MediaBench (djpeg, rawdaudio, rawcaudio), MiBench (sha with loops unrolled). Other benchmarks (dct, idct, gouraud, fft and viterbi) are performance intensive loops in digital signal processing. In Table 1, the first column (Name) represents the name of each benchmark, the second column (Description) briefly describes each benchmark, the third column (insts.) is the number of instructions in each benchmark, the fourth column (TBB) shows the total number of BBs in each benchmark, and the fifth column (BB) presents the number of critical BBs in each benchmark. The sixth column (%) expresses the number of execution cycles spent in the critical BBs as a percentage of the total number of execution cycles.

5.2 Comparison with Previous Work

We compare our work with the previous work [14], since it is the state-of-the-art framework that generates efficient custom instructions for the configurable processor with limited I/O ports in the register file and limited operands in the instruction format. The previous work exploits the forwarding mechanism in RISC processors, so that any subgraph with up to 4-input and 1-output can directly become a custom instruction for RISC processors with two source and one destination operands. More specifically, we compare custom instructions generated by our work with those generated for the architecture with four read ports in the register file and enough space to encode these operands in the instruction format. This architecture and custom instructions which is compared with our work provides the best speedup that can be achieved by the previous work [14] and the same resource usages in the CFUs as the work [14]. In this experiment, we did not perform resource sharing except I/O ports in the binding step to maximize the speedups by our work.

Table 2 summarizes the results of the comparison. In the table, codesize shows the code size and cycle shows the number of execution cycles. In the second column and the third column, the code size and the number of execution cycles ($Cycle_{nocust}$ in (2)) in the case of the base processor without custom instructions are shown respectively. In the table, sg represents the number of selected subgraphs, N shows the total number of operations executed in all the custom instructions, and ci represents the number of custom instructions generated. In case of the previous work [14], the number of custom instructions (ci) is exactly equal to the number of selected subgraphs (sg), since each subgraph (with up to 4-input and 1-output) corresponds to a custom instruction. lut, reg, and mul shows the number of ALUTs (adaptive look-up tables), the number of registers and the number of 8-bit dedicated hardware multipliers used in the Stratix II FPGA, respectively. In the table, columns cp [ns] represent the critical path delays of the CFU in the unit of nano seconds, and these numbers correspond to CP_{cfu} in formula (3). In the same columns, the numbers shown in parentheses are the critical path delays of the system including both the configurable processor and the CFU, and these numbers correspond to CP_{cust} in formula (2). cycle in the 10th and 20th columns represent the numbers of execution cycles of each

Table 2 Comparison between previous work [14] and our work.

benchmark	Original		Previous work (ideal case of [14])								Our work without resource sharing									
						Resource usage		Performance							Resource usage			Performance		
	codesize	cycle	codesize	ci	N	lut	mul	cp [ns]	cycle	s-up	codesize	sg	ci	N	lut	reg	mul	cp [ns]	cycle	s-up
rawcaudio	110	9.28 M	94	8	24	380	0	6.3 (6.5)	6.93 M	1.34	89	4	17	38	643	655	0	5.3 (6.5)	6.19 M	1.50
rawdaudio	100	608.4 M	87	5	18	309	0	5.8 (6.5)	447.8 M	1.36	89	4	9	20	278	233	0	4.6 (6.5)	474.5 M	1.28
dct	258	2,024	180	30	108	955	224	11.7 (6.5)	1,528	1.32	132	2	26	152	1,523	1,719	224	6.7 (6.7)	1,016	1.93
idct	338	2,511	262	30	106	870	320	11.5 (6.5)	1,989	1.26	229	2	36	145	1,803	3,232	320	6.0 (6.5)	1,639	1.53
gouraud	51	19.5 M	29	5	27	190	24	4.8 (6.5)	8.50 M	2.29	25	1	7	33	445	562	24	6.1 (6.5)	6.50 M	3.00
fft	148	6,042	128	8	28	304	104	4.2 (6.5)	5,082	1.19	118	5	33	63	909	1,116	120	6.0 (6.5)	4,629	1.31
viterbi	188	435	138	25	75	121	0	4.5 (6.5)	335	1.30	108	4	36	116	1,018	844	0	5.5 (6.5)	275	1.58
sha	1,493	162.6 M	1,491	1	3	32	0	0.8 (6.5)	156.2 M	1.04	547	3	107	1,053	14,570	19,273	0	5.7 (6.5)	110.7 M	1.47
djpeg	244,554	4.94 M	244,450	69	173	1,626	272	11.5 (6.5)	4.33 M	1.14	244,424	9	56	186	2,939	3,318	280	5.8 (6.5)	4.15 M	1.19
average	-	-	-	-	-	-	-	-	-	1.36	-	-	-	-	-	-	-	-	-	1.64

benchmark, and correspond to $Cycle_{cust}$ in (2). s-up shows the speedup results computed by formula (2). On average, the ideal speedup of the work [14] was 1.36, and the speedup achieved by our approach without resource sharing was 1.64.

The code sizes with custom instructions were consistently smaller than those without custom instructions, since a custom instruction contained multiple base instructions. Mostly, the code sizes with our work were smaller than those with the previous work [14]. This was because more base instructions were contained in the set of generated custom instructions by our approach than in those by the previous approach.

The number of custom instructions generated by our work (ci) is usually larger than the number of custom instructions generated by the previous work (sg), since our work typically generates a sequence of multiple custom instructions from each subgraph. The number of custom instructions generated by our work was at most 107 (for sha benchmark), and all of the custom instructions can be implemented in Nios II, since the instruction format allows up to 256 custom instructions. By comparing columns N in Table 2, we see that our work enables more operations to be executed in custom instructions compared to the previous work.

The columns lut in Table 2 show that the custom instructions generated by our work use more ALUT resources compared to the previous work. This is because our work executes more operations by custom instructions compared to the previous work as stated above, so that more functional units are allocated in CFUs. In addition, the custom instructions generated by our work require registers to store temporary variables in external special registers in order to increase data-bandwidth, whereas the custom instructions generated by the previous work [14] do not use external registers. The numbers of 8-bit multipliers used in our work were almost the same as those used in the previous work.

Now, we compare the custom instructions generated by the previous work and our work in terms of performance improvement. As shown in formula (2), performance is the product of the number of execution cycles required to execute a benchmark, and the critical path delay of the configurable processor system. In the previous work, the critical path delays of the CFUs (cp [ns]) were smaller than the critical path delays of the configurable processor (6.5 ns) except dct, idct and djpeg as shown in Table 2. Thus, the latency of all the custom instructions for the benchmarks except dct, idct and djpeg was one cycle. The CFUs of dct, idct and djpeg contained many 8-bit dedicated hardware multipliers. Since these multipliers were placed in special and limited locations in the Stratix II device, many use of the multipliers caused unexpected interconnection delays, so that the critical path delays of some custom instructions were longer than estimated. Such custom instructions were executed as multi-cycle custom instructions so that the critical path delay of the whole system was not degraded at all. Therefore, the critical path delays CP_{cust} of the configurable processor system with custom in-

Table 3 Effect of resource sharing.

benchmark	Our work with resource sharing				
	Resource usage (ratio)			Performance	
	lut (ratio)	reg (ratio)	mul (ratio)	cp [ns]	s-up
rawcaudio	714 (1.11)	269 (0.41)	-	5.8 (6.5)	1.50
rawdaudio	361 (1.30)	110 (0.47)	-	4.7 (6.5)	1.28
dct	2,285 (1.50)	516 (0.30)	65 (0.29)	9.1 (9.1)	1.42
idct	2,398 (1.33)	646 (0.20)	189 (0.59)	7.9 (7.9)	1.26
gouraud	659 (1.48)	281 (0.50)	16 (0.67)	7.7 (7.7)	2.53
fft	1,218 (1.34)	268 (0.24)	32 (0.27)	8.5 (8.5)	1.00
viterbi	1,038 (1.02)	160 (0.19)	-	5.8 (6.5)	1.58
sha	8,305 (0.57)	2,313 (0.12)	-	6.9 (6.9)	1.38
djpeg	4,262 (1.45)	465 (0.14)	31 (0.11)	11.0 (11.0)	0.70
average	- (1.23)	- (0.29)	- (0.39)	-	1.41

structions generated by the previous approach was 6.5 ns for all benchmarks, so the CFUs did not deteriorate the critical path delays of the overall system. By using our work without resource sharing, only the CFU of dct benchmark had slightly longer critical path delays of 6.7 ns than that of the system (6.5 ns). The reason for the long critical path delay of dct in our work is the same as that for dct, idct and djpeg in the previous work. For all benchmarks except dct, our work could generate the CFUs whose critical path delays are shorter than that of the configurable processor system (6.5 ns). The numbers of execution cycles with our approach were significantly fewer than those with the previous approach except rawdaudio benchmark. In the rawdaudio, the previous work could achieve better speedup, since it could generate a single-cycle custom instruction from a bottleneck subgraph with three inputs, whereas our approach generated a sequence of two custom instructions from the same subgraph. In summary, our work generated custom instructions whose performance is significantly (28% on average) better than the state-of-the-art [14].

5.3 Effect of Resource Sharing

Table 3 summarizes the results of our work when we shared not only I/O ports as in Section 5.2 but also multipliers and registers using left edge binding algorithm [13] in the binding step. The numbers of registers and multipliers were the minimum, since left edge algorithm was used. The number of selected subgraphs (sg), the number of custom instructions generated (ci), and the total number of operations executed in all the custom instructions (N) are exactly the same with and without resource sharing, so these numbers are not shown in Table 3. Resource usage in the table shows the resource usage by our work with resource sharing, and ratio in the parenthesis shows a ratio of resource usage by our work with resource sharing to the resource usage by our work without resource sharing which is shown in Table 2. From Table 3, we can see that our work with resource sharing could reduce both the number of ALUTs and the number of registers for sha benchmark. On average, the number of ALUTs increased by 23% when resources were shared, since multiplexers were added in front of the registers and multipliers. Significant reduction in the numbers of registers and multipliers (71% and 61%, respectively) were observed because of the resource sharing. For three benchmarks (rawcaudio, rawdaudio, and viterbi), our approach with resource sharing could satisfy the system clock constraint (6.5 ns) while reducing the number of registers significantly. For the remaining six benchmarks out of nine, the critical path delays of the CFUs were larger than the critical path delay of the configurable processor system (6.5 ns), so we can see that it is better not to perform resource sharing, if we need highest performance.

To know the effectiveness of resource sharing on area reduction, we roughly estimated the total area of CFUs in terms of 2-input NAND gates by multiplying the numbers of ALUTs (lut), the numbers of registers (reg) and the numbers of 8-bit hardware multipliers (mul) in Tables 2 and 3 by 12, 10 and 400, respectively. We had to use the rough estimation for the total area because Verilog-HDL files generated by our current tool contained instances of LPMs (Library of Parametrized Modules) from Altera [1] and could not directly compiled to ASICs. The result of rough estimation showed that our work with resource sharing reduced the area by 39% on average in terms of 2-input NAND gates compared to our work without resource sharing.

Our work with resource sharing achieves moderate speedup (1.41%) compared to the previous work [14] (1.36%) with significant increase in the number of ALUTs. Our work with resource sharing also requires registers, while the previous work [14] does not. Our work with resource sharing, however, can be effective when the number of available multipliers is limited and there are many multiply operations in the target application, since our work with resource sharing requires

less multipliers compared to the previous work [14].

6. Conclusions

We proposed a technique to generate custom instructions for configurable processors with limited register file I/O ports and fixed-length instruction formats. As a unique feature of our approach, it makes a custom instruction from each control step of the scheduled subgraphs, and can perform resource sharing when necessary in the binding step. Because of the feature, our approach provides both of the following advantages simultaneously: (1) Generation of effective custom instructions from general MIMO subgraphs without changing the pipeline and the instruction format of the configurable processors, (2) Generation of a single, area-efficient Custom Functional Unit (CFU) for a set of custom instructions in which resources are shared among different custom instructions. Experimental results showed that our approach without resource sharing could generate custom instructions with significant speedups of 28% on average compared to the state-of-the-art previous work of custom instruction generation for common configurable processors with limited register file I/O ports and fixed-length instruction formats. Therefore, we can conclude that our approach is a promising way of generating performance-effective custom instructions for commonly used RISC-type configurable processors with limited register file I/O ports and fixed-length instruction formats.

References

1) Altera Corp.: http://www.altera.com
2) Altera Corp.: Nios II Custom Instruction User Guide.
3) ASIP Solutions, Inc.: http://www.asip-solutions.com/en/products.html
4) Atasu, K., Dundar, G. and Ozturan, C.: An Integer Linear Programming Approach for Identifying Instruction-Set Extensions, *Proc. 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (*CODES-ISSS*), pp.172–177 (2005).
5) Atasu, K., Pozzi, L. and Ienne, P.: Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints, *Proc. 40th Design Automation Conference* (*DAC*), pp.256–261 (2003).
6) Biswas, P., Banerjee, S., Dutt, N., Pozzi, L. and Ienne, P.: ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement, *Proc. Design, Automation, and Test in Europe* (*DATE*), pp.1246–1251 (2005).
7) Biswas, P., Dutt, N.D., Pozzi, L. and Ienne, P.: Introduction of Architecturally Visible Storage in Instruction Set Extensions, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.26, No.3, pp.435–446 (2007).
8) Brisk, P., Kaplan, A. and Sarrafzadeh, M.: Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs, *Proc. 41st Design Automation Conference* (*DAC*), pp.395–400 (2004).
9) Clark, N., Zhong, H. and Mahlke, S.: Processor Acceleration Through Automated Instruction Set Customization, *Proc. International Symposium on Microarchitecture*, pp.129–140 (2003).
10) Clark, N.T., Zhong, H. and Mahlke, S.A.: Automated Custom Instruction Generation for Domain-Specific Processor Acceleration, *IEEE Trans. Comput.*, Vol.54, No.10, pp.1258–1270 (2005).
11) Cong, J., Han, G. and Zhang, Z.: Architecture and Compilation for Data Bandwidth Improvement in Configurable Embedded Processors, *Proc. 2005 IEEE/ACM International Conference on Computer-Aided Design*, pp.263–270 (2005).
12) CoWare, Inc.: http://www.coware.com
13) Gajski, D.D., Dutt, N.D., Wu, A.C.-H. and Lin, S.Y.-L.: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).
14) Jayaseelan, R., Liu, H. and Mitra, T.: Exploiting forwarding to improve data bandwidth of instruction-set extensions, *Proc. 43rd Design Automation Conference* (*DAC*), pp.43–48 (2006).
15) Kurdahi, F.J. and Parker, A.C.: REAL: A program for REgister ALlocation, *Proc. 24th Design Automation Conference* (*DAC*), pp.210–215 (1987).
16) MIPS Technologies, Inc.: http://www.mips.com
17) Pozzi, L. and Ienne, P.: Exploiting Pipelining to Relax Register-file Port Constraints of Instruction-Set Extensions, *Proc. 2005 international conference on Compilers, architectures and synthesis for embedded systems* (*CASES*), pp.2–10 (2005).
18) Seto, K. and Fujita, M.: Custom Instruction Generation with High-Level Synthesis, *Proc. 6th IEEE Symposium on Application Specific Processors*, IEEE, pp.14–19 (2008).
19) Sun, F., Ravi, S., Raghunathan, A. and Jha, N.K.: A Scalable Application-Specific Processor Synthesis Methodology, *Proc. 2003 IEEE/ACM International Conference on Computer-Aided Design*, pp.283–290 (2003).
20) Tensilica Inc.: http://www.tensilica.com
21) The LLVM Compiler Infrastructure: http://llvm.org/
22) Toshiba Corporation: http://www.semicon.toshiba.co.jp/eng/index.html
23) Yu, P. and Mitra, T.: Characterizing Embedded Applications for Instruction-Set Extensible Processors, *Proc. 41st Design Automation Conference* (*DAC*), pp.723–728 (2004).
24) Zuluaga, M. and Topham, N.: Resource Sharing in Custom Instruction Set Exten-

sions, *Proc. 2008 Symposium on Application Specific Processors* (*SASP*), pp.7–13 (2008).

(Received May 16, 2009)
(Revised September 4, 2009)
(Accepted October 31, 2009)
(Released February 15, 2010)

(Recommended by Associate Editor: *Hiroyuki Tomiyama*)

Kenshu Seto received the B.S. in electrical engineering, the M.S. and D. Eng. in electronics engineering from the University of Tokyo in 1997, 1999 and 2004, respectively. From 2004 to 2006, he was a researcher at VLSI Design and Education Center (VDEC), University of Tokyo. He joined the Department of Electrical and Electronic engineering, Musashi Institute of Technology (renamed as Tokyo City University) in 2007. His research interests include high-level synthesis and compiler techniques for System-on-Chips (SoCs).

Masahiro Fujita received the B.S. degree in electrical engineering in 1980, and the M.S. and Ph.D. degrees in information engineering from the University of Tokyo, Tokyo, Japan in 1982 and 1985, respectively. From 1985 to 1993, he was a Research Scientist with Fujitsu Laboratories, Kawasaki, Japan. From 1994 to 1999, he was the Director of the Advanced Computer-Aided Design Research Group, Fujitsu Laboratories of America, Sunnyvale, CA. He is currently a Professor in the Department of Electrical Engineering, the University of Tokyo, Tokyo, Japan. He has been on program committees for many conferences dealing with digital design and is an Associate Editor of Formal Methods on Systems Design. His primary research interest is in the computer-aided design of digital systems. Dr. Fujita received the Sakai Award from the Information Processing Society of Japan in 1984.