IPSJ Transactions on System LSI Design Methodology Vol. 3 91–104 (Feb. 2010)

Regular Paper

High-level Synthesis Challenges for Mapping a Complete Program on a Dynamically Reconfigurable Processor*1

Takao Toi,†1 Noritsugu Nakamura,†1
Yoshinosuke Kato,†1 Toru Awashima†1
and Kazutoshi Wakabayashi†1

This paper presents a high-level synthesizer to map a complete program efficiently on a dynamically reconfigurable processor (DRP). Initially, we introduce our DRP architecture, which is suitable for control-intensive programs since it has a stand-alone finite state machine that switches "contexts" consisting of many processing elements (PEs). Then, we propose three new techniques optimized for our DRP. Firstly, we explain how synthesized control steps are mapped onto the contexts. Several control steps are combined as a context to utilize PEs efficiently since each control step does not require the same amount of operational units. Secondly, we describe a modulo scheduling algorithm for loop pipelining, considering both spatial and time dimensions of our DRP. Lastly, we explain a scheduling technique to optimize clock frequency, which can take advantage of multiplexer, wire and routing switch delays. We have demonstrated a JPEG-based image decoder example to evaluate our methods. Experimental results show that high area efficiency is achieved by balancing the number of PEs between contexts. Despite an overall increase in performance on pipelining of 3.6 times that without pipelining, the number of operational units increased by a factor of 2.2. The clock frequency is maximized with accurate delay estimation.

1. Introduction

A coarse-grained dynamically reconfigurable processor (DRP) has a new programmable architecture that enables switching of time-multiplexed data-paths. Each data-path is configured as a "context" consisting of many operational and storage units and the wire connections between them. This enables the DRP to execute highly complex and parallel data-paths.

†1 NEC Corporation
⋆1 This paper is based on Ref. 10).

The continuing growth in the demand for flexible, low power, and high-performance processors has led to the development of two new types of DRPs. The first type is a frequent context switching processor, such as Berkeley's Garp [9], IMEC's ADRES [1], and Keio's MuCCuRA [2], and our DRP [3]. These processors switch the context within one clock cycle, usually with nanosecond order. The second type, such as Chameleon [4], IPFlex's DAP/DNA [5], and Panasonic's (formerly Elixent's) D-Fabrix, PACT's XPP [6], switches the context less frequently. These processors take one or more cycles to switch the context.

The applications of these architectures spend most of their time executing a few time-critical code segments, typically loops. Most of hardware accelerator-type processors are designed to accelerate these parallelizable code segments. Because many of the applications are a mix of parallelizable code segments and control-intensive code segments, these accelerators are usually used with a CPU, which is more suitable for the control-intensive segment. A bottleneck of this accelerator is data transfer between both processors. The performance is decreased if there are frequent data transfers through a CPU bus. IMEC's ADRES [1] is tightly coupled with a Very Long Instruction Word (VLIW) processor and a reconfigurable matrix to solve this bottleneck. We took the other approach that both control-intensive and parallelizable segments are processed in the same reconfigurable architecture. There is no need to transfer data between different architectures with this approach. However, control-intensive segment is difficult to deal with for most reconfigurable architectures. We solve this problem by effectively combining the context switching mechanism and traditional multiplexing.

There are many control steps in the control-intensive segment. To share resources among control steps, two major methods are used:

- Multiplexing the input of a resource such as an operator
- Changing the operation whose instruction code is fetched from an instruction memory

The first method is mainly used with a wired logic for a single context device. It is efficient, especially when the number of multiplexer's inputs is limited in a few control steps. The second method is widely used in programmable architectures such as CPUs, DSPs, SIMD processors, and VLIW processors. A DRP, which switches the context frequently, can be classified as one of the programmable ar-

chitectures. The difference between a traditional CPUs and a DRP is data-path configurability. Not only operations but also wire connections are configurable, which enables highly complex data-paths. Ideally, all the resources are equally used in contexts. Although, performance does not depend on the number of control steps, it is difficult to balance the resources used in contexts, especially with the control-intensive part. We solve this problem by combining limited control steps to maximize resources at any one context without exceeding the allowable maximum for that context. In a context, a resource is shared by multiplexers. A few control steps at most are combined as a context.

Although the context switching mechanism works well with the control-intensive segment, the two dimensional context is basically used for accelerating the parallelizable segment. We exploit most parallelisms of wired logic, such as operation, memory, and loop levels. These parallelisms are automatically extracted by a high-level synthesizer (HLS) from a C source code. We developed a C-based HLS as a front-end tool for the compilation flow, which is based on our proprietary HLS for application-specific integrated circuits (ASICs)[13),14)]. The tool is used to extract operation-level and memory-level parallelism by generating a control data-flow graph (CDFG) that splits up the description of each control step based on given constraints. Because the compiler needs to fill the context not only in two spatial dimensions but also vertically (time-multiplexed), a modulo scheduling algorithm is incorporated to exploit loop-level parallelism.

Comparing multi-context-type DRPs to single context-type reconfigurable devices such as field-programmable gate arrays (FPGAs), although these two architectures seem to be different, the performance is not so different for two reasons. First, custom arithmetic logic operations are only slightly faster than look-up table (LUT)-based arithmetic operations with dedicated architectural features of FPGAs[7)]. Second, and more importantly, wire delay is a dominant factor in this type of reconfigurable device. FPGAs and DRPs have a wiring structure with many switches between segmented wires. If there is no limitation on the area and with using the same process technology, the performance of the same circuit design would not be the same but not quite different. Their performances mainly depend on parallelizing techniques used in a circuit design. These techniques are generally applicable to any wired logic devices including FPGAs and DRPs.

It is difficult to predict the delay at HLS level for fine grained architectures such as FPGAs. The delay on a critical path mainly depends on the level of a look-up table, but its level is subject to change in logic optimization. On the other hand, a coarse-grained architecture might control the delay more accurately. To reduce the configuration memory size, all the introduced architectures[1)–6),9)] are coarse-grained, although their granularities depend on the application on which they focus. The processing element (PE) level in an HLS can be constrained since few logic optimization techniques can be applied. There are still error factors in placement and routing that are not necessary to add.

In this paper, we mainly focus on three points. The first point is obtaining high area efficiency by context compaction. The second point is reducing cycle count by exploiting loop-level parallelism. The third point is improvement of delay estimation accuracy.

2. Dynamically Reconfigurable Processor

A DRP is a coarse-grained, multi-context, reconfigurable core that can be integrated with an ASIC. Some other reconfigurable architectures, including FPGAs, use data-paths to synthesize a sequencer[8)], making it difficult to control the sequencer from outside the chip. In some DRP architectures, an embedded CPU is used instead of a sequencer[5)], but a CPU is too slow to handle the state transition of data-paths. One architectural characteristic of our DRP[4)] is that the sequencer is not synthesized using data-paths; it is a stand-alone unit. Therefore, it is fast enough and is controllable thorough a bus. Many PEs and memory units are arranged on a DRP. Both the operations and the wires to use between the PEs and other resources, such as the on-chip memories and external ports, are selected based on configuration codes stored in each PE. The configuration is selected within one clock cycle (less than a nanosecond) by the sequencer, which is called a "state transition controller (STC)".

A primitive DRP unit is called a "tile", and a DRP core consists of an arbitrary number of tiles. In our second generation chip "DRP-2", there are four tiles on the chip. As shown in **Fig. 1**, each tile consists of 64 PEs, an STC, one- or two-port on-chip embedded memory units, and 16-bit multiplier units around the edge. A VMEM is an 8-bit, 512-word synchronized memory with two data inputs and

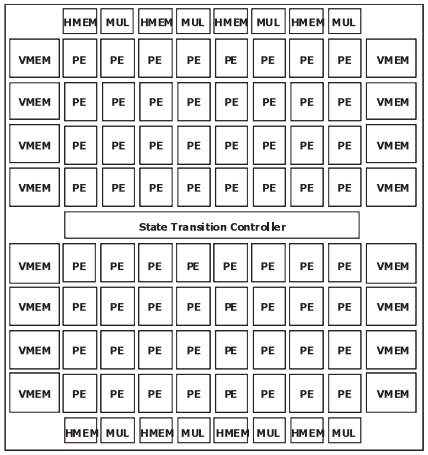

Fig. 1 Tile structure of DRP-2. Each tile consists of processing elements (PEs), a state transition controller (STC), 2-port memory units (VMEMs), 1-port memory units (HMEMs) and multiplier units (MULs).

outputs. A HMEM is an 8-bit, 4096-word synchronized memory unit with one data input and output. Both the bit width and the depth can be expanded using four units without occupying any PEs. The STC, located at the center of the tile, controls both the context and state transition based on an internal state transition table. Our DRP can execute multiple processes concurrently up to the number of tiles.

As mentioned in the introduction, to reduce the size of configuration memory, our DRP consists of 1/8/16-bit granularity architecture. As shown in **Fig. 2**, each PE has an 8-bit arithmetic logic unit (ALU), a data manipulation unit (DMU) for both 8-bit shift/mask operations and 1-bit logic operations, two 8-bit register file units (RFUs), and wire switches. By combining both the ALU and the DMU in the PE, an operand is extended for a 16-bit adder or 4-to-1 multiplexer (8-bit). Up to 32 different configuration codes are stored on-chip. Additional codes can be downloaded on-the-fly from external memory.

3. High-level Synthesis for DRP

As silicon process technology progresses, it has become a challenge to design

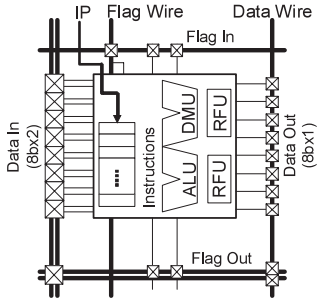

Fig. 2 PE architecture. Each PE has an arithmetic unit (ALU), a data manipulation unit (DMU), 2 register file units (RFUs), and wire switches.

complex logic circuits. High-level synthesis has attracted much attention for handling the complex design of reconfigurable chips [11]. Some of them use C compiler for a CPU as a front-end phase of compilation [12]. Compared to a register-transfer-level (RTL) design, there are several advantages to use a C-based HLS in the compilation flow for a DRP. Designers can write a code more efficiently by using a higher abstraction-level language. When a C compiler for a CPU is used during design verification flow, behavioral-level simulation is faster than RTL simulation. The C-source code can be debugged functionally using a sophisticated source-level debugger for a CPU. It is more important for reconfigurable chips to shorten the design turn-around time than ASICs, which takes longer time to fabricate.

However, existing compilers for dynamically reconfigurable processors do not adequately meet these challenges. Some use schematic entry, one uses its own proprietary language at the RTL, and a limited "C language" can be used only for data-path generation [8]. A designer who uses these tools still has to deal with traditional problems such as meeting timing and area constraints, as well as the additional problem of using the resources not only two dimensionally but also vertically (time-multiplexed). It should be noted that we had reported a compiler for our first generation chip DRP-1 [10]. In this paper, we supported the DRP-2 and added new features including pipelining and context switch reduction.

We have developed an integrated C compiler for our DRP. Like a C compiler for a microprocessor, our compiler includes the entire system environment. It

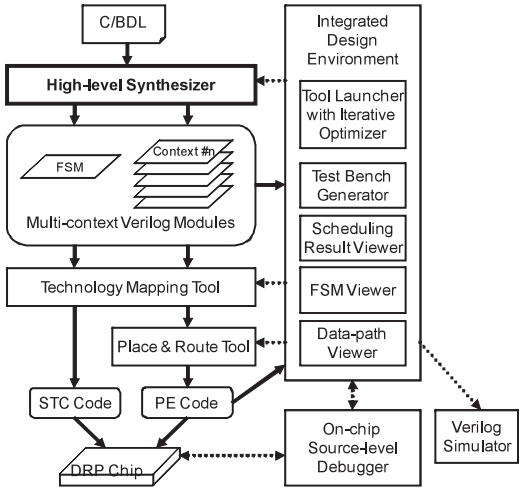

Fig. 3 Compilation flow (left) and design environment (right) for DRP.

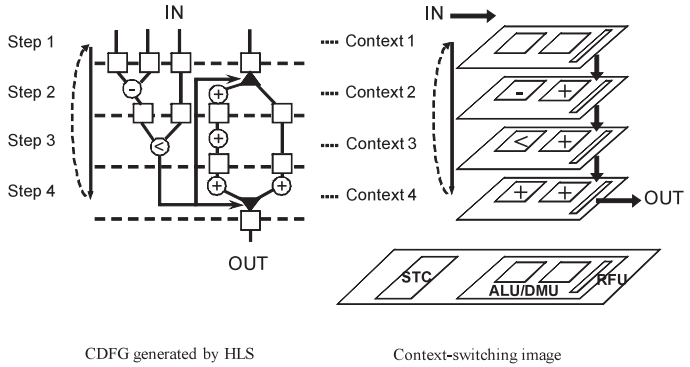

Fig. 4 Basic relationships between steps and contexts.

includes a HLS, a technology mapping tool, a place and route (P&R) tool, an on-chip source-level debugger, and an integrated development environment (IDE) with a graphical user interface, as shown in **Fig. 3**.

Unlike microprocessors, the clock frequency of our DRP varies since each resource has its own delay, and the PEs can be chained without inserting a register or memory. Therefore, a HLS with an automatic scheduler is useful for controlling delays on the data-paths. Moreover, a HLS usually extracts both data-paths and finite state machine (FSM) from the description. This corresponds to the DRP architecture in which the data-paths and STC are handled separately. Right from the start, we designed our DRP architecture with this C-based HLS in mind.

3.1 Allocating Multiple Contexts

Our DRP compiler inputs C, or behavioral design language (BDL), and outputs the downloadable configuration code (STC Code and PE Code shown in Fig. 3). The BDL is a subset/superset of standard C language. For example, hardware-specific notation, input and output port declarations, bit-level extraction, and concatenation are all BDL extensions. There are some restrictions which are difficult to realize on hardware such as recursive call and dynamic memory allocation. Some types of pointer are supported if they are statically analyzable.

Our DRP HLS is a front-end tool that generates a "multi-context Verilog", in which the contexts are divided into separate modules. Although the generated Verilog code cannot be synthesized using generic logic synthesizers, it can be simulated using an RTL simulator. **Figure 4** shows the communication path between the data-flow graph and the DRP resources. A FSM generated with our HLS is mapped onto the STC. Basically, the control steps have a one-to-one relationship with the contexts. We can treat the context switching mechanism as a state transition, because the STC is fast enough. However, a function that combines multiple control steps into a single context obtains better area efficiency, as described below.

An operator is mapped onto either the ALU or DMU as an instruction for that unit. A register is mapped onto the RFU, and an array is mapped to the embedded memory unit (VMEM, HMEM), which is automatically selected based on its depth.

3.2 High-level Synthesis Flow

A block diagram of our HLS is shown on the left in **Fig. 5**. To shorten the synthesis time, the flow is straight forward. The C/BDL description is translated into a tree-structured control-flow graph (tCFG). Some optimizations, such as constant propagation, common sub-expression elimination, loop unrolling, in-

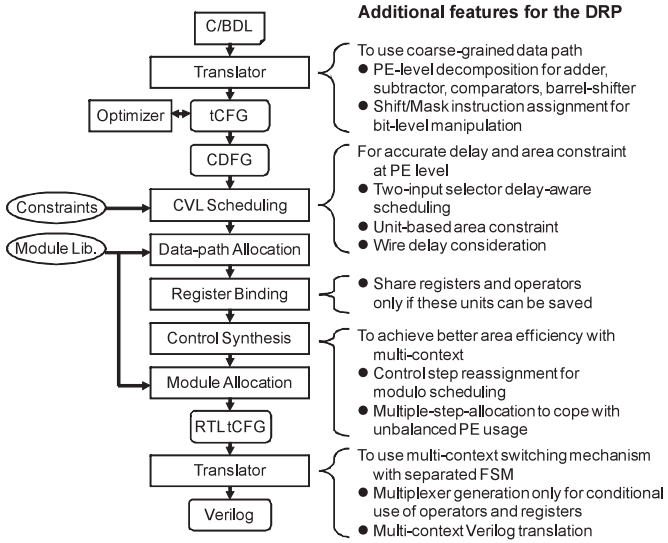

Fig. 5 Flow of HLS and additional features for DRP.

line procedure expansion, and dead code elimination, are applied to the tCFG. A CDFG is generated from the tCFG. Scheduling, data-path allocation, module binding, and control synthesis are processed using the CDFG. A condition vector list scheduling (CVLS) algorithm is used to efficiently allocate resources under the given constraints [15),16)]. Other optimizations, such as redundancy elimination, are applied to the RTL tCFG. Finally, the internal format is translated into an RTL language.

4. Optimization Techniques for DRP

Some otherwise unimportant issues for ASICs could become critical problems for our DRP. Still, some problems must be anticipated. **Table 1** summarizes the differences between a HLS for ASICs and that for the DRP. Most of the differences are caused by our DRP's coarse-grained architecture. Sharing rules have changed because the delay of selector instruction (2-input multiplexer) is not negligible. In operator level, ALU/DMU unit-level decomposition is necessary. Area constraint must be based on number of units. Other differences are caused by the multi-context architecture. The following circuit-size variance is one example.

Table 1 Differences between HLS for ASICs and for our DRP.

	For ASICs	For the DRP
Data path	Single context	Multiple contexts (Must average the number of operational unit)
FSM	Wired logic (with variable delay)	STC code (with fixed delay)
Operator / Register sharing	By using multiplexer (Its delay is negligible. Must do scheduling and sharing simultaneously)	Basically no sharing (Share only if unit is saved)
Conditional expression		By using selector instruction in either ALU or DMU (Must consider delay)
Operand and wires (Granularity)	Any bits (Fine-grained)	8-bit: 2 / 3 input and 1 output, 1-bit: 1 / 3 input and 1 output (Coarse-grained)
Bit concatenation/ extraction	Wire connection	Shift/Mask instruction (Must consider delay)
Area constraint	Based on number of operators or memory elements	Based on number of units (ALU / DMU / VMEM / HMEM)

4.1 Context Compaction Technique

Circuit-size variance between control steps is one potential problem. While this is not an issue for ASIC designs, the variance in the number of PEs used among control steps becomes a problem in our DRP, especially if we assume a one-to-one relationship between control steps and contexts. This assumption increases the number of unused PEs in the context. We implemented a synthesis technique called multiple-step-allocation to achieve higher area efficiency by utilizing the benefit of having multiple contexts.

If a specified description is scheduled using only a delay constraint, the number of PEs used in the control step might vary. If the upper limit of the number of PEs is constrained in the control step, an operation that exceeds the limit slides to the next control step. However, the control step might be changeable without reaching the limit. For instance, since only one access in a clock cycle for each data-port is permitted using synchronous memory, and if any output data of an array has a data dependency on the same memory's input address, the control step is changed because of that data dependency, even though it does not push

the limits of any of the constraints. In this case, the context might only have a few memory accesses, and most of the PEs are not used. There are several causes of state transition (in order of descending priority):

- Control statements including loops, such as "while" or "goto"
- Access to synchronous resources, such as memory or external ports
- Delays and area constraints

The multiple-step-allocation technique helps reduce the number of contexts, in which several control steps are combined based on the number of PEs in the control step and other constraints. Similar to single-context devices, this technique uses a multiple-input multiplexer for sharing resources, such as register units, memory units, and the external ports between the control steps. For the memory units and external ports, the number of multiplexer's inputs is equal to the number of data inputs. For the number of register units, it is basically equal to the number of data inputs plus one with self-feedback. Because the RFU has its own write enable (WE) controlled as a part of the instruction set, the input with self-feedback can be eliminated if the register is not alive in the control steps. The scheduler provides lifetime information for these resources.

A step activation signal for the multiplexer is sent from the STC, in which a table of the relationships between control steps and contexts is stored. This technique enables our DRP to store more control steps than the number of contexts, which is limited by the size of the configuration memory in a PE. Special care must be taken not to increase the delay of the circuit when inserting the multiplexer. It should not be inserted into a critical path. Other control steps that are not on the longest path can be combined unless doing so creates a path with a delay that exceeds that of the critical path. Of course, the number of PEs has to be within the available PEs in the context. Therefore, the number of multiplexers is estimated each time when combining control steps into various possible combinations. To obtain the initial data-path information, this technique is applied to the translator for the RTL tCFG after the shared resources are allocated (Fig. 5).

There are two basic methods for combining the control steps:

- Combine consecutive control steps and change the context when the constraint limits are hit.

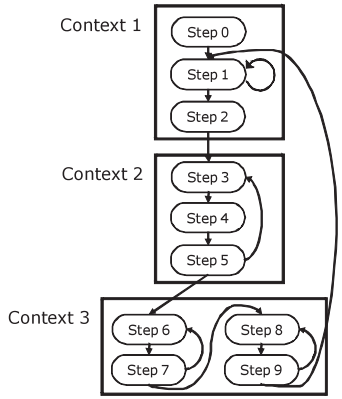

Fig. 6 Combining consecutive steps from inner loops.

- Combine as many control steps as possible until limits are hit, regardless of whether they are contiguous.

The first method can be used to reduce the amount of context switching, which consumes one third to half of the power for this kind of DRP. This method usually uses more multiplexers than the second method because there is a trend to use the same resources near the control step. The amount of context switching can be further reduced by combining the consecutive control steps from the inner loops (**Fig. 6**). Two or more loops can be combined as a single context if the number of control steps in the loops is limited. The second method provides better area efficiency and reduces the number of PEs more than the first by optimizing the control step selection process. The best case is that no multiplexer is inserted.

4.2 Pipelining

Pipelining is an effective technique for parallelizing data-paths. Most HLSs or compilers for this type of processor have some restrictions for describing in a loop to be pipelined [17),18)]. Our HLS removes these restrictions such as function call, branches, and multiple loop-breaks. A modulo scheduling algorithm [19)] is incorporated in our HLS for using the context switching mechanism effectively.

The same number of contexts as that of the data initiation interval (DII) is used. The control steps are folded by the number of DII and are combined as contexts. For example, there are four control steps in a loop after scheduling.

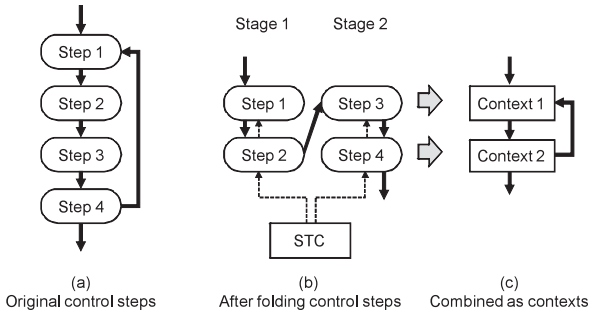

Fig. 7 Example illustration of pipelining (DII = 2).

The state transition is shown as (a) in **Fig. 7**. When DII = 2, the control steps are folded in the middle of the control steps (b). In this case, step 1 is the first stage and step 3 is the second stage in context 1 (c). The multiplexers for both conditional register writes and port outputs are controlled by stage activation signals provided by the STC.

The number of DIIs is increased until there are no pipeline hazards. Data hazards are avoided by a register forwarding technique in the scheduler to the maximum extent. To avoid many small state machines from being generated in the PE array, a generated pipeline does not stall at the stage level. All pipeline stages are stalled one at a time by the STC. When a pipeline stall can not be avoided, e.g. handshaking for I/Os in the loop, the STC halts the state transition. The STC has a special input to stop its behavior from the PE array. Thanks to this stall support, there is no need to insert multiplexers to all the shared resources. Only port output and conditional register write require multiplexers and furthermore, some multiplexers for the first stage can be eliminated.

We also use the STC to control stages for the prologue and epilogue. There are two options for controlling pipeline stages for prologue and epilogue:

- Use dedicated contexts, which only contain active stages.
- Use a multiplexer in the PE for shared resource access.

Although the first option can reduce the number of PEs in a context, our HLS incorporates the latter stage control for two reasons. The first reason is that limited contexts are stored in on-chip configuration memory. The second reason

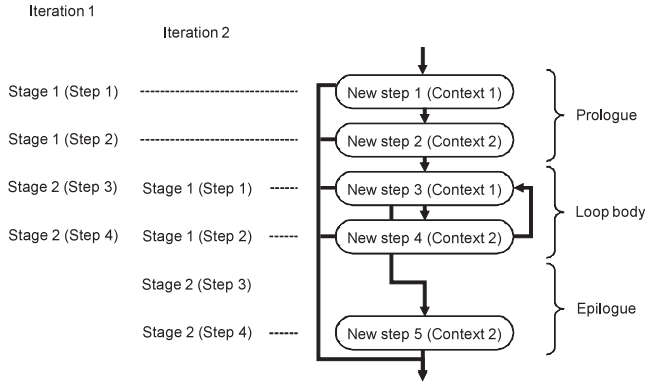

Fig. 8 Control step reassignment for prologue and epilogue.

is that most of the multiplexers can be eliminated because there is hardware support for a pipeline stall.

Although we do not allow dedicated contexts for the prologue and epilogue, we use the STC in a different way similar to the multiple-step-allocation in the previous section. **Figure 8** shows control step reassignment for the prologue and epilogue. The first stage is activated in new-steps 1 and 2, which belong to contexts 1 and 2, respectively. All stages are activated in new-steps 3 and 4 in a pipelined loop body. In this case, new-step 4 will never transit to the epilogue because the first iteration is already finished after step 2 in the second iteration is executed. Therefore, no new-step is assigned for the epilogue in context 1. There is no need to insert multiplexers in context 1, because the first stage is always active.

If there are two or more breaks in a loop, the break in an earlier iteration has a priority. For example, if there are breaks in the steps 1 and 3, the latter step 3 in the first iteration has the priority. All possible breaks in the new-steps are shown in Fig. 8.

4.3 Delay Consideration

4.3.1 Wire Delays and PE Granularity

In general, the wire delays of reconfigurable devices are much longer than the operational delays because they include both metal-wire and buffer and tri-state

route-switching delays. Wire delays can account for up to 60% of the overall design delays in FPGAs [20] and are a dominant factor in the performance of a DRP. Even in a HLS for ASICs, wire delays cannot be ignored, especially for deeper submicron processes. However, accurate wire delays can only be obtained after the P&R tool has finished the static timing analysis. Layout-driven scheduling-binding synthesis reportedly can avoid time consuming iterations of the design process [21]. IPFlex [5] avoids this issue architecturally by guaranteeing the worst-case delay between two operators in any location and prohibiting PE chaining. By utilizing coarse-grained data-paths, our DRP HLS can control the delay of the circuit at the PE level.

We focus on wire delays since they are dominant. A typical wire delay between PEs was added to each operational delay. It was calculated based on previous experimental measurement using the P&R tool. Other pre-measured delays were added to the setup and delay of the embedded memory units and the delay of external ports because their placements on the chip are more restricted than those of the PEs.

PE-level decomposition during the pre-scheduling translation stage takes into account the coarse-grained architecture of our DRP. Although our DRP basically has an 8/16-bit granularity architecture, both the ALU and DMU have 1-bit inputs and output. They are used for carry-in and carry-out flags, comparison result flags, logical operations, etc. A 32-bit adder, for instance, is decomposed into two cascaded 16-bit adders, each of which has a wire delay. Without this decomposition, the original 32-bit adder, which has longer wire delays for two wire legs cannot be scheduled unless it fits within the timing constraint. This decomposition is applied to C-operators, which are decomposed into more than one PE level during the pre-scheduling translation stage. These operators include an adder, a subtractor, comparators, and a barrel-shifter. With this decomposition, the scheduler more precisely controls the delay at each PE level, reducing the number of control steps required.

Care must be taken when there is bit-level extraction or concatenation. Since these operations become the wiring connection in an ASIC, they do not have to be considered in the scheduler. The DRP requires a shift/mask instruction in the DMU, and the delay and area for this instruction must be carefully considered. In our DRP HLS, bit-level extraction or concatenation that does not align to one or eight bits is transformed into a shift/mask instruction during the pre-scheduling translation stage; this instruction is then treated the same as any other instruction by the scheduler.

4.3.2 Multiplexer Delays in Context Switching

A HLS for single-context devices, such as ASICs or FPGAs, treats the data-path and FSM separately. A generated RTL FSM is merged into the data-path using a logic synthesizer. On the other hand, multiple contexts must be considered in our DRP HLS, in which the STC is a stand-alone module that handles context transition and is outside the PE array. This separation enables the DRP scheduler to treat multiplexing with a completely new approach.

There are two types of multiplexers: those for resource sharing among control steps and those for conditional branching. The two types are usually merged for a single-context device. However, this can prevent accurate timing estimation in the scheduler because the number of inputs for a multiplexer is difficult to predict only from the CDFG, especially if resources are shared among several control steps. Operator sharing is an effective technique for optimizing the data-path. Therefore, except for the primitive logical operators, which are smaller than a multiplexer, operators are shared using the CVLS algorithm in our ASIC HLS. Although the delay of a multiplexer with only few inputs is negligible in an ASIC, the scheduling of such multiplexers is difficult because the resources are actually shared among the control steps during the "Data-path Allocation" or "Register Binding" stages (see Fig. 5). An example illustration of resource sharing is shown in **Fig. 9**. If the adders in steps 2 and 3 are shared, assuming registers A and C are the same, a two-input multiplexer is needed only if both registers B and D are not shared. If all adders including the two in step 4 are shared, a multiplexer with more input may be needed. However, the number of inputs is difficult to predict during the scheduling stage because registers must be optimally assigned simultaneously.

For our DRP, a multiplexer for sharing resources among control steps is hidden in the fast context switching mechanism with a fixed delay. Because the RFU has its own write-enable flags as a configuration bit, a multiplexer for preserving the value of a flip-flop for more than one step is not necessary. The context is switched

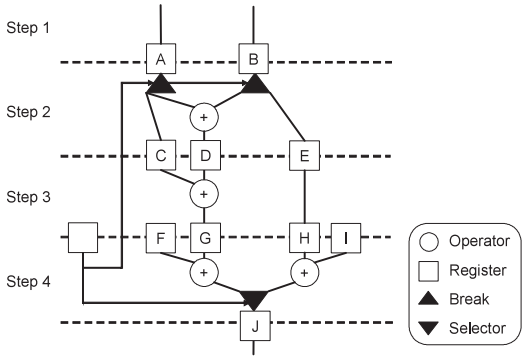

Fig. 9 Example illustration of resource sharing.

by changing both the wire connection and the operational code without using a selector instruction (SEL) in either the ALU or DMU. The SEL is conventionally used for conditionally selecting signals in a control step.

In our DRP HLS, a SEL is treated the same as the other instructions. For example, although the operational delay of a SEL and an adder are not the same, the delay for each PE level differs slightly since wire delay is the dominant factor in a reconfigurable device. From the perspective of PE usage, both a SEL and adder are the same ALU instruction, although their actual areas on the silicon differ. For these reasons, the operators are shared in a context only if both the number and the level of the PE do not increase. In Fig. 9, because the two adders in step 4 are exclusively used, they should be shared only if registers F and I are the same. Although this sharing does not reduce the PE level because it simply moves the multiplexer to be inserted from the input of the adder to the output, the number of operational units is reduced from three (two adders and one selector to select the adders) to two (one adder and one selector to select register G or H).

Eventually, only a SEL for conditional use of an operational, register, or memory unit or external port in a control step must be scheduled. This helps the scheduler more precisely estimate the conditional delay. A SEL placed before either an operational or register unit is depicted as a node in the CDFG. Our DRP HLS treats this selector node as an operator node which has a delay. In Fig. 9, if the total delay of both a selector node and the adders in step 4 is more than the designed delay, the selector node is moved to step 5, like the other operator nodes. A selector node with only one input can be ignored for a register or operator but is necessary for an external or memory port. These ports are depicted as more than one node in a control step on the CDFG when they are conditionally used. For these types of resources, we add the SEL delay into the node. We must carefully treat a SEL for the data input port of a memory unit, because the unit has a write or read-enabling signal, which also works as a selector.

This new multiplexer handling also simplifies the work of the P&R tool. Because most operational instructions are not shared in a context or among contexts, there is no restriction on the multiplexer location. Essentially, this means the operational instructions including the SELs are bound at placement. Therefore, a faster data-path with fewer wire connections can be synthesized.

5. Evaluation

We evaluated the performance of our DRP HLS by using a JPEG-based image decoder. All functions of the decoder are compiled and stored in the configuration memory of the DRP-2. We manually divided the decoder program into three processes, an inverse discrete cosine transform (IDCT) with an inverse quantize (IQ) and two variable length decoders (VLDs). The VLDs are control-intensive codes and the IDCT/IQ is a parallelizable code. Many loops in the IQ/IDCT are pipelined. The IQ/IDCT process uses two tiles, and each of the VLDs uses one. These processes use their own STCs (as FSMs) and work simultaneously. Any two processes are connected by VMEMs, which works as FIFOs.

It should be noted that we had compared the performance to CPUs and DSPs on several stream applications [22] and several encryption algorithms [23].

5.1 Effect of Context Compaction

The numbers of operational units allocated in each context of the VLD's secondary process using the following three allocation rules were counted and sorted numbers are shown in **Fig. 10**. This process achieves the best context compaction effect in the three processes. The summary of the effect for all process are shown in the next section. Because two operational units, the ALU and DMU, com-

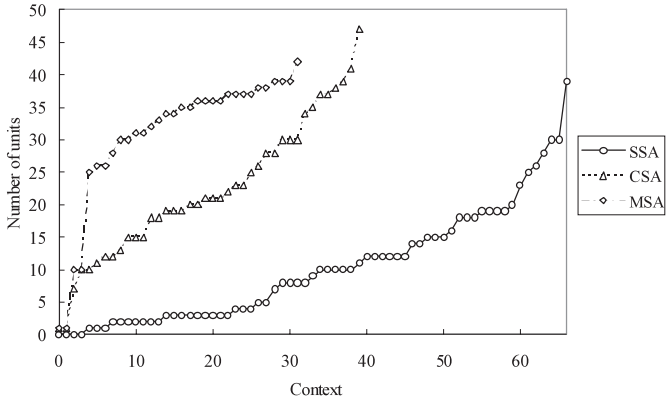

Fig. 10 Number of operational units assigned in each context of VLD's secondary process.

Table 2 Comparison between step-allocation rules of VLD's secondary process.

	SSA	CSA	MSA
Total contexts	67	38	32
Rate of context switch	0.45	0.36	0.45
Maximum units	39	47	42
Total units	683	891	979

prise a PE, we took the larger of these counts as a result. The total number of contexts, rate of context switch, maximum number of units, and total number of units are listed in **Table 2**. The rate of context switch is calculated as 1 if the context switches every cycle.

- Single-step-allocation (SSA)

  Allocating the single-step to single-context rule results in several contexts having many operational units. Some of the other contexts use only a few operators.
- Consecutive-step-allocation (CSA)

  Allocating the consecutive-steps to single-context rule results in a reduction in the total number of contexts although it still exceeds 32, which is the number of contexts that can be stored in the DRP-2. The rate of context switch is reduced by 20%. The maximum number of operational units slightly increases because multiplexers are inserted to share the resources.
- Multiple-step-allocation (MSA)

  Allocating the multiple-steps to single-context rule results in a reduction in the total number of contexts to approximately half compared to that of the single-step-allocation rule. This rule combines control steps regardless of whether they are contiguous. The number of operational units is well balanced between contexts. More context compaction is possible but will increase the total number of operational units for the multiplexer insertion. Because the increase of the unit leads to wire congestion, the total number of context is set to 32.

5.2 Volume Efficiency

Two criteria, context and area, are used to evaluate "area efficiency" for multi-context devices. Therefore, it should be renamed "volume efficiency" for this type of architecture.

To evaluate the previous context, we defined the context usage rate $C_{rate} = C/32$, where C is the number of contexts used and 32 is the number of contexts that can be stored in the configuration memory of the DRP-2. If C_{rate} exceeds 1, all the contexts cannot be stored simultaneously. With the MSA rule, all the processes are successfully stored in the configuration memory (**Fig. 11**). Because there is no need to spare the context, the number of contexts is constrained to no less than 32.

The area is primarily resolved using the number of PEs for a process, 128 PEs in two tiles for the IDCT/IQ for example. Let the number of operational units allocated in context i be OPE_i, where 64 is the number of PEs in a context, the filling rates of operational unit OPE_{rate} are defined as follows and shown in **Fig. 12**.

$$OPE_{rate} = \frac{\sum_{i=0}^{C} OPE_i}{64C \cdot \left\lceil \frac{\max\{OPE_i\}}{64} \right\rceil}.$$

A higher filling rate is better for efficient area usage. The MSA offers the best

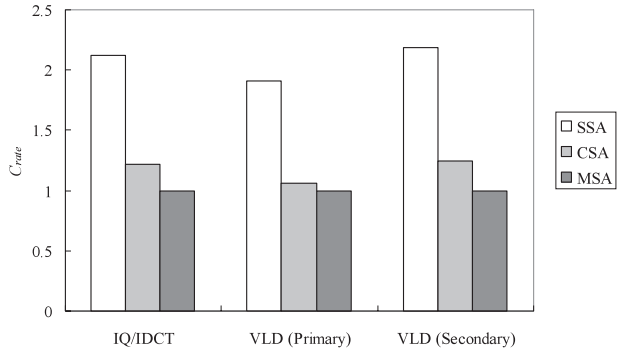

Fig. 11 Context usage rates.

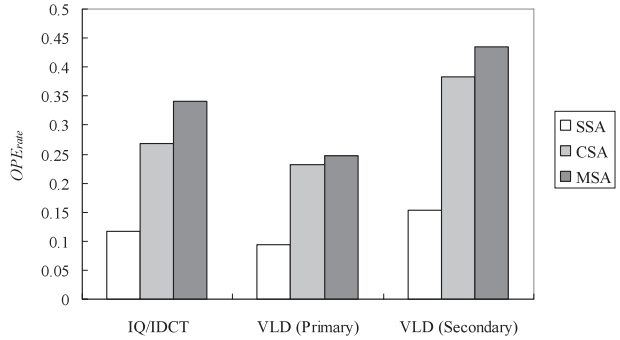

Fig. 12 Operational unit filling rates.

from the standpoint of "volume efficiency".

5.3 Estimated Delay and Cycle Count

Figure 13 shows the relationship between the delay estimated with our HLS and the delay obtained by static timing analysis in the P&R tool of all three processes with the MSA rule. The close delay estimation between the results shows that both of our PE-level wire-delay and multiplexer-delay controls works well. Because of wire congestions, the delay of the P&R slightly increased from that of the HLS in IQ/IDCT and the secondary VLD process. Both processes have higher unit filling rates than the primary VLD process. The slowest delay

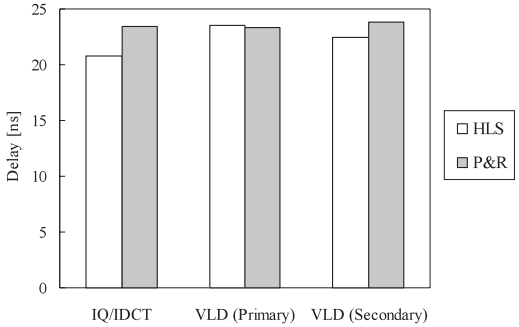

Fig. 13 Delay estimation.

Table 3 Pipelining effect on IQ/IDCT process.

	SSA (not pipelined)	SSA (pipelined)	MSA (pipelined)
Cycle count	2425	672	672
Maximum units	44	97	97
Total units	948	1015	1394

is the actual delay since there is only one clock domain in the DRP-2.

The circuit throughput depends on both the delay and the number of cycles it takes to execute test data. The cycle is determined by the number of scheduled control steps and by the sequence of FSM generated with the HLS. The IQ/IDCT process has 13 pipelined loops, 6 loops are DII = 1 and 7 loops are DII = 2. The number of cycles for a macro-block and the number of operational units are listed in **Table 3**. The macro-block consists of 6 blocks, each with 64 (= 8 x 8) elements. Despite the number of cycles is reduced to less than 1/3 with pipelining, the maximum number of units is increased by the factor of 2.2. The total number of units increases only 7% for the multiplexer, which is required for pipelining. The fewer increase is better from the point of wire congestion. Note that total number of units with MSA increases because the IDCT/IQ process has higher data bandwidth than the other two processes.

Accurate delay estimation of our HLS enables the optimal throughput to be found more quickly without running the P&R tool. This is especially valuable when trying to identify the best optimization options and constraint combination in a reasonable amount of time. Since there are many combinations of the numbers to be constrained, there is an iterative optimization function in the integrated development environment (see Fig. 3) to support the designer. It automatically tries all combinations of constraints, which are the number of operational units and the delays, in the given ranges. Our HLS also automatically attempts many compiler options such as the multiple-step-allocation, pipelining, a loop unrolling and a function inlining with the support of the iterative optimization function. Along with design productivity, this is one of the key advantages of a HLS compared to designing in RTL language. Moreover, it fits well with the nature of our DRP, which offers a trade-off between the best throughput (by increasing circuit parallelization) and the minimum area (by dynamically switching the contexts).

6. Conclusions and Future Work

We have developed a C-based compiler for our DRP, using C language in all phases of development; algorithmic-level optimization, circuit optimization considering hardware architecture, functional-level simulation, and on-chip debugging. Using a short turn-around-time tool set, the designer can take full advantage of the reconfigurable processor, for example, by starting a design before the functional specifications have been fixed or by designing using with one or a few developers. By making use of multi-context and coarse-grained data-paths of our DRP architecture, our HLS can control delay at the PE level and properly constrain the number of elements properly. A more accurate delay estimation, which takes care of the routability, will increase performance. Although the techniques introduced in this paper are primarily for our DRP, most can be applied to other coarse-grained reconfigurable processors.

Although, the compiler which converts a high-level description into logic-level coding can deal with a wide range of parallelization, a source code optimization process is needed. To overcome the disadvantages of adopting a high-level language, more parallelizing techniques are needed to expand the synthesis search space. To exploit concurrent loop-level parallelism, an automatic multiple processing is needed since we manually divide the application into several processes.

References

1) Mei, B., et al.: Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study, *Proc. DATE'04*, Vol.2, pp.21224–21229 (Feb. 2004).
2) Amano, H., et al.: MuCCRA chips: Configurable dynamically reconfigurable processors, *A-SSCC 2007*, pp.384–387 (Nov. 2007).
3) Motomura, M.: A Dynamically Reconfigurable Processor Architecture, *Microprocessor Forum* (Oct. 2002).
4) Bondalapati, K.: Parallelizing DSP nested loops on reconfigurable architectures using data context switching, *Proc. 38th DAC*, pp.273–276, (2001).
5) Sato, T., Watanabe, H. and Shibata, K.: Implementation of dynamically reconfigurable processor DAPDNA-2, *VLSI Design, Automation and Test*, pp.323–324 (Apr. 2005).
6) Vorbach, M. and Becker, R.: Reconfigurable processor architectures for mobile phones, *Proc. Parallel and Distributed Processing Symposium*, pp.22–26 (2003).
7) Trimberger, S.M.: *Field-Programmable Gate Array Technology*, Kluwer Academic Publisher, p.10 (1994).
8) Cardoso, J.P. and Weinhardt, M.: Fast and Guaranteed C Compilation onto the PACT-XPP Reconfigurable Computing Platform, *Proc. 10th FCCM* (2002).
9) Hauser, J. and Wawrzynek, J.: Garp: A MIPS Processor with a Reconfigurable Coprocessor, *Proc. 5th FCCM*, pp.12–21 (1997).
10) Toi, T., et al.: High-Level Synthesis Challenges and Solutions for a Dynamically Reconfigurable Processor, *Proc. ICCAD'06*, pp.702–708 (Nov. 2006).
11) Callahan, T.J., Hauser, J.R. and Wawrzynek, J.: The Garp Architecture and C Compiler, *IEEE Computer*, Vol.33, No.4, pp.62–69 (Apr. 2000).
12) Sullivan, C., et al.: Deterministic hardware synthesis for compiling high-level descriptions to heterogeneous reconfigurable architectures, *Proc. 38th HICSS* (2005).
13) Wakabayashi, K.: C-based synthesis experiences with a behavior synthesizer, "Cyber", *Proc. DATE'99*, pp.390–393 (Mar. 1999).
14) Wakabayashi, K. and Okamoto, T.: C-based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective, *IEEE Trans. CAD*, Vol.19, Issue 12, pp.1507–1522 (Dec. 2000).
15) Wakabayashi, K. and Yoshimura, T.: A resource sharing and control synthesis method for conditional branches, *ICCAD-89*, pp.62–65 (Nov. 1989).
16) Wakabayashi, K. and Tanaka, H.: Global scheduling independent of control dependencies based on condition vectors, *Proc. 29th DAC*, pp.112–115 (Jun.1992).
17) Callahan, T., et al.: Adapting Software Pipelining for Reconfigurable Computing,

Proc. CASES'00, pp.57–64 (Nov. 2000).
18) Mei, B., et al.: DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures, *Proc. FPT'02*, pp.166–173 (Dec. 2002).
19) Lam, M.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proc. ACM SIGPLAN Conference on Programming language design and implementation*, pp.318–327 (Jun. 1988).
20) Xu, M. and Kurdahi, F.J.: Area and timing estimation for lookup table based FPGAs, *Proc. ED&TC*'96, pp.151–157 (Mar. 1996).
21) Xu, M. and Kurdahi, F.J.: Layout-driven high level synthesis for FPGA based architectures, *Proc. DATE'98*, pp.446–450 (1998).
22) Suzuki, N., et al.: Implementing and Evaluating Stream Applications on the Dynamically Reconfigurable Processor, *12th FCCM*, pp.328–329 (Apr. 2004).
23) Suzuki, M., et al.: Stream Application on the Dynamically Reconfigurable Processor, *Proc. 2004 FPT*, pp.137–144 (Dec. 2004).

(Received June 1, 2009)
(Revised September 3, 2009)
(Accepted October 31, 2009)
(Released February 15, 2010)

(Recommended by Associate Editor: *Akihisa Yamada*)

Takao Toi received his B.E. and M.E. degrees in mechanical engineering from Keio University, Japan, in 1993 and 1995. He was a visiting researcher in electrical engineering, Princeton University during 2005 and 2006. He is a Ph.D. candidate in information and computer science, Keio University, Japan. Since joining Central Research Laboratories, NEC Corporation, in 1995, he has been engaged in research and development in image processing for digital cameras and C-based high-level synthesis for dynamically reconfigurable processors.

Noritsugu Nakamura received his B.E. and M.E. degrees in electronic engineering from Kyoto University, Kyoto, Japan, in 1995 and 1997. He joined NEC Corporation, Kanagawa, Japan, in 1997. He is currently researching and developing high-level synthesis for dynamically reconfigurable processors. He is a member of the Information Processing Society of Japan.

Yoshinosuke Kato received his B.S. and M.S. degrees in Information science from Tokyo Institute of Technology in 2000 and 2002. He is currently with NEC Corp. His interests include high level synthesis and compiler for reconfigurable processers.

Toru Awashima received his B.E. degree in electrical communication engineering in 1988, and the M.E. and Dr. Eng. degrees in electrical communication engineering from Waseda University, Tokyo, Japan, in 1990, 1993. In 1993, he joined Central Research Laboratories, NEC Corp. Since then he has been working on physical design automation for LSI and printed circuit boards. His current interests include C-based design automation, place and route algorithms, and tests for dynamically reconfigurable processors.

Kazutoshi Wakabayashi received his B.E. and M.E. degrees and Dr. of Engineering from the University of Tokyo in 1984 and 1986. He was a visiting researcher at Stanford University during 1993 and 1994. He joined NEC Corporation in Kawasaki, Japan in 1986, and he is currently a Senior Manager of EDA R&D Center, Central Research Labs. Dr. Wakabayashi has been engaged in the research and development of VLSI, CAD systems; high-level and logic synthesis, formal and semi-formal verification, system-level simulation, HDL, emulation, HLS and floorplan links, and reconfigurable computing. He served on executive or organizing committees of international conference including: ASP-DAC'09 General Chair, CODES+ISSS'09 Co-Technical Program Chair, Secretary of Steering Committee of ASPDAC, Asian Rep. of ICCAD, Asian Rep. of DAC, Tutorial Chair of ASP-DAC'06, and Steering Committee of ITC-CSCC (09-). He has served on the program committees for several international conferences including: DAC, ICCAD, DATE, ASP-DAC, ISSS, SASIMI, and ITC-CSCC, ISCAS, VLSI-TSI, and SBCCI. He is also a member of IEEE, IPSJ, and IEICE. He received the Yamazaki-Teiichi Prize in 2004, the IPSJ Convention Award in 1988, Sakai Kinen Special Award in 2001.