Ph.D. DISSERTATION

Efficient Predication Techniques on Coarse-Grained Reconfigurable Architectures

재구성형 구조에서의 효율적인 조건실행 기법

BY

Kyuseung Han

AUGUST 2013

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Efficient Predication Techniques on Coarse-Grained Reconfigurable Architectures

재구성형 구조에서의 효율적인 조건실행 기법

BY

Kyuseung Han

AUGUST 2013

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Efficient Predication Techniques on Coarse-Grained Reconfigurable Architectures

재구성형 구조에서의 효율적인 조건실행 기법

지도교수 최 기 영

이 논문을 공학박사 학위논문으로 제출함

2013 년 5 월

서울대학교 대학원

전기 컴퓨터 공학부

한 규 승

한규승의 공학박사 학위논문을 인준함

2013 년 6 월

Abstract

Coarse-Grained Reconfigurable Architecture (CGRA) is one of viable solutions in embedded systems to accelerate data-intensive applications. It typically consists of an array of processing elements (PEs) and a centralized controller, which can provide high performance, flexibility, and low power. Parallel array processing reduces execution time of applications, reconfigurability of PEs allows changing its functionality, and simplified control structure with static scheduling for instruction fetching and data communication minimizes power consumption.

However, as applications become complex so that data-intensive parts are having control flows in them, CGRAs face a challenge for its effectiveness. Since the entire PEs are controlled by a centralized unit, it is impossible to execute programs having control divergence among PEs. To overcome the problem, we can adopt the technique called predicated execution, which is the unique solution known so far, but conventional predication techniques have a negative impact on both performance and power consumption due to longer instruction words and unnecessary instruction-fetching/decoding/nullifying steps.

Thus, this thesis reveals performance and power issues in predicated execution when a CGRA executes both data- and control-intensive applications, which have not been well-addressed yet. Then it proposes high-performance and low-power predication mechanisms. Experiments conducted through gate-level simulation show that the proposed mechanism improves energy-delay product by 11.9%, 14.7%, and 23.8% compared to three conventional techniques. In addition, this thesis also reveals mapping issues when mapping applications on CGRAs using the proposed predication. A power-saving mode introduced into PEs prohibits multiple conditionals from being parallelized if conventional mapping algorithms are used. Thus, this thesis proposes the framework to release this problem by mapping conditionals to different PEs. Experiments show that mapping results from the proposed approach lead to 2.21 times higher performance than those of the naïve approach.

Keywords: CGRA, reconfigurable, predication, predicated execution, low power, high performance

Student Number: 2008-21002

Contents

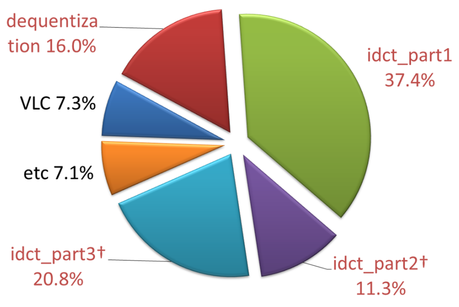
List of Figures

List of Tables

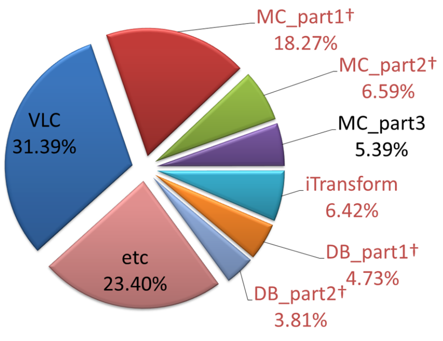
Chapter 1

Introduction

A Coarse-Grained Reconfigurable Architecture (CGRA) is an array of Processing Elements (PEs) which can be reconfigured to perform word-level operations. It is a viable solution for embedded systems since it can meet performance, flexibility, and low power consumption at the same time. Parallel execution on abundant PEs reduces the execution time of applications, and reconfigurability of PEs enables to change its functionality according to applications. To simplify the control of PEs, the schedule of configuration (or instruction) fetching and data communication are statically decided at compile time and the single controller orchestrates all. This feature makes CGRAs have the scalability on the number of PEs despite of the use of a single controller and also gives the efficiency on power consumption. Due to these benefits, there have been many researches proposing various kinds of CGRAs [1–10] and surveys on them [11, 12].



(a) JPEG decoder



(b) H.264 decoder

Figure 1.1: Application profiling results.

However, CGRAs face a challenge for its effectiveness as applications become complex so that data-intensive parts are having control flows in them. Figure 1.1 shows profiling results when two representative multimedia applications are running on a single ARM 9 processor[1]. Pieces with red labels are data-intensive parts and the dagger symbol (†) indicates that the corresponding part has *if* structures. We can observe from the figure that lots of data-intensive parts consuming big portions of the total execution time have control flows in them.

Although accelerating both data- and control- ntensive parts is becoming an important issue, conventional CGRAs are not suitable to handle them. Since they have a single controller for the simplified control structure, it is impossible to execute programs having control divergence among PEs. The only way to overcome this architectural limitation is known as the technique called predicated execution [14]. However, conventional predications can threaten the competitiveness of CGRAs since it causes serious overhead in both performance and power consumption.

[1]We measured the execution time using ARM Developer Suite 1.2 [13].

Thus, in this thesis, we reveal performance and power issues of predicated execution and propose a novel mechanism to overcome drawbacks of the conventional techniques. The main contributions of the thesis include the following.

- We investigate power consumption related with predicated execution techniques for the first time not only in the domain of CGRAs, but in all domains related to predicated execution. Most of the previous research on predicated execution has concentrated only on performance improvement and design automation through architecture-level [15, 16] and/or compiler-level [17, 18] modifications, but no one has considered power consumption.

- We propose a low-power predication mechanism to mitigate power consumption overhead of predicated execution. Conventional full predication techniques require both additional instruction bits for instruction encoding and unnecessary decoding of instructions, which incur extra power consumption in configuration memory and processing elements, respectively. It reaches 32.0% on average over the main target applications (H.264 video CODEC).

- We propose a predication mechanism to accelerate execution of control flows. Conventional predication techniques have focused on “correct execution” of control flows, and thus have negative or no impact on performance. On the contrary, our approach not only correctly executes control flows, but also accelerates their execution through a technique called DISE (*Dual-Issue-Single-Execution*). Experiments show that DISE accelerates the main target applications by 15.1% on average.

- We also reveal mapping issues when using the proposed predication. A power-saving mode introduced into processing elements prohibits multiple conditionals from being parallelized if conventional mapping algorithms are used. Thus, we propose the framework to map conditionals separately. Experiments show that mapping results from the proposed approach lead to 2.21 times higher performance than those of the naïve approach.

Chapter 2

Background and Related Work

2.1 Coarse-Grained Reconfigurable Architecture

2.1.1 Introduction

As the size of semiconductor reaches physical limits but the amount of computation required in an application increases, it becomes more and more difficult to satisfy the performance requirement of an embedded system with just software running on a processor. Therefore, many embedded systems or System-On-Chips (SoCs) are equipped with one or more dedicated hardware IPs. Such hardware IPs may provide sufficient performance but lack flexibility, and so they should be redesigned from the scratch even when the required functionality changes slightly. Moreover, since hardware can hardly be fixed after fabrication, it needs much more time and effort put into the verification of the design before fabrication. Even further, considering that it is a trend to support various multimedia applications such as music or video codec in a cellular phone or a portable multimedia player, developing an IP for each application takes too much cost and development time.

One way to solve such a problem is to use a CGRA, which provides hardware-like performance through an array processing as well as software-like flexibility through reconfigurability. Instead of developing a new IP for a new application, to implement an application on a CGRA, one can just reconfigure the existing CGRA such that the CGRA provides the necessary functionality of the application. Due to these benefits, there have been many researches proposing various kinds of CGRAs [1–10] and surveys on them [11, 12].

2.1.2 Target Domain

CGRAs target both Instruction-Level Parallelism (ILP) and Data-Level Parallelism (DLP), but rely more on DLP to utilize abundant PEs since the amount of ILP is generally limited [19, 20]. Thread-Level Parallelism (TLP) is hardly supported due to the simplified control structure although there are some attempts [21, 22]. Thus, many works [3, 6, 23–25] have showed that CGRAs are suitable for multimedia applications, which are the traditional DLP applications. In addition, [26–28] have argued that CGRAs can support multiple domains, which include not only multimedia but also 3D graphics, and [29–31] have proved that the target domains can be extended to wireless communication system.

2.1.3 Comparison with Other Architectures

Compared to Very Long Instruction Word (VLIW) processors [32], CGRAs have much simpler control structures. This makes it hard to support dynamic behaviors like stalls, which requires interactions among PEs or between PEs and the controller. On the other hand, CGRAs have scalability and can have tens or hundreds of PEs while VLIWs generally have less than ten PEs due to complexity of such logics.

SIMD is well known to be effective and is used popularly [33, 34], but [35] recently reported the utilization problem in conventional SIMD architectures. They found that the amount of DLP is quite varying according to applications so PEs are wasted when applications do not have as much DLP as the number of PEs. In such cases, CGRAs can exploit ILP on the unused ones, resulting in better utilization. Instead, CGRAs have more complex hardware and mapping process than those of SIMD.

There is another kind of reconfigure architecture, which is Field-Programmable Gate Array (FPGA) [36,37]. FPGAs have bit-level reconfigurability rather than word-level one, which makes hardware much more optimized for specific applications. However, making configuration is much more complex and also takes longer in order to assign bit-level functionality to reconfigurable logics, and the area/power efficiency can be lower in cases where the applications consist mostly of word-level operations and so bit-level reconfigurability is neglected.

Many-core systems and CGRAs are similar in that both have many PEs. However, PEs of CGRAs are much simpler ones since they do not have autonomy and receive orders from a single shared controller. Thus, CGRAs have efficiency in exploiting ILP and DLP while many-core systems are suitable for TLP.

2.1.4 Application Mapping

To run applications on CGRAs, it is needed to generate configuration of CGRAs corresponding to the required functionality. The process of generating the configuration is called application mapping or just mapping in short. Since it is neither possible nor beneficial to execute the entire application on CGRAs, only data-intensive parts called kernels that have large parallelism are mapped to CGRAs. Most of the kernels are a loop where iterations have no dependency between them and thus can be executed in parallel or have some dependency but can be pipelined with short initiation interval. The functionality of a kernel is generally expressed as a CDFG (control data flow graph) and the kernel mapping is a process of generating configuration of CGRAs such that the PEs perform the operations in the CDFG while communication through the interconnections of PEs handles the data dependency among the operations.

Mapping of applications onto CGRAs have been researched actively. Some researched mapping methodology for better quality of solution or faster running time [38–45], and some want to integrate new issues on mapping process [16, 44, 46, 47].

2.1.5 Target CGRA

The target architecture is a CGRA called FloRA [9, 48]. It mainly targets the embedded system applications having ILP and DLP, which include multimedia applications such as video CODEC (MPEG4, H.264) and 3D graphics. It has been implemented on a chip and its functionality and performance have already been verified [48].

The overall architecture is shown in Figure 2.1. It consists of four main components: the PE array, configuration memory, data memory, and controllers. The PE array has 8×8 PEs, each of which is comprised of an integer ALU, a shifter, and a local register file and can be dynamically reconfigured every cycle if needed. The configuration memory contains configuration information (or instructions) used by the PE array, which defines not only the operation of each PE but also the interconnection among the PEs. Currently, the bit-width of the information used by one PE for each cycle is 20 bits and configuration memory can hold at most 3072 entries. The data memory stores the input/ output data used/generated by the PE array. It is accessible from the outside of the reconfigurable computing module through the bus, which enables host processors to provide data for the CGRA. The execution controller manages macro instructions which generate signals that control the execution of the CGRA at a macro level. A macro instruction controls issuing instructions (in the form of a configuration stream that specifies the start address and the end address in the configuration memory) and fetching/storing data from/into the data memory.

FloRA has three distinguished features compared to other CGRAs. First one is efficient resource sharing [9]. Multipliers and dividers are area-critical, but are used less frequently than ALUs. Thus, it is quite wasteful that each PE has its own such resources. To tackle this problem, [9] proposed the way that several PEs share area-critical resources.

Second, FloRA has the loop pipelining technique depicted in Figure 2.2 [49]. In this technique, configuration information is pipelined through the PEs in the same row, instead of being directly fetched from the configuration memory for each PE. It contributes to reducing the amount of configuration information,

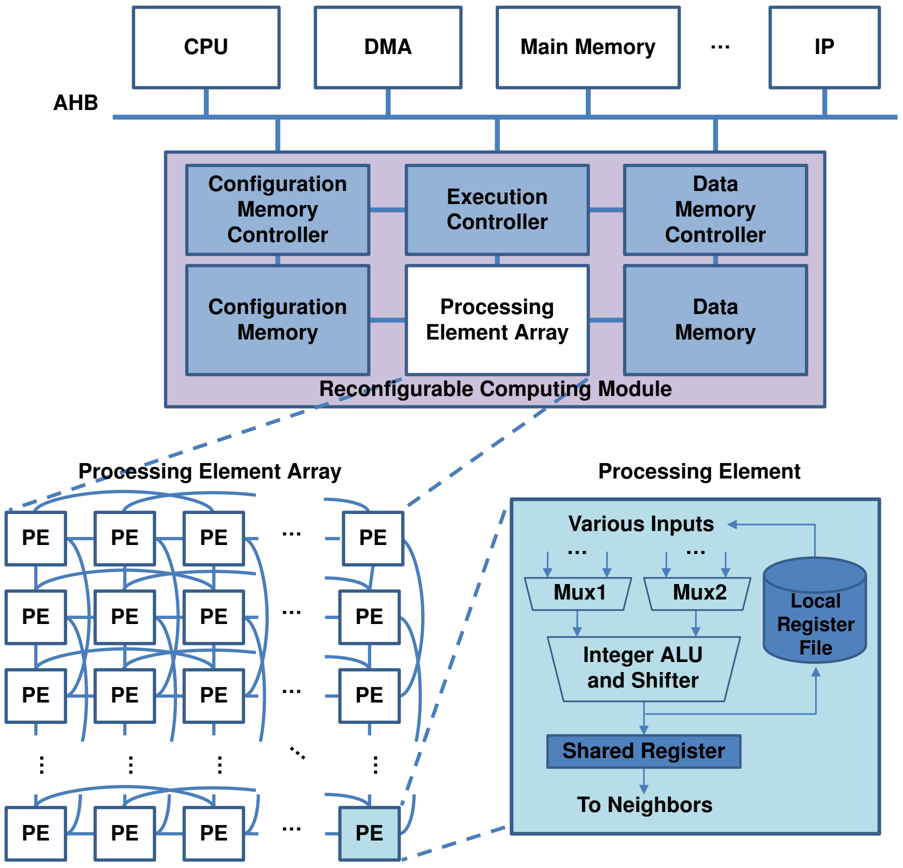


Figure 2.1: The target architecture FloRA.

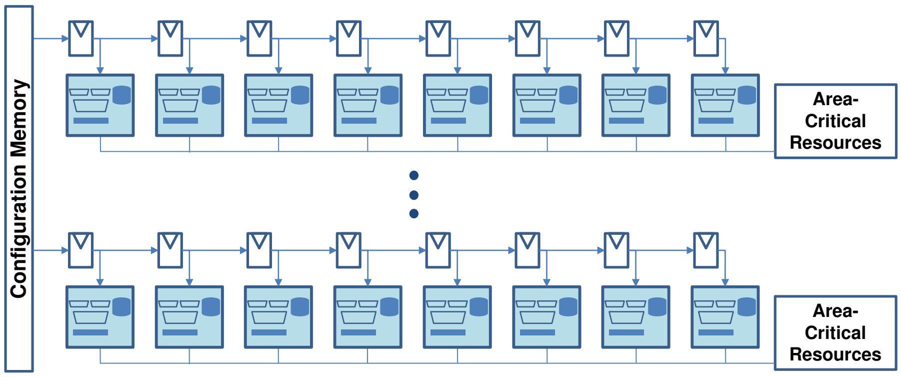


Figure 2.2: Architecture for loop pipelining technique for FloRA.

and thus saves power consumption in accessing the configuration memory. Also, it simplifies the programming model for shared area-critical resources by allowing each PE in the same row to use the shared resources in a round-robin manner.

Lastly, FloRA can perform floating-point operations by combining two integer PEs. If separate floating point units are integrated to architectures, they are useless when integer operations are performed. On the other hand, when floating point units are used, integer functional units will be wasted. Thus, this approach can save area significantly by utilizing integer functional units for floating point operations.

2.2 Predicated Execution Technique

2.2.1 Introduction

The existence of control flow in programs limits parallelism in two ways [50]. First, instructions having control dependences cannot be executed until they are released even though data dependences are released. Second, multiple conditionals have to be serialized due to lack of resources that are needed to handle control flow. Processors targeting fine-grained parallelism like ILP and DLP generally execute single thread at a time, and thus, lack branch handling units, program counters, and instruction memory ports, preventing several control flows from being handled concurrently.

To overcome this limitation and exploit more parallelism, predicated (or guarded) execution technique [14] or predication in short is adopted by processors [15, 18, 50–54]. It is a technique to converts control flows to data flows by modifying both architectures and compilers. It handles control flows by fetching all instructions but selectively executing them, rather than branching. A *predicate* indicates whether an instruction is executed or not, and *predication mechanism* denotes the way of determining a predicate. An instruction that can be nullified by a predication mechanism is called a *predicated instruction.*

2.2.2 Classification

[50] classified predication techniques into partial and full predication and compared their effects on performance in the domain of ILP, not of DLP. Its notion of full predication is limited to a narrow sense in that they consider only *condition-based full predication.* We extend their classification by introducing *state-based full predication* [54].

There have been several works related to the state-based full predication. [52] revealed that state-based full predication can virtually implement full predication only at the cost of partial predication. [51] emphasized that state-based full predication can handle *nested-if* structures, whereas condition-based full predication cannot.

2.2.3 Different Roles in ILP and DLP processors

Predication improves the performance of ILP processors directly and indirectly. Increasing parallelism by removing control dependences is a direct way, which is stated in Section 2.2.1. However, when an *if* structure is long, applying predication to it takes longer time than branching since predicated instructions always take cycles even if they are not executed actually. If predication is optionally given, however, speedup of programs is guaranteed by simply not using predication when it gives worse results than branching. The side effect of predication is related to branch prediction. It can remove hard-to-predict branches which cause frequent branch miss [55] and help increase the success ratio of branch prediction. The removal of branches may make it hard to predict remaining branches as branch predictors cannot use correlation information anymore, but the problem can be solved by predicate prediction [56].

On DLP processors including CGRAs, predication is a much more critical issue. If a loop body has control flows but an architecture does not support predication, the loop cannot be parallelized and each iteration should be executed sequentially. The following code shows an example of such programs.

```
for (i = 0; i < 8; i++)
{
  if (c[i] == 1)
    x[i] = 0;
  else
    x[i] = 1;
}
```

Each loop iteration requires different instruction flows, but each PE cannot select its own flow since multiple PEs are controlled by a single control unit. This problem, called *control divergence*, prevents programs requiring different control for each PE from being parallelized. Thus, in order to exploit DLP on such loops, it is essential to adopt predication into architectures [16, 18, 51–54, 57].

2.2.4 Predication Support on CGRAs

Partial predication is used for CGRAs in [57–59]. [57] showed how to implement partial predication efficiently, and [58] maximized performance by using partial predication in a speculative way. [59] presented automatic mapping framework proposed in [58]. On the other hand, [16] proposed an automatic way to accelerate control-intensive kernels by using condition-based full predication.

However, they all tried to use conventional predications instead of inventing a better predication, and [16] and [59] showed that only minimal efforts are needed for automatic mapping of control flows using conventional predications. By adopting *if*-conversion (e.g., [60]), control dependency can be converted to data dependency. As a result, conventional mapping algorithms do not need to handle control flow explicitly, but only need to consider data dependency as usual.

Chapter 3

Conventional Predicated Execution Techniques

Through this and next sections, we present detailed comparison of three existing predication techniques and two proposed ones. We discuss performance using 3-address form ISA, which can be extended to a general case. We also discuss power analysis for fetching/decoding/executing instructions in such a way that it can be adopted in other processing units, not restricted to the domain of CGRAs. The characteristics of CGRAs can affect pros and cons of each predication technique in some cases, but we explicitly separate such parts from ISA- or machine- independent parts.

To explain the mechanism of each predication technique, we use an example of C program shown in Figure 3.1a throughout this section. We assume that the code represents a part of a loop body and the variables `c[i]`, `x`, and `y` are stored in registers `R0`, `R1`, and `R2`, respectively. Also, we consider several types of *if*

```
1   if (c[i] == 1)
2   {
3     x = x+1;
4     y = y+1;
5   }
6   else
7   {
8     x = x-1;
9     y = y-1;
10  }
```

(a) C code

```
1         cmp R0 #1
2         b neq ELSE
3   IF:   add R1 R1 #1
4         add R2 R2 #1
5         b uc END
6
7
8   ELSE: sub R1 R1 #1
9         sub R2 R1 #1
10  END:
```

(b) branch

Figure 3.1: An example C code and its branch equivalent.

structures; *if-only* structures, *if-else* structures, and *nested-if* structures. Since each predication shows different characteristics according to these types, we will use *if-else* structure to show the basic mechanism and add the explanation for other types.

3.1 Partial Predication (Partial)

Predication is classically divided into two categories, *partial* and *full*, according to the range of predicated instruction [50]. *Partial predication* (Partial) simply adds a number of special predicated instructions such as *conditional mov* to ISA (Instruction Set Architecture), whereas full predication makes all instructions predicated using more architectural modification.

To emulate predication effects for normal instructions using some special ones, Partial first executes instructions for every control path and commits results from only one path selected according to the condition using the special predicated instructions, which is called a *transformation* process. For example, line 3 in Figure 3.1a is converted to two instructions as shown in Figure 3.2: one for storing the result of a normal addition to a temporary register R3 (line

	Assembly Code	`R0 == 0`		`R0 == 1`	
		Executed?	Flag	Executed?	Flag
1	`add R3 R1 #1`	Yes		Yes	
2	`add R4 R2 #1`	Yes		Yes	
3	`sub R5 R1 #1`	Yes		Yes	
4	`sub R6 R2 #1`	Yes		Yes	
5	`cmp R0 #1`	Yes	`lt`	Yes	`eq`
6	`cmov eq R1 R3`	**No**	`lt`	Yes	`eq`
7	`cmov eq R2 R4`	**No**	`lt`	Yes	`eq`
8	`cmov neq R1 R5`	Yes	`lt`	**No**	`eq`
9	`cmov neq R2 R6`	Yes	`lt`	**No**	`eq`

Figure 3.2: Partial predication (PARTIAL).

1) and the other for committing it if the `eq` flag (in the status register) is set (line 6).

The main advantage of PARTIAL is minimal architectural support which is to add a few instructions to an ISA as stated before. Since an ISA usually has spare encoding space so that the designer can add new instructions, this approach does not need many changes in the hardware structure. On the other hand, it has lower performance than other predication mechanisms since the transformation inserts predicated instructions additionally and increases register pressure [50].

The increased register pressure could threaten potential speedup achievable through CGRAs. Originally CGRAs do not have many registers since they are typically targeted for data-intensive kernels[2], where the live ranges of most variables are relatively short and not much overlapped with each other. Also, the high cost of registers in terms of area and power consumption hinders the number of registers from being increased. Consequently, it drives the user to

[2]CGRAs and SIMD processors may be targeted to accelerate the control-intensive part of programs as well as the data-intensive part, but they generally have two separate hardware modules, one for each part [7, 34, 61, 62].

rely on software-level approaches, which generally degrade performance.

Register spill is the most popular and intuitive way to solve the problem, but it is not appropriate for CGRAs since a limited number of load-store units and a single, centralized control unit act as a bottleneck to spill registers. For example, the target CGRA requires at least 12 cycles to read the spilled data back. Instead, we can allocate more PEs to provide more registers per loop body. For example, if a loop body has 12 variables and each PE has only eight registers, executing it on only a single PE would spill four variables, whereas mapping the loop body into two PEs completely eliminates the need for spilling. However, it could still have lower performance compared to other approaches since the PEs may be underutilized, or it may make it impossible to map bigger applications. In the experiments, average 1.75 more registers are required per PE, and one example (`itpl`) cannot be accelerated by the CGRA since it has 16 divergent paths.

In addition, PARTIAL could incur additional overhead in energy consumption because it executes even *unnecessary paths* whose results will not be taken. This makes the execution units including ALUs and register files consume up to 2.88 times (1/(1-0.653)) energy compared to other predication techniques that do not execute instructions on unnecessary paths, which is shown in Section 5.3.1. The power gap becomes even larger if PARTIAL is used for *switch* statements having multiple paths that need not be executed.

3.2 Condition-Based Full Predication (CONDFULL)

Unlike partial predication, full predication makes all instructions predicated. *Condition-based full predication* (CONDFULL) is a type of full predication that

Assembly Code	`R0 == 0`		`R0 == 1`	
	Executed?	Flag	Executed?	Flag
`uc  cmp R0 #1`	Yes		Yes	
`eq  add R1 R1 #1`	**No**	`lt`	Yes	`eq`
`eq  add R2 R2 #1`	**No**	`lt`	Yes	`eq`
`neq sub R1 R1 #1`	Yes	`lt`	**No**	`eq`
`neq sub R2 R2 #1`	Yes	`lt`	**No**	`eq`

Figure 3.3: Condition-based full predication (CondFull).

introduces an additional field called *condition operand* in all instructions. This condition operand is compared to the flag in each PE and the result determines whether the corresponding instruction should be executed or not. In other words, each instruction is annotated by its own activating condition. For example, condition operands `uc`, `eq`, and `neq` in Figure 3.3 denote that the corresponding instruction is executed unconditionally, only if the `eq` flag is set, and only if the `neq` flag is set, respectively.

Compared to Partial, CondFull does not have overhead in performance or energy consumption since it does not require the transformation process [50]. However, there are several reasons that make CondFull increase the overall energy consumption. First, CondFull requires additional instruction bits used for condition operands. It affects both dynamic and static energy since it increases the capacity of the configuration memory. This could eventually lead to an increase in energy consumption even when the architecture executes normal programs that do not use the predication effect. In addition, although it eliminates the need for executing instructions on unnecessary paths (refer to Partial), it still needs to decode the instructions to check the condition operands in them.

C Code	`a == 0`		Assembly Code	`R0 == 0`	
	Executed?			Executed?	Flag
`if (a == 1)`	Yes	1	`uc  cmp R0 #1`	Yes	
`  if (b == 1)`	No	2	`eq  cmp R1 #1`	No	`lt`
`    x = 0;`	No	3	`eq  mov R2 #0`	No	`lt`
`  else`		4			
`    x = 1;`	No	5	`neq mov R2 #1`	**Yes**	`lt`

(a) naïve conversion

C Code	`a == 0`		Assembly Code	`R0 == 0`	
	Executed?			Executed?	Flag
`if (a == 1 && b == 1)`	Yes	1	`uc  testeq  R3 R0 #1`	Yes	
		2	`uc  testeq  R4 R1 #1`	Yes	
		3	`uc  and  R4 R3 R4`	Yes	
		4	`uc  cmp  R4 #1`	Yes	`lt`
`  x = 0;`	No	5	`eq  mov  R2 #0`	No	`lt`
`if (a == 1 && b != 1)`	Yes	6	`uc  testneq R4 R1 #1`	Yes	`lt`
		7	`uc  and  R4 R3 R4`	Yes	`lt`
		8	`uc  cmp  R4 #0`	Yes	`lt`
`  x = 1;`	No	9	`neq mov  R2 #1`	No	`lt`

(b) flattened

Figure 3.4: An example program that naïve conversion to CondFull produces incorrect assembly code due to its *nested-if* structure.

Also, CondFull has another limitation in handling *nested-if* structures. Since the mechanism is based on condition operands, it can express only one control flow at a time. In other words, the flag that controls the execution of a predicated instruction is determined only by the most recent comparison instruction, and that makes CondFull hard to handle *nested-if* structures. As an example, Figure 3.4a shows a program that is executed incorrectly with naïve conversion to CondFull. Assuming that variables `a`, `b`, and `x` are stored in the registers `R0`, `R1`, and `R2`, respectively, the assembly code generates incorrect results when the value of `a` (i.e., `R0`) is zero.

To overcome this limitation, CondFull uses a *flattening* technique in software level, in which *nested-if* structures are converted into non-*nested* ones during architecture-specific optimization in the backend process. For example, the code in Figure 3.4a could be converted to the one in Figure 3.4b. However, this flattening technique could incur performance overhead as can be seen in the figure. This is mainly because it needs to recalculate flags for each combination. Moreover, it makes register pressure higher due to the need for extra temporary registers used to keep intermediate values for calculating flags (see `R3` and `R4` in the figure). For example, the `deblock` application has an *if* structure nested four times, so flattening causes about 13% overhead compared to the proposed predication technique that can support nested structures.

Chapter 4

State-Based Full Predication

State-based full predication is yet another type of full predication proposed in [54]. It uses a shared state to make instructions predicated instead of adding a condition operand to each instruction. In this mechanism, only a few special instructions are added to manage this shared state, thereby minimizing changes to the architecture. Nevertheless, since the state affects the entire set of instructions, state-based full predication virtually obtains the effect of full predication without any modification of normal instructions. To achieve this, an 1-bit state register is added to each PE, which indicates either *awake* or *sleep* state. PEs execute instructions normally in AWAKE state, whereas they nullify every instruction in SLEEP state. Therefore, by controlling the state of each PE using a special instruction, either the if-path or else-path can be executed selectively.

	Assembly Code	`R0 == 0`			`R0 == 1`		
		State	Tag	Flag	State	Tag	Flag
1	`cmp   R0 #1`	AWAKE			AWAKE		
2	`sleep neq END_IF`	AWAKE		`lt`	AWAKE		`eq`
3	`add   R1 R1 #1`	**SLEEP**	`END_IF`	`lt`	AWAKE		`eq`
4	`add   R2 R2 #1`	**SLEEP**	`END_IF`	`lt`	AWAKE		`eq`
5	`sleep uc END_ELSE`	**SLEEP**	`END_IF`	`lt`	AWAKE		`eq`
6	`awake END_IF`	**SLEEP**	`END_IF`	`lt`	**SLEEP**	`END_ELSE`	`eq`
7	`sub   R1 R1 #1`	AWAKE		`lt`	**SLEEP**	`END_ELSE`	`eq`
8	`sub   R2 R2 #1`	AWAKE		`lt`	**SLEEP**	`END_ELSE`	`eq`
9	`awake END_ELSE`	AWAKE		`lt`	**SLEEP**	`END_ELSE`	`eq`

Figure 4.1: Instruction-based wake-up for state-based full predication (PseudoBranch).

4.1 Previous Approach (PseudoBranch)

To control the state of each PE, [51] proposed to use *sleep* and *awake* instructions (PseudoBranch). It associates each sleep instruction with a tag in order to support *nested-if* structures without flattening them. Each PE has a *tag register* to store the tag of the most recently executed sleep instruction. When a sleep instruction is invoked, the PE checks the status flag and enters into SLEEP state with saving a tag to the register if the flag value satisfies the condition. Once the PE enters into SLEEP state, it wakes up only when an awake instruction with the same tag is invoked.

Figure 4.1 shows an example of using PseudoBranch. "`sleep neq END_IF`" in the line 2 denotes a sleep instruction that changes the state of the PE to SLEEP state and saves tag as END_IF if the `neq` flag is set. Awake instructions in the line 6 and 9 have tags as an operand. It can be seen that in the case of `R0 == 1` the awake instruction in line 6 has no effect since the tag of the PE does not match to the one in the awake instruction.

PseudoBranch can support *nested-if* structures compared to conventional predications, but it has performance overhead in non-*nested* structures since it requires two additional instructions per *if* structure. In addition, it will consume as much power as CondFull since it needs to decode every instruction even in SLEEP state to check for an awake instruction.

4.2 Counter-Based Approach (StateFull)

We propose a novel state-based full predication, which uses *per-PE counters* and a *sleep* instruction for state transition (StateFull). The sleep instruction changes the state of a PE from AWAKE state to SLEEP state, which has two operands: *condition* and *offset*. When a PE enters SLEEP state, the per-PE counter is initialized to the value of the offset operand so that the PE returns to AWAKE state after skipping as many instructions as specified in the offset operand (i.e., *sleep period*). The sleep instructions have similar semantics to branch instructions, except that PEs simply ignore (neither decode nor execute) the instructions during SLEEP state instead of jumping to a new address by modifying the program counter.

Figure 4.2 shows the mechanism of StateFull. "`csleep neq 3`" is a sleep instruction that makes the PE sleep during the next three instructions if the `neq` flag is set. For example, if `R0` is not 1, the sleep instruction in line 2 is activated (see State column) to put the PE into SLEEP state and keep it in that state until the counter becomes zero. During the sleep period, the counter keeps track of the number of instructions to be skipped including the current instruction.

There can be some issues related to multicycle operations or stalls. First,

	Assembly Code	`R0 == 0`			`R0 == 1`		
		State	Counter	Flag	State	Counter	Flag
1	`cmp R0 #1`	AWAKE			AWAKE		
2	`csleep neq 3`	AWAKE		`lt`	AWAKE		`eq`
3	`add R1 R1 #1`	**SLEEP**	3	`lt`	AWAKE		`eq`
4	`add R2 R2 #1`	**SLEEP**	2	`lt`	AWAKE		`eq`
5	`csleep uc 2`	**SLEEP**	1	`lt`	AWAKE		`eq`
6	`sub R1 R1 #1`	AWAKE		`lt`	**SLEEP**	2	`eq`
7	`sub R2 R2 #1`	AWAKE		`lt`	**SLEEP**	1	`eq`

Figure 4.2: Counter-based wake-up for state-based full predication (StateFull).

we assume that no operations require varying number of cycles (e.g., data-dependent operation cycles) since such operations can be hardly supported by CGRAs or other parallel architectures having a single controller and passive PEs. It is almost impossible for the controller to adjust the execution of PEs considering the status of all PEs. Second, fixed-multicycle operations cause no problem in the proposed approach since they are converted to the consecution of several single-cycle operations in the scheduling phase due to exactly the same reason as the first one. Thus, a PE does not need to consider multicycle operations at all. Lastly, the architecture can be stalled dynamically due to runtime conditions. For example, ADRES architecture [7] should be stalled if data read operations cause bank conflicts. In such situations, however, we can simply solve the problem by stalling the counters, too.

StateFull improves performance over PseudoBranch, since it does not require awake instructions which cause performance overhead. On the other hand, since it still needs to insert special sleep instructions to control the states of PEs, it could incur performance overhead compared to CondFull for programs that do not have any *nested-if* structure. This performance overhead can be relatively large for *short-if* statements (*short-if* means the body of the *if* structure is short; refer to Section 5.3.2 for the definition). However, StateFull provides better performance than CondFull for programs having *nested-if* structures, since it naturally can handle them without flattening. In StateFull, nesting of *if* structures does not further increase the register pressure nor the number of instructions.

Moreover, StateFull contributes to reducing power consumption of PEs compared to CondFull. A major source of the reduction is that a PE knows a priori whether the next instruction will be executed or not before decoding the instruction. This is possible since the predicate is determined solely by the state register thereby completely eliminating the need for decoding the next instruction. Thus, when the PE is in SLEEP state, it does not need to do anything but just counts down until it wakes up. We exploit this observation by activating only some small logic circuits for counters and blocking unnecessary switching in registers and combinational circuits, including instruction registers and decoders. This can lead to a huge reduction in dynamic power consumption if it is implemented through clock-gating techniques. Note that it is impossible for CondFull since the PEs recognizes the predicate only after the instruction has been decoded and the corresponding condition has been evaluated. As a result, the proposed approach reduces power by 43.4% compared to CondFull (refer to Section 5.3.1).

In addition, it could reduce the power consumption of configuration memory as well. Considering the fact that StateFull does not require adding any additional field to the instructions, it uses smaller configuration memory compared to that for CondFull. Thus it reduces the power consumption of the

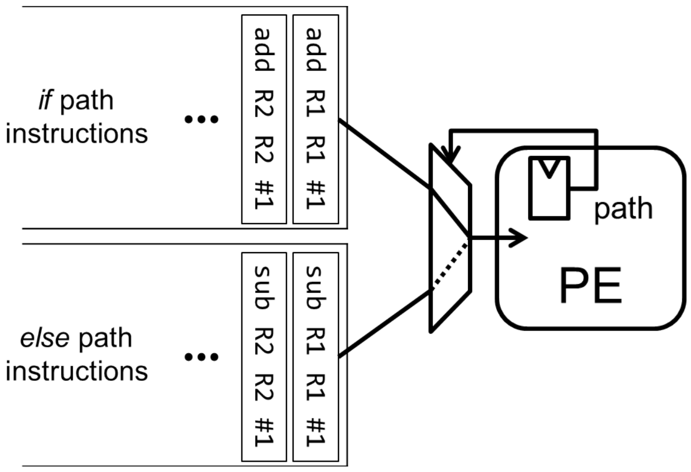


Figure 4.3: The concept of the DISE technique.

configuration memory as well.

4.3 Dual-Issue-Single-Execution (DISE)

We propose another novel approach called *Dual-Issue-Single-Execution* (DISE) to accelerate execution of control flows. Figure 4.3 summarizes this concept. Considering that only one path (branch) is taken for each *if-else* construct, it is possible to issue instructions from both paths at the same time and let the PE execute only one path depending on the predicate. There is an internal 1-bit state register called *path register* for each PE to keep track of the path that the PE is currently executing.

The value of the path register is toggled by a predicated instruction named `changepath`. To provide enough instruction bandwidth, the PE fetches two instructions at each cycle, one for *true-path* and the other for *false-path*. Between them, the PE selects true-path instruction for execution if the value of the path register is TRUE, and selects the false-path one, otherwise.

If the true-path is longer than the false-path, then the false-path should be filled up with dummy nop instructions, and vice versa. Adding the dummy nop instructions is done during compile time and thus increases the code size thereby taking more configuration memory space. In case of *normal code* that has no control flow in it[3], all the original instructions are put into the true-path (path register is set to TRUE) and the false-path is filled up with nop instructions. In this mechanism, however, DISE could waste lots of configuration memory space due to the increased code size. For example, the `round` application in the experiments has an *if-else* structure that takes 16.7% of the entire execution time. It means that nop instructions should be inserted into the false-path for the normal code part, which takes 83.3%, so the dynamic energy on configuration memory will be increased by 83.3% due to more instruction fetches.

To avoid this problem, we handle the two code parts differently by introducing two instruction-fetch modes. Normal code is fetched in the *normal mode*, where only one instruction is fetched and assigned to the true-path, and if-else code is fetched in the *DISE mode*, where two instructions are fetched (refer to Section 5.1.2 for the implementation detail).

Figure 4.4 shows an example of DISE. "**`changepath neq`**" flips the value of the path register if the specified flag is asserted. In the case of `R0 == 0`, the path register, which is initially set to TRUE to fetch normal code, is changed to FALSE by the changepath instruction in line 2 since the given predicate is true. After that, three instructions from the false-path are executed instead of

[3]Hereafter, we call the portions of the program corresponding to if-else paths as *if-else code* and all others as *normal code*.

	Assembly Code		`R0 == 0`		`R0 == 1`	
	True-path	False-path	Path	Flag	Path	Flag
1	`cmp R0 #1`		TRUE		TRUE	
2	`changepath neq`		TRUE	`lt`	TRUE	`eq`
3	`add R1 R1 #1`	`sub R1 R1 #1`	**FALSE**	`lt`	TRUE	`eq`
4	`add R2 R2 #1`	`sub R2 R2 #1`	**FALSE**	`lt`	TRUE	`eq`
5	`nop`	`changepath uc`	**FALSE**	`lt`	TRUE	`eq`

Figure 4.4: Dual-Issue-Single-Execution (DISE).

true-path instructions until another changepath instruction in the false-path is invoked to return to the true-path, indicating the termination of *if-else code.*

The major objective of DISE is to accelerate the execution of control flows. The previous approaches focus mainly on "correct execution" of control flows and their execution time is proportional to the total number of instructions in the if-else code. However, since DISE issues two instructions—one from the true-path and the other from the false-path—at every cycle, its execution time depends on the number of instructions in the longest side between if- and else-paths. This can be achieved without any additional functional unit, but with minimal modification in the memory structure, causing only 2% area overhead [53]. DISE is also preferable in terms of performance to the existing technique that uses more PEs to accelerate the execution of control flows by executing if- and else-paths at the same time and then selecting the right result [58]. Although their technique may reduce the latency of one iteration, it does not improve the throughput when the target application has enough parallelism.

As mentioned in Section 4.2, StateFull reduces dynamic power consumption by not decoding instructions during sleep periods. Similarly, DISE reduces dynamic power consumption since only instructions in either of the two paths are decoded. Note that dual-issuing does not increase the number of fetched instructions. Although it issues twice as many instructions as other predication techniques for each cycle, the total number of fetched instructions over the entire execution remains almost the same (ignoring the extra instructions for DISE, which take typically a very small portion), which is the sum of the lengths of if- and else-paths. Hence, DISE consumes almost the same dynamic energy as that of StateFull except the overhead to the circuits added to support dual-issuing and doubled port width of configuration memory. Fortunately, we develop several techniques to keep the overhead very low (2.4% power overhead in the reconfigurable array and configuration memory) as detailed in Section 5.3.6. Moreover, DISE could eventually reduce energy consumption even with the additional circuit overhead through clock/power gating during the slack period obtained by performance improvement.

However, it could be inefficient in the case that the lengths of if- and else-paths are unbalanced. The technique requires balancing the lengths of the two paths, and thus extra nop instructions are inserted for unbalanced if-else paths as depicted in Figure 4.5. This incurs increase in code size, and eventually leads to larger dynamic energy consumption in the configuration memory. This problem should be addressed carefully since real programs may have a considerable number of unbalanced *if-else* structures or even *if-only* structures making DISE inefficient.

Besides, DISE has a limitation in applying it to *nested-if* structures. This is because the mechanism uses path registers to choose the right instructions to be executed, which can handle only one control flow at a time. Therefore, this

C Code		Assembly Code	
		True-path	False-path
`if (x[i] == 1)`	1	`cmp R0 #1`	
`{`	2	`changepath neq`	
`  a = a + 1;`	3	`add R1 R1 #1`	`sub R1 R1 #1`
`  b = b + 1;`	4	`add R2 R2 #1`	*nop*
`  c = c + 1;`	5	`add R3 R3 #1`	*nop*
`  d = d + 1;`	6	`add R4 R4 #1`	`changepath uc`
`}`	7		
`else`	8		
`  a = a - 1;`	9		

Figure 4.5: Additional nop instructions inserted for DISE to balance the lengths of if- and else-paths.

mechanism also requires flattening the *nested-if* structures as in CondFull, and thus would cause performance degradation. However, its effect is smaller than that in CondFull since it has the capability of accelerating the execution of control flows thereby compensating the negative effect of flattening.

4.4 Hybrid Predication

4.4.1 Motivation

Each predication has pros and cons. CondFull shows relatively good performance in both *short-if* and *long-if* structures, but it incurs notable overhead in energy consumption and requires significant modification of the existing ISA. As an alternative, StateFull could be chosen since it does have significantly lower energy consumption while maintaining or even improving the performance over CondFull. However, it performs poorly in the case of *short-if* structures, which could be a serious problem since *short-if* structures take a considerable portion of control flows in real programs. Other predication techniques such as Partial and DISE are not appropriate to be used solely since their charac-

Table 4.1: Characteristics of Predicated Execution Techniques

	ISA	Config. Mem.	Performance			Energy Consumption		
			Short-if	Long-if	Nested-if	Short-if	Long-if	Nested-if
Partial	+insts	Identical	Moderate	Slow	Slow	Moderate	High	High
CondFull	width ↑	Bigger	Fast	Fast	Slow	Moderate	High	High
StateFull	+insts	Identical	Slow	Moderate	Fast	High	Low	Low
DISE	+insts	Identical	Moderate	Very Fast	Moderate	Moderate	Low	Moderate
StateFull+Partial	+insts	Identical	Moderate	Moderate	Fast	Moderate	Low	Low
StateFull+Partial+DISE	+insts	Identical	Moderate	Very Fast	Very Fast	Moderate	Low	Low

teristics are rather specialized to specific kinds of programs (*short-if* structures for Partial and balanced *if-else* structures for DISE). The first four rows of Table 4.1 summarize the characteristics of predication techniques introduced in the previous section.

Therefore, we propose to combine StateFull with Partial and/or DISE. The key idea behind this is to compensate the weakness of StateFull by adopting other predication techniques, and so, to bring synergistic effects in terms of both performance and energy consumption as shown in the last two rows of Table 4.1. Also they could be easily integrated into the architecture without interfering with each other as they are implemented as special instructions rather than modifying the entire ISA. Note that combining CondFull with Partial is not beneficial since Partial does not provide any benefit over CondFull. Hybridizing CondFull with DISE is not beneficial either since neither DISE nor CondFull can execute *nested-if* structures efficiently. Lastly, the combination of CondFull and StateFull is inefficient in terms of power consumption as both provide similar benefits from full predication itself and CondFull causes lots of overhead in configuration memory.

C Code	StateFull	Partial

`if (x[i] > 255)`
 `x[i] = 255;`
	`mov R1 #255`
	 `cmp R0 R1`
	 `csleep leq 1`
	 `mov R0 R1`
		`mov R1 #255`
		 `cmp R0 R1`
		 `cmov gt R0 R1`

Figure 4.6: An example program that Partial executes with better performance than StateFull.

4.4.2 StateFull+Partial

StateFull inserts sleep instructions into the code to control *if* structures, which incurs overhead in energy consumption as well as performance. For *long-if* structures, reduction in energy consumption on unnecessary paths is usually large enough for compensating the overhead of sleep/awake instructions. However, this overhead could eventually lead to an increase in total energy consumption in the case of *short-if* structures, in which energy reduction on unnecessary paths is small.

To mitigate this overhead, we propose to incorporate Partial into StateFull to cover *short-if* structures with very low overhead in performance and energy consumption. This is based on the observation that `mov`-only *if* structures are common for *short-if* structures, for example, 79.4% of *short-if* structures in three examples (`idct`, `chroma`, and `max`) are composed of only `mov` operations, and Partial can handle `mov`-only *if* structures without performance overhead by just replacing `mov` instructions with `conditional mov` instructions, as shown in Figure 4.6.

However, Partial is not always preferred against StateFull in handling *short-if* structures, especially when they contain instructions other than `mov` instructions. This is because the transformation of *short-if* structures containing non-`mov` instructions could degrade performance and consume more energy due to unnecessary execution of instructions, as explained in Section 3.1. This is the reason why we decide to use PARTIAL only for `mov`-only *short-if* structures in a software manner. This strategy is investigated through experiments in Section 5.3.3.

This approach is not ISA specific. Almost all ISAs have `mov` instructions and adding a `conditional mov` instruction into an ISA is a representative implementation of partial predication [50]. Thus, `mov` instructions can be easily transformed to `conditional mov` instructions added to general ISAs. Note that the strategy may have to be changed if a different type of partial predication is implemented (e.g., partial predication based on `select` instructions [50]).

4.4.3 STATEFULL+PARTIAL+DISE

We propose to combine DISE with STATEFULL to further improve performance and energy efficiency. This is remarkably beneficial because DISE can effectively accelerate simple *if-else* structures while STATEFULL efficiently covers the case of *nested-if* structures. STATEFULL is very effective even for *if-only* structures (as well as *nested-if* structures), which are not efficiently handled by DISE. Using DISE for unbalanced *if-else* structures has a problem of increased code size because the extra nop instructions are inserted to balance the true- and false-path. This eventually leads to the increase in dynamic power consumption due to the extra instruction fetches.

To solve this problem, we propose a hardware-level solution to handle unbalanced *if-else* structures in a power-efficient way. The solution is adding a predicated instruction that has the capability of changing the state of a PE

	Assembly Code		`R0 == 0`			
	True-path	False-path	State	Counter	Path	Flag
1	`cmp R0 #1`		AWAKE		TRUE	
2	`changepath neq`		AWAKE		TRUE	lt
3	`add R1 R1 #1`	`sub R1 R1 #1`	AWAKE		FALSE	lt
4	`add R2 R2 #1`	`changepath_csleep uc 2`	AWAKE		FALSE	lt
5	`add R3 R3 #1`		**SLEEP**	2	**TRUE**	lt
6	`add R4 R4 #1`		SLEEP	1	TRUE	lt

Figure 4.7: A solution to eliminate the need for nop instructions. This is the result when the solution is applied to the example code shown in Figure 4.5.

to SLEEP state and, at the same time, alternating between true-path and false-path. This special instruction, `changepath_csleep`, eliminates the need for extra nop instructions completely thereby reducing the size of code so dynamic power consumption as well. Figure 4.7 shows an example of using the special instruction. Instead of filling the false-path (which is shorter than the true-path) with nop instructions, a `changepath_csleep` instruction is inserted right after the termination of the false-path. Note that changing the path right after the termination of the false-path may not work correctly since it would result in executing the remaining instructions in the true-path. For example, changing the path right after the sub instruction in Figure 4.7 would make the PE execute two add instructions, which leads to incorrect execution of the program. That is why it sleeps for two cycles before changing the path.

Basically, in *nested-if* structures, DISE can be applied to either the outermost or innermost *if-else* structures and the rest should be covered by StateFull. However, such structures can be handled even more efficiently when DISE and StateFull are applied together. Figure 4.8 shows such an example. Figure 4.8a shows the C code and Figure 4.8b shows the case of applying DISE to the outermost *if-else* structure while Figure 4.8c shows the case of applying it to the innermost one. Figure 4.8d shows how DISE can be applied several times to further improve the performance by moving multiple code blocks to the false-path side. Note that in all three cases, converting StateFull to DISE improves the performance but maintains the same number of instructions as that of the StateFull-only case; applying DISE just requires the change of the predicated instruction type (`csleep` to `changepath`) but does not require additional instructions. Therefore, we can accelerate nested structures effectively by selecting the one giving the maximum performance among the candidates[4].

Consequently, these software- and hardware-level techniques help DISE to overcome its weakness. Moreover, we could further improve the technique by putting Partial together with it to cover the weakness of StateFull on *short-if* structures as discussed in the previous section. Therefore, we use the technique StateFull+Partial+DISE, as a universal solution to predicated execution in terms of both performance and power.

[4]Identifying the best candidate efficiently remains to be a problem to be solved. In the experiments, we tried all combinations of else code blocks to be moved and chose the one with highest performance among the feasible (no overlap in the false-path) combinations.

C Code		StateFull
`if (x[i] == 1)`	1	`cmp R0 #1`
`{`	2	`csleep neq 10`
`  d = 0`	3	`mov R5 #0`
`  if (y[i] == 1)`	4	`cmp R1 #1`
`  {`	5	`csleep neq 4`
`    a = 0;`	6	`mov R2 #0`
`    b = 0;`	7	`mov R3 #0`
`    c = 0;`	8	`mov R4 #0`
`  }`	9	`csleep uc 2`
`  else`	10	`mov R2 #1`
`    a = 1;`	11	`mov R3 #1`
`    b = 1;`	12	`csleep uc 1`
`}`	13	`mov R5 #1`
`else`	14	
`  d = 1;`	15	

(a) example

	True-path	False-path
1	`cmp R0 #1`	
2	`changepath neq`	
3	`mov R5 #0`	`mov R5 #1`
4	`cmp R1 #1`	`changepath_csleep uc 7`
5	`csleep neq 4`	
6	`mov R2 #0`	
7	`mov R3 #0`	
8	`mov R4 #0`	
9	`csleep uc 2`	
10	`mov R2 #1`	
11	`mov R3 #1`	

(b) applying DISE into the outermost *if-else* structure

	True-path	False-path
1	`cmp R0 #1`	
2	`csleep neq 7`	
3	`mov R5 #0`	
4	`cmp R1 #1`	
5	`changepath neq`	
6	`mov R2 #0`	`mov R2 #1`
7	`mov R3 #0`	`mov R3 #1`
8	`mov R4 #0`	`changepath uc`
9	`csleep uc 1`	
10	`mov R5 #1`	

(c) applying DISE into the innermost *if-else* structure

	True-path	False-path
1	`cmp R0 #1`	
2	`changepath neq`	
3	`mov R5 #0`	`mov R5 #1`
4	`cmp R1 #1`	`changepath_csleep uc 4`
5	`changepath neq`	
6	`mov R2 #0`	`mov R2 #1`
7	`mov R3 #0`	`mov R3 #1`
8	`mov R4 #0`	`changepath uc`

(d) applying DISE into both *if-else* structures

Figure 4.8: Applying DISE and StateFull together into *nested-if* structures for better performance.

Chapter 5

Evaluation

5.1 Implementation

5.1.1 Conventional Techniques

Partial Predication (Partial)

We added a *conditional mov* (`cmov`) instruction to the ISA for Partial. Instead of the `cmov` instruction, a select instruction could have been implemented for the purpose of Partial. However, it was impossible to integrate the select instruction into the target architecture because it had four operands, whereas the ISA allowed only three-address form.

Condition-Based Full Predication (CondFull)

For the implementation of CondFull, we appended a 3-bit condition operand to each instruction and implemented a mechanism to nullify instructions on unnecessary paths by generating signals to disable writes into the registers and latches (disabling writes into the latches keeps the functional units from unnecessary switching). Due to this, the length of each instruction was increased from 20 bits to 23 bits and the capacity of the configuration memory was increased from 7.5 KB (=3072×20bits) to 8.625 KB (=3072×23bits).

Another State-Based Full Predication (PseudoBranch)

We tried to implement PseudoBranch as proposed in [51]. We added `sleep`/ `awake` instructions to the ISA, and added an 1-bit state register (representing either AWAKE or SLEEP state) and a 5-bit tag register to each PE. However, as explained in Section 4.1 and Section 4.2, PseudoBranch cannot save power with extra logic.

5.1.2 Proposed Techniques

State-Based Full Predication (StateFull)

We added a `csleep` instruction to the ISA. To keep track of information on sleep states, we added an 1-bit state register (representing either AWAKE or SLEEP state) and an 8-bit sleep period counter (in short, sleep counter) for each PE. This sleep counter was implemented by extending the existing 3-bit counter, which had been originally introduced to support multicycle operations such as multiplication. The counter could be shared like this since the existing counter would have not been used otherwise during SLEEP state. The 1-bit state register is used for gating clock signals of all registers (except the sleep counter) to prevent unnecessary bit changes during SLEEP state, thereby reducing dynamic power consumption dramatically.

Since we use an 8-bit sleep counter, it limits the maximum sleep period to 256 instructions. However, it is sufficient for the target applications having DLP as they do not have extremely *long-if* structures in most cases. In the ten

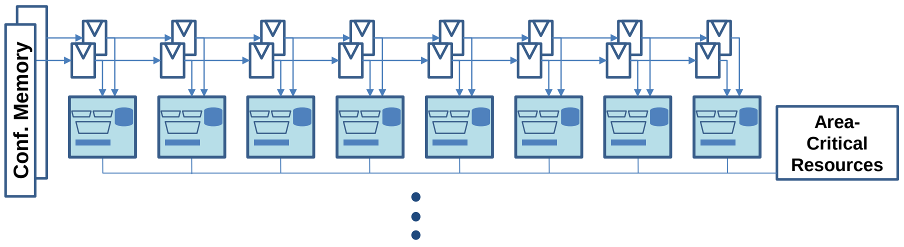


Figure 5.1: Modification of architecture to support DISE.

applications, the longest *if* structure consists of 143 instructions. Note that this does not imply that the target architecture cannot execute longer *if* structures. Rather, such *long-if* structures can be handled by applying software-level techniques such as splitting a long sleep period into multiple, short sleep periods having at most 256 instructions.

Dual-Issue-Single-Execution (DISE)

The implementation of the DISE technique requires adding one more memory bank and datapath to fetch one more instruction. The actual implementation can be varying according to the baseline architectures since they have their own ways of configuration, but Figure 5.1 shows how it is combined with the feature shown in Figure 2.2. The instruction registers, which pipeline instructions through the PEs in the same row, were doubled and the configuration memory was divided into two banks (thus having the same capacity, 1536 entries for each bank). We also added a `changepath` instruction to the ISA and incorporated an 1-bit path register and a two-to-one 20-bit multiplexer into each PE to select the instructions to be executed.

To control the number of instructions issued per cycle (two for if-else code and one for normal code), it was necessary to modify the execution controller of

FloRA. Since the execution of FloRA was controlled by macro instructions, we implemented this feature by adding a new type of macro instructions to fetch two instructions (instead of one) in each cycle. For example, if a program is composed of a normal code block (A), an if-else code block (B), and another normal code block (C) in sequence, it is executed by three macro instructions in sequence: `fetch A`, `fetch_DISE B`, and `fetch C`.

Also, we developed an efficient way to fully utilize both banks of the configuration memory. A naïve way of implementing DISE is to place true-path code and false-path code into the first and the second bank, respectively. However, it makes the second bank underutilized in the case of normal code since it is located only in the true-path by default. To solve this problem, we placed the i-th instruction in normal code into the (i mod 2)-th bank as in interleaved memory so that normal code instructions reside in both banks. Then, we let the execution controller give PEs the 2-bit information: the 1-bit control signal indicating whether it is DISE mode or not and the least significant bit of its program counter. Thus, PEs could select instructions considering the information as well as the internal path register value.

Hybrid Approaches

Hybrid approaches such as StateFull+Partial and StateFull +Partial +DISE can be easily implemented by applying all changes needed for the involved predication techniques. For StateFull+Partial+DISE, a `changepath-_sleep` instruction needs to be added to the ISA additionally, which is discussed in Section 4.4.3.

5.2 Experimental Setup

To evaluate and compare the mentioned predication techniques accurately, we implemented all of them on the baseline architecture at register-transfer level using Verilog HDL. From the RTL description, gate-level circuits were synthesized with 500 MHz of target clock frequency using Synopsys Design Compiler with TSMC 45 nm technology library. We measured the performance and the energy consumption of the reconfigurable array through the gate-level simulation using Synopsys Design Compiler and Mentor Graphics ModelSim. We used CACTI 6.5 [63] to estimate power consumption of the configuration memory also with 500 MHz of target clock frequency at 45 nm technology library.

The experiment is conducted on the reconfigurable array and configuration memory but not on data memory since data memory access is affected minimally in the experiment. Predication could possibly affect data memory access in two ways. According to [50], Partial can cause more memory access since it works in a speculative way compared to full predication. Thus, it can cause notable differences in ILP applications. However, it does not in DLP applications since the amount of accessed data is hardly changed even if multiple paths of control flow are executed. `itpl` is the only exception in the experiment, but it cannot be mapped using Partial anyway. Another factor is increased register pressure. However, we mapped the application to the CGRA using more PEs instead of spilling registers, which is stated in Section 3.1.

We selected ten kernels from real-world applications and faithfully mapped each application to architectures with different predication techniques using libraries [64] and/or manually for comparison. We categorized applications into

SHORT-IF and LONG-IF since the efficiency of predication techniques is much more important for programs having longer *if* structures. We define a *short-if* structure as an *if* structure that has size less than or equal to four according to the experimental results (refer to Section 5.3.2) and SHORT-IF as a set of applications that have only *short-if* structures. According to the classification, the following five examples belong to SHORT-IF.

- *IDCT* (`idct`) performs discrete cosine transform and clips values into the predefined ranges, which is one of the most compute-intensive parts in the JPEG decoder.
- *Chromakey* (`chroma`) is a technique for composition of two images.
- *Finding max* (`max`) searches a given set of integers for the maximum value.
- *Sum of absolute differences* (`sad`) calculates the sum of absolute differences in pairs of integers, which is widely used for video applications.
- *Shift instead of division* (`shift`) divides the given integer by 16 using a shift operation. A control flow is necessary due to the case that the given integer is negative.

On the contrary, the following five examples have much more complex control flows including long and/or *nested-if* structures, which we will refer to as LONG-IF.

- *Rounding* (`round`) approximates the given set of floating point values to the nearest integer. We use half-away-from-zero rounding.

- *SECDED decoding* (`secded`), Single-Error-Correction-and-Double-Error-Detection, is an error-correcting method widely used for communication. We choose Hamming (8,4) among several different ways to perform SECDED decoding.

- *Deblocking filter* (`deblock`) smooths the sharp edges between macroblocks, which arise by the effect of block coding techniques in the H.264 video decoder. The pixels are handled in different ways according to the strength of the blocking effect.

- *Interpolation* (`itpl`) interpolates values between pixels, which is one of the most important steps in the H.264 video decoder. It chooses the mode of interpolation among 16 different modes, and thus forms a long control flow.

- *Efficient pyramid image coder* (`epic`) is the image compression method. Among kernels in the program, *unquantize_image* has a *nested-if* structure.

Many applications are from real multimedia applications; `idct` from JPEG decoder, `sad` from MPEG4, and `deblock` and `itpl` from H.264. In addition, `max`, `shift`, and `round` are commonly used operations. Each kernel consists of one loop and the detailed information for one iteration of the loop is shown in Table 5.1. The execution cycles and the percentage of execution cycles taken by *if* structures are measured based on the case when StateFull is used. Only the outermost *if* structures are considered in the measurement. The average execution cycles of Short-If and Long-If are 45.80 and 172.75 cycles, respectively.

Table 5.1: The Detailed Information of the Applications

	SHORT-IF					LONG-IF				
	`idct`	`chroma`	`max`	`sad`	`shift`	`round`	`secded`	`deblock`	`itpl`	`epic`
Unrolling factor	1	2	1	1	8	4	8	2	4	8
Number of Operations	544	144	328	136	56	128	624	624	3136	176
Data Memory Access (Byte)	32	48	80	32	16	32	24	48	152	24
Execution Cycle	103	25	63	24	14	36	85	105	465	30
If Structures	2.9%	4.0%	7.5%	4.2%	7.1%	19.4%	24.7%	61.0%	50.8%	40.0%
Short *If* Structures	2.9%	2.0%	7.5%	4.2%	7.1%	0%	0%	0%	5.0%	13.3%
Long *If* Structures	0%	0%	0%	0%	0%	19.4%	24.7%	61.0%	45.8%	26.7%

As shown in the table, two groups of applications have different characteristics in terms of the portion of execution time spent for the execution of *if* structures. For *short-if* structures, it is beneficial to simply use PARTIAL (which is also possible for hybrid techniques) or CONDFULL. However, in the case of *long-if* structures, conventional techniques are expected to be inefficient in terms of performance and power consumption. The problem could be even more serious if the *if* structures form nested ones, which is quite common in *long-if* structures. Therefore, we are focusing on programs having long and/or *nested-if* structures (LONG-IF).

5.3 Experimental Results

In this section, we first show that the proposed STATEFULL scheme reduces the power consumption in a PE significantly. Then, we compare STATEFULL scheme and hybrid scheme with conventional approaches to show the improvement in energy consumption as well as performance.

5.3.1 Effect of Predication Mechanism on Power Consumption of a PE

To show the pure effect of different predication mechanisms, we used synthetic applications to reproduce unnecessary paths, which lasted for 10 cycles with a random configuration and input data. If we used real applications, the results could be slightly different but not much because the power consumption on an unnecessary path is mainly due to instruction decoding and the amount of power consumption is about the same regardless of what kind of instructions are decoded.

Figure 5.2 shows power consumption of major components in one PE. It can be seen that, although the predication techniques have no notable difference in static power consumption, they have significant impact on dynamic power consumption. More specifically, StateFull reduced 76.9%, 57.7%, and 65.9% of dynamic power consumption compared to Partial, CondFull, and PseudoBranch, respectively. This is mainly because StateFull does not require decoding of instructions, whereas CondFull and PseudoBranch require at least decoding them and Partial requires even executing them. It enables StateFull to reduce activities in the main components of a PE including the decoder, the ALU, and the register file, which eventually leads to huge savings in dynamic power consumption. Although the counter incurred extra dynamic power consumption to keep track of the sleep period, reduction in dynamic power consumption in the aforementioned components was much larger than the overhead, and thus contributed to reducing the total power consumption (including static power consumption) by 65.4%, 43.4%, and 52.2% compared to Partial, CondFull, and PseudoBranch, respectively.

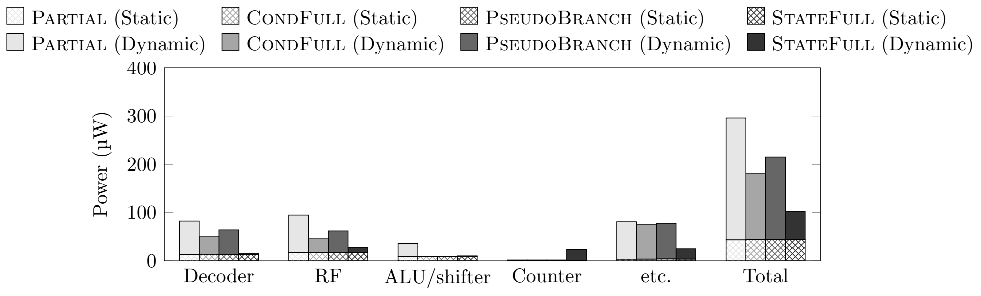


Figure 5.2: Power consumption of a PE on an unnecessary path. Here *etc.* includes power consumption of state registers and wires.

5.3.2 Quantitative Definitions of *short-if* and *long-if*

In order to classify *short-if* and *long-if*, we measured energy on synthetic applications with various sizes of *if* structures under the environment mentioned earlier. We tested two cases; one having addition or shift operations in the body and the other having only `mov` operations in the body. As mentioned in Section 4.4.2, it is because `mov` operations do not use functional units and PARTIAL does not have performance overhead in handling them. The result is shown in Figure 5.3. In `add/shift` cases, CONDFULL and STATEFULL consume almost same energy when the size of body is four. In `mov` cases, the energy consumptions of all three become similar for size five or above. Based on this observation, we define an *if* structure as *short* if its size is less than or equal to four. Otherwise, we regard it as *long*.

5.3.3 Compilation Strategy in STATEFULL+PARTIAL

Figure 5.3 also shows why we use PARTIAL for only special cases where *short-if* structures are only composed of `mov` instructions in the combined approach of STATEFULL+PARTIAL. PARTIAL consumes much more energy than CONDFULL

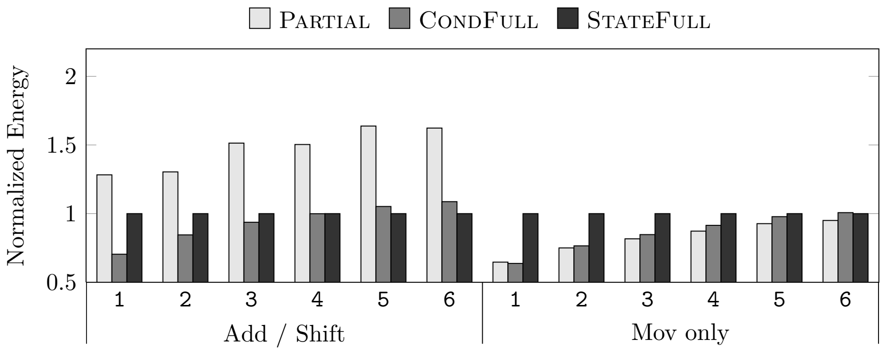


Figure 5.3: Energy normalized to that of StateFull.

or StateFull in `add/shift` cases, but it is quite competitive in `mov` cases. Actually, Partial outperforms StateFull even for long `mov`-only structures, although such cases rarely exist in real applications.

5.3.4 Conventional Techniques (Partial, CondFull, and PseudoBranch) vs. Proposed StateFull Technique

Partial and CondFull are designed to be optimized for simple (non-*nested*) *if* structures, so they will be slow when executing *nested-if* structures. PseudoBranch can support *nested-if* structures, but it requires more instructions to handle control flow than StateFull. Also, StateFull consumes less energy. Note that Partial could not be used for `itpl` since it was composed of 16 control paths, and thus required too many (at least 16) registers to store the outputs of all the paths.

Performance

Figure 5.4 compares the execution time of the four predication techniques, which is normalized to that of StateFull. In the case of Short-If, Partial shows almost the same performance as CondFull since *if* structures in these ap-

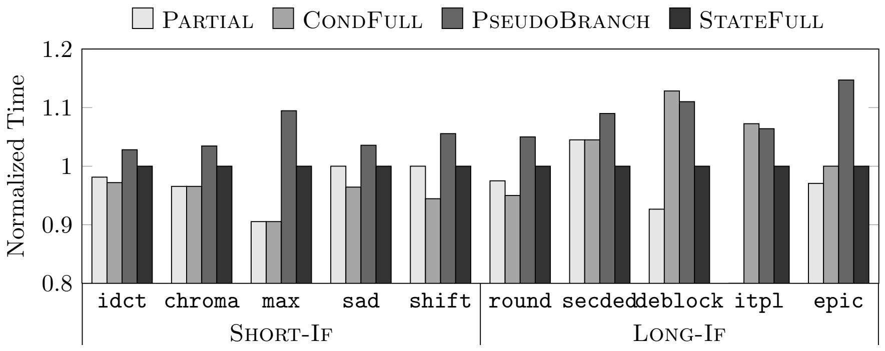


Figure 5.4: Execution time normalized to that of StateFull.

plications have only `mov` instructions in most cases; however, PseudoBranch and StateFull show worse performance than CondFull in every case due to its control overhead incurred by sleep instructions. On the contrary, PseudoBranch and StateFull show their strengths on Long-If since they can avoid the overhead of CondFull due to flattening of *nested-if* structures. Most especially, StateFull mostly outperformed CondFull in terms of performance though CondFull executed `round` faster since the example does not have any *nested-if* structures. One outlier is Partial executing `deblock` much faster than other mechanisms. This is because Partial can extract common subexpressions among different control paths. Considering that Partial executes all possible paths and takes only one result from the selected path, executing the common subexpressions only once for multiple paths helps Partial to improve performance.

Energy Consumption of the Reconfigurable Array

Figure 5.5 shows energy consumption of the reconfigurable array. For Short-If, Partial, and CondFull consumed less energy than PseudoBranch and

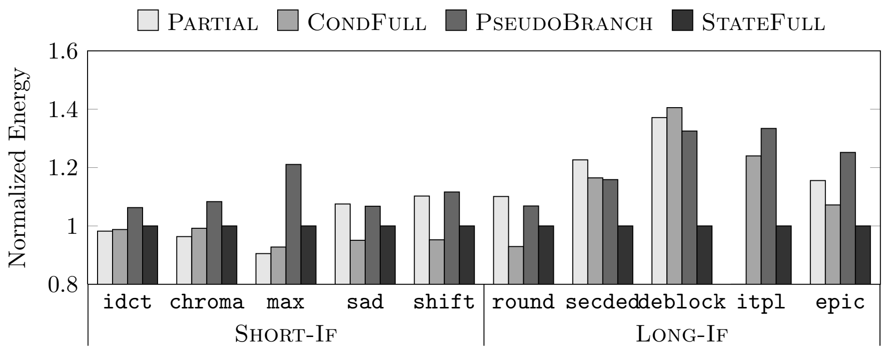


Figure 5.5: Energy consumption of reconfigurable array normalized to that of StateFull.

StateFull mainly due to their faster execution time. Between Partial and CondFull, Partial consumed more energy than CondFull for the last two examples since Partial actually executed arithmetic operations on unnecessary paths, thereby incurring large overhead in energy consumption of the ALU as well as the register file. On the other hand, Partial consumed smaller energy than CondFull for the other examples in Short-If, in which most of the unnecessary paths have only `mov` instructions, because the penalty of executing unnecessary paths was small and Partial did not require any modification to ISA. For Long-If, StateFull consumed 17.3%, 13.1%, and 18.3% lower energy compared to Partial, CondFull, and PseudoBranch, respectively, as it reduced both execution time (Figure 5.4) and power consumption of a PE (Figure 5.2).

Energy Consumption of Configuration Memory

Predication mechanisms also affect energy consumption of the configuration memory, which is shown in Figure 5.6. In most cases, Partial and StateFull consumed less energy on the configuration memory than CondFull though

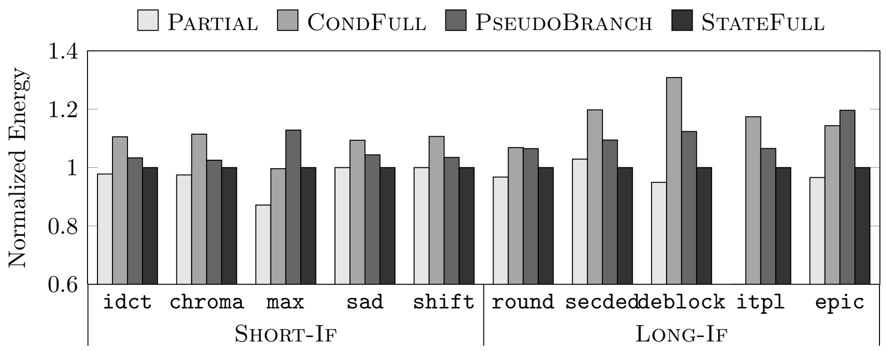


Figure 5.6: Energy consumption of configuration memory normalized to that of StateFull.

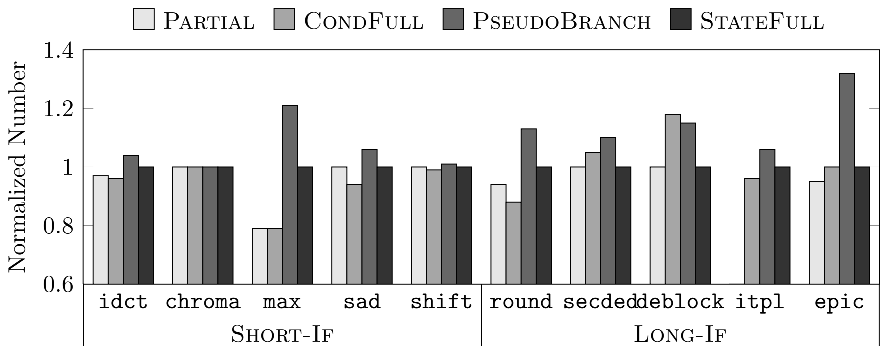


Figure 5.7: Number of fetched instructions normalized to that of StateFull.

CondFull fetched less number of instructions in many cases (see Figure 5.7). This is because CondFull adds a condition field to each instruction, and thus increases both static and dynamic energy consumption of the configuration memory. The only exception is `max` where StateFull suffered from relatively high performance overhead (see Figure 5.4) due to the increase in the number of fetched instructions.

5.3.5 Proposed Hybrid Predication Techniques

We hybridize StateFull with Partial and DISE to improve performance in *short-if* and *if-else* structures, respectively. Figure 5.8a shows the execution time of StateFull and hybrid mechanisms. For applications in Short-If, Partial combined with StateFull contributed to reducing the execution time by up to 9.5% compared to the StateFull-only mechanism. Moreover, DISE further reduced the execution time of applications in Long-If by up to 23.7%, which is mainly due to dual-issuing of instructions. The reduced execution time by Partial directly affected the total energy consumption as depicted in Figure 5.8b since Partial appeared to have negligible overhead of energy consumption caused by extra circuits to support it.

On the other hand, the effect of DISE on energy is varying according to the applications. Basically DISE incurred some degree of extra energy consumption due to the additional logics, but performance improvement reduced the energy consumption in some examples (*round*, *deblock*, and *itpl*). When there is no *if-else* structure like Short-If applications, you can see from Figure 5.8b that energy consumption is increased, which is 2.4% of total energy consumption on the average. Since StateFull and DISE consume almost the same power as DISE also significantly improve the energy consumption compared with the three existing approaches as StateFull does, which is proven in the previous subsections, energy consumption of the three approaches is proportional to the execution time except the logic overhead.

Meanwhile, we can see from Figure 5.8c that DISE does not affect the number of fetched instructions except the case of *round*. As shown in Figure 4.8, since `csleep` instructions are directly replaced by `changepath` instructions and changepath_csleep eliminates the insertion of nop instructions in unbalanced *if-else* structures, no additional instruction overhead is needed in most cases. The only exception is when *if-else* structures are perfectly balanced, which *round* example corresponds to. If the sizes of true-path and false-path are exactly equal, then a nop instruction should be inserted to true-path like Figure 4.4. However, we can obtain huge benefit inversely in such cases since the overhead is just one more fetches of instructions but both paths are overlapped well so the execution time is greatly reduced (see results of *round* in Figure 5.8a and Figure 5.8b).

5.3.6 Putting Together

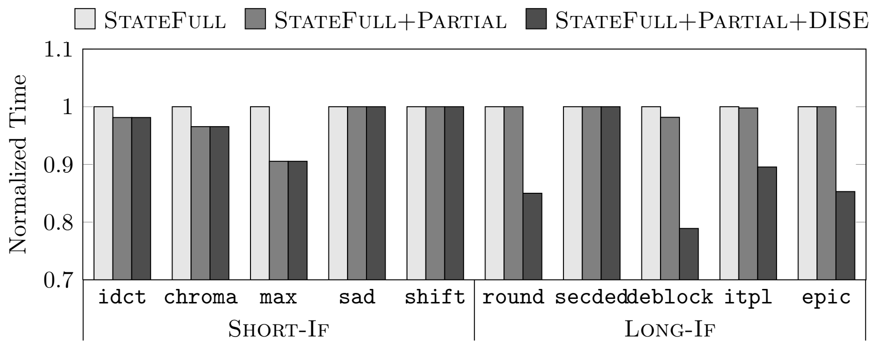
Table 5.2 and Table 5.3 summarize all results discussed so far[5]. We define *improvement* as follows to reflect that less energy/delay/EDP (energy-delay product) is better.

$$\text{Improvement} = 1 - \text{GEOMEAN}\left(\frac{\text{Energy/Delay/EDP of the hybrid mechanism}}{\text{Energy/Delay/EDP of the baseline}}\right)$$

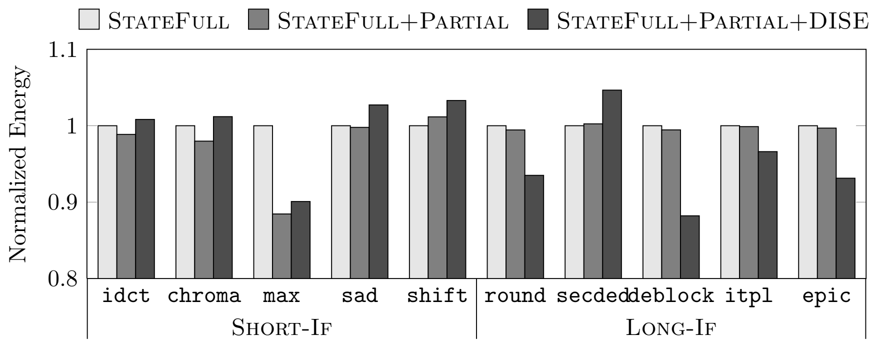
Thus, higher improvement implies better design according to this definition. The first rows in the tables indicate the baseline mechanisms.

According to the experimental results, StateFull+Partial turned out to be suitable for a "universal predication mechanism" that operates efficiently for different kinds of programs, compared to other mechanisms. It consumed less energy than either Partial or StateFull since any of the two mechanisms can be applied according to their specialty. For Long-If, performance was slightly sacrificed compared to Partial to further reduce energy consumption

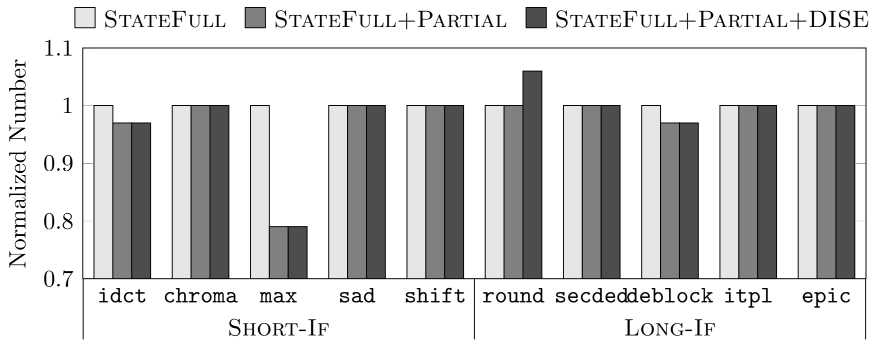
[5]Long-If and Total for Partial do not include the results for itpl since Partial could not be used for it, as explained in Section 5.3.4.



(a) execution time



(b) energy consumption of both reconfigurable array and configuration memory



(c) number of fetched instructions

Figure 5.8: Comparison among StateFull and hybrid approaches. the values are normalized to the results of StateFull.

Table 5.2: Improvements of StateFull+Partial

		Improvement over			
		Partial	CondFull	PseudoBranch	StateFull
Short-If	Energy	1.5%	3.4%	10.6%	2.9%
	Delay	0.0%	-2.1%	7.6%	3.0%
	EDP	1.5%	1.4%	17.4%	5.8%
Long-If	Energy	12.0%	13.8%	15.7%	0.3%
	Delay	-1.7%	4.0%	8.8%	0.4%
	EDP	10.4%	16.1%	20.1%	0.6%
Total	Energy	6.3%	8.8%	13.2%	1.6%
	Delay	-0.8%	1.0%	8.2%	1.7%
	EDP	5.6%	9.0%	18.7%	3.2%

thus to get lower EDP. Also, compared to CondFull, it reduced much energy consumption and EDP in Long-If. Even for Short-If, it reduced energy consumption compared to CondFull despite its slightly worse performance, eventually resulting in better EDP. Lastly, PseudoBranch, another kind of state-based full predication, showed notably worse energy as well as performance than StateFull, which implies that the proposed mechanism is better optimized to both high performance and low power consumption.

Moreover, StateFull+Partial+DISE achieved even better performance with minimal overhead. In particular, it improved performance by 12.2% on average compared to StateFull+Partial in the case of Long-If. In addition, the improved performance contributed to reducing energy consumption thereby making the predication mechanism much more energy efficient, although extra logic circuits of DISE and dual-bank structure of the configuration memory incurred 2.4%[6] overhead in power consumption. As a result, it improved EDPs

[6]This can be inferred from the comparison with StateFull+Partial for Short-If in

Table 5.3: Improvements of StateFull+Partial+DISE

		Improvement over				
		Partial	CondFull	Pseudo Branch	StateFull	StateFull + Partial
Short-If	Energy	-0.9%	1.0%	8.4%	0.5%	-2.4%
	Delay	0.0%	-2.1%	7.6%	3.0%	0.0%
	EDP	-0.9%	-1.0%	15.4%	3.5%	-2.4%
Long-If	Energy	16.4%	17.9%	19.7%	4.9%	4.7%
	Delay	11.1%	15.7%	19.9%	12.5%	12.2%
	EDP	25.7%	27.9%	31.4%	14.7%	14.1%
Total	Energy	7.2%	9.9%	14.2%	2.7%	1.2%
	Delay	5.1%	7.2%	13.9%	7.9%	6.3%
	EDP	11.9%	14.7%	23.8%	9.3%	6.2%

by 25.7%, 27.9%, and 31.4% compared to the conventional mechanisms of Partial, CondFull, and PseudoBranch, respectively. Especially, H.264 applications (`deblock` and `itpl`) are improved by 38.3%, 39.6%, and 32.7%[7]. Considering that there is a trade-off between performance and energy in general, this hybrid approach provides a unique merit that enables an energy-efficient acceleration of control flow execution.

5.3.7 Speedup of Applications

We measured how much the two applications shown in the introduction are accelerated by the proposed hybrid approach (StateFull+Partial+DISE). The execution time is obtained from ARM Developer Suite 1.2 [13] when the applications are running on a single ARM 9 processor. We assume that there is no communication overhead between the processor and the CGRA, which can be eliminated by sharing memory partially [65–68] or entirely [7].

Table 5.3 because DISE is not used for Short-If.

[7]The results are not separately shown in the table, but one can calculate it from the absolute values in the appendix.

Table 5.4: Execution time of JPEG decoder (cycle)

Function Name			ARM9	CGRA w/o Predication Support	CGRA with StateFull+Partial+DISE
Data-Intensive Parts	IDCT	Part1	7,716k	784k	900k
		Part2†	2,335k	2,335k	
		Part3†	4,283k	4,283k	
		Total	14,336k (x1.0)	7,404k (x1.9)	900k (x15.9)
	Dequantization*		3,303k		
Control-Intensive Parts			2,983k		
Entire Application			20,622k (x1.0)	13,692k (x1.5)	7,188k (x2.9)

†a data- and control- intensive part.
*a data-intensive part which has not been mapped yet onto the CGRA.

The results of JPEG decoder are shown in Table 5.4. `Part1` of `idct` is just a data-intensive part so the CGRA can accelerate it as usual. On the other hand, since `Part2` and `Part3` of `idct` have control flows, the ARM 9 processor should execute them if predication is not supported on the CGRA. As a result, without predication, `idct` is accelerated by only 1.9 times while the speedup on the predication-supporting CGRA reaches 15.9 times. Since `idct` takes considerable portion of the entire execution time (69.5%, see Figure 1.1), performance of the entire application is improved by 2.9 times. This result can be improved further if another data-intensive part (`dequantization`) is mapped to the CGRA, but in this thesis we concentrate on both data- and control-intensive parts to see the effect of predications.

Table 5.5 shows the execution time of H.264 decoder. Since most data-intensive parts in H.264 decoder have *if* statements, the CGRA rarely accelerates the application without predication. On the contrary, by using State-

Table 5.5: Execution time of H.264 decoder (cycle)

	Function Name	ARM9	CGRA w/o Predication Support	CGRA with STATEFULL+PARTIAL+DISE
Data-Intensive Parts	MC_part1†	38,273k (x1.0)	38,273k (x1.0)	11,021k (x3.5)
	MC_part2†	13,805k (x1.0)	13,805k (x1.0)	3,065k (x4.5)
	DB_part1†	9,909k (x1.0)	9,909k (x1.0)	1,940k (x5.1)
	DB_part2†	7,981k (x1.0)	7,981k (x1.0)	1,700k (x4.7)
	Integer Transform*	13,449k		
Control-Intensive Parts		126,069k		
Entire Application		209,487k (x1.0)	209,487k (x1.0)	157,245k (x1.3)

†a data- and control- intensive part.
*a data-intensive part which has not been mapped yet onto the CGRA.

FULL+PARTIAL+DISE, one can accelerate four kernels by 3.5 to 5.1 times, which results in 1.3 times improvement over the entire execution time. Similarly to the case of JPEG decoder, this result can also be improved if more parts are mapped onto the CGRA.

Chapter 6

Mapping Framework

6.1 Motivation

There are a number of challenges in compiling loops for CGRAs that support StateFull. At the core of the problem lies operation-to-PE binding, which would be trivial if the target architecture has only one PE like simple SIMD execution. But if the target architecture allows non-SIMD execution, that is, different operations can be performed by different PEs in the same cycle as in a VLIW processor, the ways to map operations to multiple PEs can affect the performance of the CGRA significantly.

First, operations in one conditional need to be scattered among multiple PEs to exploit ILP in the conditional, but the overhead of managing the state registers of several PEs may be large. Thus, to decide how many PEs execute one conditional, we need to consider ILP inside the conditional and the existence of other parts which can run in parallel with it. Second, if operations from several conditionals are interleaved in their schedule on the same PE, switching the state register will cause large overhead. To avoid such overhead, conditionals should not be mapped in the mixed way.

Another issue arises from the fact that power saving mode of a PE renders almost all the resources of a PE including the local register file of a PE inaccessible. If a PE in a power-saving (sleep) state has a variable stored in its register file, and the variable is needed by another PE, we must route the variable in advance before the first PE goes into the power-saving state. Otherwise, routing must be processed after exiting the power-saving mode so performance will be greatly degraded. Even worse, if another routing is required in the opposite direction (i.e., the PE in sleep state wakes up when the data from the other PE is available) at the same time, then the execution can go into deadlock and will be failed.

To handle the above issues, we need to know the size of each conditionals and parallelism among conditionals and let multiple conditionals mapped separately. However, since several conditionals are merged into one DFG by *if*-conversion technique (e.g., [60]) in conventional mapping algorithms using conventional predications, we cannot know the information nor use the algorithms directly. Thus, we propose CDFG representation that expresses conditionals (each will be a DFG) explicitly and reveals parallelism between DFGs. Then, we generate information to map DFGs separately in a temporal or spatial manner. With this information, we can map DFGs not to be mixed each other.

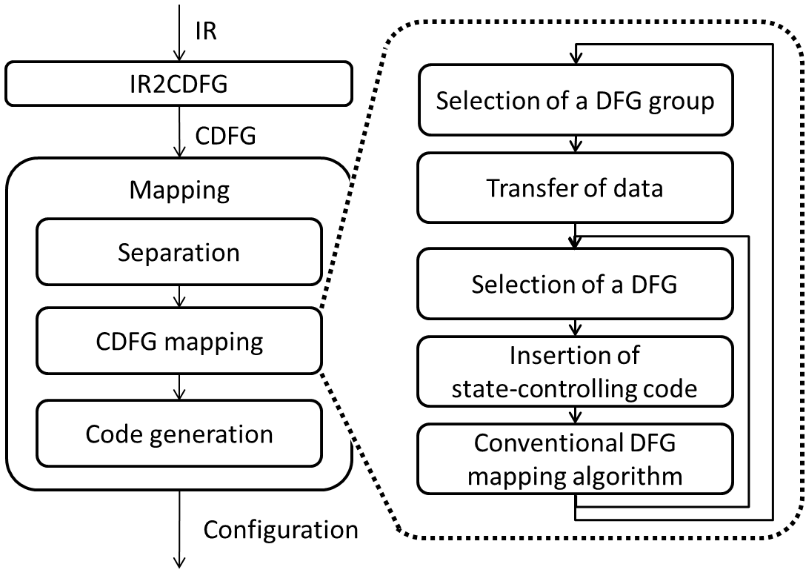


Figure 6.1: The mapping framework on StateFull-based CGRAs.

6.2 Proposed Approach

6.2.1 Overall Flow

Figure 6.1 illustrates the overall flow of the proposed framework for mapping loops with control flow on StateFull-based CGRAs. It starts from IR (intermediate representation), which can be obtained easily by frontend tools. The framework largely consists of two parts. The first part converts IR to CDFG (control data flow graph), extracting parallelism on the way. The second part takes the CDFG and allocates PEs to different parts of the CDFG (each part is a DFG) so that each part can be mapped separately in a temporal or a spatial manner using known mapping algorithms.

6.2.2 From IR to CDFG

CDFG Generation

The IR for a loop body is given as a CFG of DFGs, where each node (DFG) represents a basic block. Figure 6.2a illustrates the CFG of a loop body, which contains one *nested-if* construct followed by a simple *if-else*.

We first transform the CFG to our CDFG representation so that the control structure and parallelism can be captured more explicitly. We use a hierarchical CDFG defined as follows. Each node of the CDFG is a block of either of two types: *unipath* and *multipath*. A unipath block is simply a DFG, whereas a multipath block contains one or more CDFGs with a condition for each CDFG. Figure 6.2b illustrates the CDFG corresponding to the IR in Figure 6.2a. In the figure, ovals, solid round boxes, and dashed ones represent DFGs, blocks, and CDFGs, respectively. Directed edges indicate data dependency between two blocks. Note that the edges in Figure 6.2b are obtained through data flow analysis and are different from those in Figure 6.2a.

To transform an IR to a CDFG representation, we first identify conditionals to generate CDFGs at lower levels of hierarchy (see Figure 6.2b). Since multipath blocks can contain other multipath blocks in such lower level CDFGs, *nested-if* structures can be naturally represented.

Exploiting Parallelism

We then update data dependence among blocks, which may reveal parallelism between blocks. In Figure 6.2b, if DFGs D, $E0$, and $E1$ are not dependent on $B0$, $B1$, or $C0$, we can remove the dependence edge between them, making D an immediate successor of A as illustrated in Figure 6.2c. In addition, we exploit more parallelism by extracting operations from A or F if they have no dependency with any operation in the conditionals. We separate those operations out as new DFGs (G and H) as shown in Figure 6.2c.

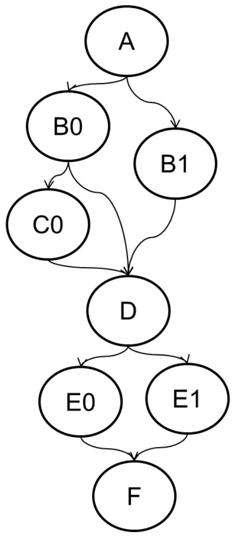
6.2.3 Separation

To ensure correctness and maximize performance, we map operations from different conditionals separately either in temporal or spatial manner, which can be achieved by DFG grouping and PE-to-DFG allocation. A DFG group is defined as a set of DFGs running in parallel. We group DFGs and order groups as a list. Within a group, we allocate different PEs to different DFGs, which corresponds to spatial separation. The groups are put into a list in the order of their generation, which corresponds to temporal separation.

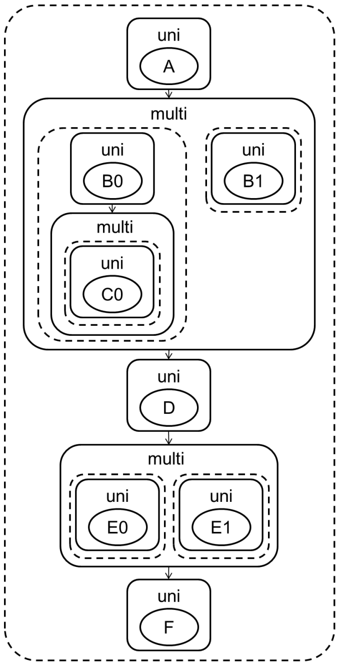
When grouping DFGs, we need to consider two aspects. If many DFGs are put into one group so that they can run in parallel, it can help increase performance especially when each DFG has low ILP. On the other hand, if too many DFGs are in one group, registers can be spilled, resulting in performance degradation. Thus our strategy is to assign enough number of PEs to each DFG to avoid register spills and then to group DFGs as long as PEs are available. Therefore we first estimate the register requirement of each DFG, followed by DFG grouping and PE-to-DFG allocation.

Register Requirement Estimation

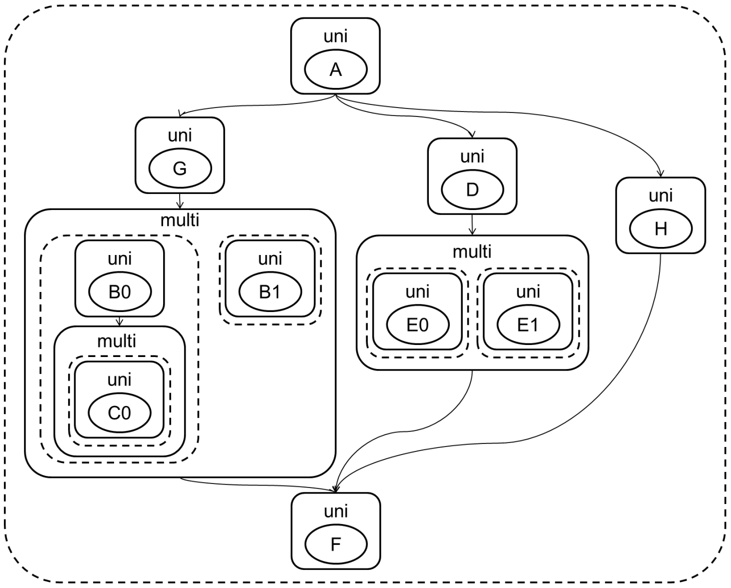
To group as many DFGs as possible, we need to calculate the minimum number of registers that guarantees no registers spill during CDFG mapping. The register requirement of the components of CDFGs can be calculated recursively as follows. The register requirement of a CDFG is the maximum of all the register



(a) A loop body represented as a CFG of DFGs



(b) Identifying conditionals (i.e., fork-join structures)



(c) Exploiting parallelism

Figure 6.2: Conversion process from IR to CDFG.

requirements of its blocks. For a unipath block, its register requirement is that of its DFG. For a multipath block, its register requirement is the maximum of those of its CDFGs. Thus we only need to obtain the register requirement of DFGs.

The register requirement of a DFG is rather complex since it is related to the actual mapping algorithm. Depending on the number of operations mapped together in DFGs and the order of mapping of operations, register requirement can differ. In this thesis, we will use the mapping algorithm where only one operation is considered at a time and the mapping order of operations is decided dynamically. Thus, we calculate the register requirement by assuming the worst case mapping order that uses registers maximally, which will guarantee that registers are not spilled even if operations are mapped in any order.

DFG Grouping and PE-to-DFG Allocation

Based on register requirements, we calculate PE requirements of *ready* DFGs considering available number of registers. If an architecture has four PEs and total eight registers are available now and if a DFG requires three registers, then the PE requirement of the DFG is two since there are two available registers per PE on the average (in the later mapping phase, if the actual number of available registers is different from the requirement, then we may have to take extra cycles to move data around).

After obtaining PE requirements, grouping is performed in a way similar to heaviest-first selection in the knapsack problem. That is, DFGs are selected with most PE requirement first and packed into a sack while the capacity of the sack is not exceeded. If PEs remain but more DFGs cannot be packed due to the capacity violation, spare PEs are distributed to balance the number of operations in DFGs and that of PEs allocated to DFGs. After packing one sack, we calculate the number of registers that will be available after the execution of the previous group and update *ready* DFGs. And then we pack another sack by repeating the above processes starting from calculating PE requirements of *ready* DFGs.

6.2.4 CDFG Mapping

CDFG mapping flow is shown in Figure 6.1 surrounded by the dotted line. Differing from conventional DFG mapping flow, the proposed framework requires two more simple processes. First, after selecting a DFG group, we route their input data that are not in the PEs assigned to their DFGs. Then, we move data irrelevant to each DFG to PEs that do not belong to the DFG if the number of available registers in the allocated PEs is not sufficient compared to its register requirement. Note that the total number of available registers in all PEs is enough since we group DFGs not to violate it. The second process is that state-controlling operations (sleep instructions) are inserted at the entry of the DFGs for conditionally executed DFGs.

After the two processes, mapping a DFG to a set of PEs can be done using known mapping algorithms (e.g., [38, 40, 43, 69]). Since the framework solves the problem of handling CDFG by separating control flow and data flow, control flow needs not be considered during data flow mapping, thus conventional algorithms can be easily integrated into our framework. However, note that according to DFG mapping algorithm, we need to modify the algorithm to estimate register requirement of DFGs presented in Section 6.2.3.

6.3 Implementation

We extended the LLVM compiler infrastructure [70] to implement the proposed mapping framework. Clang compiler [71] was used as the frontend tool to obtain IR. For the DFG mapping we used a variant of list-scheduling-based mapping algorithm that performs scheduling, operation-to-PE binding, and register binding all at once, similarly to [38] except that we do not adopt modulo scheduling and the mapping order of operations is decided dynamically. However, other mapping algorithms can also be used as mentioned in Section 6.2.4.

6.4 Experiments

6.4.1 Experimental Setup

Any kernel having control flow can be a target application. Especially, parallel conditionals often appear as a result of loop unrolling, although they can also appear within a single iteration. Thus, we experimented with the following applications with various unrolling factors (1,2,4, and 8).

- Clipping (`clip`): it saturates values into the predefined ranges.
- Sum of absolute differences (`sad`): it calculates the sum of absolute differences between pairs of integers.
- Shift instead of division (`shift`): it divides the given integers by 16 using shift operations. If the integer is negative, then control flow is needed.
- SECDED decoding (`secded`): it means single-error-correction-and-double-error-detection.

For the architecture, we use the one verified in the previous chapter.

6.4.2 Verification of Mapping Framework

We verified the functional correctness of the proposed mapping framework by simulating mapping results obtained from the framework on FloRA at RTL using ModelSim. We tested with total 16 cases (4 examples with 4 unrolling factors for each), and confirmed that the proposed framework works correctly in all cases.

6.4.3 Quality of Mapping Results

We need to check the framework to see if it really exploits parallelism among multiple conditionals well. It is hard to measure the relative quality of mapping results since this is the first work for compiling applications using StateFull, but one way is to compare with a naïve approach where multiple conditionals are handled sequentially. The comparison results are shown in Figure 6.3. We assume that there are total 64 iterations. 8-way SIMD is supported in the architecture.

In the figure, all the examples show a similar tendency, but the results of 'sad' show what we want to do in this thesis. Each iteration in 'sad' has low ILP so there exist many idle PEs if it executes only one iteration in a column (8 PEs). Thus, we unroll the loop to increase the utilization. However, if structures are serialized in the naïve approach, thus unrolling does not give enough benefit. On the other hand, the proposed method fully parallelizes eight *if* structures in one column, maximizing benefit from unrolling. The average improvement of our approach for the cases with unroll factor of 8 is 2.21 in the harmonic mean. Note that our work is irrelevant to how much benefit loop unrolling gives, but tries to maximize the performance given that a loop is unrolled.

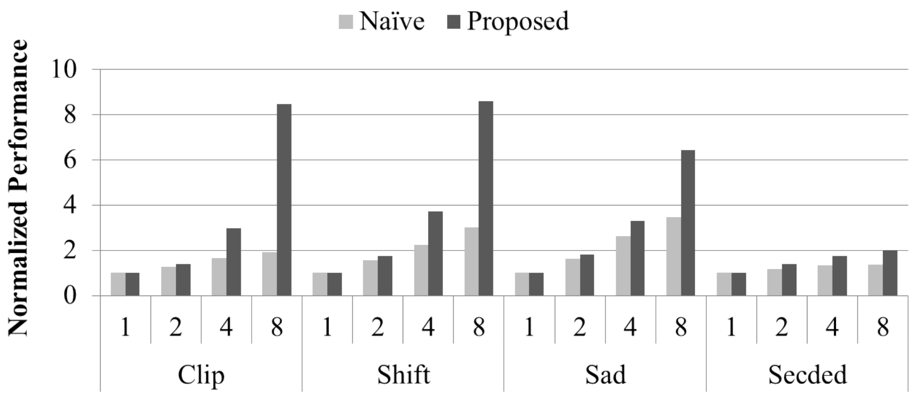


Figure 6.3: Comparison of mapping results on performance. The upper X axis means the unrolling factors. The values are normalized to baselines. A baseline of each example is the case when unrolling factor is 1 and the naïve approach is used.

Chapter 7

Conclusion

7.1 Summary

This thesis has presented a comprehensive analysis of predicated execution techniques in terms of performance and power consumption. Although predication is imperatively necessary to leverage on data-level parallelism, little has been known about its impact on performance, and more importantly, power consumption. We have found that conventional predication techniques have deficiency in power consumption due to the instruction decoding (and execution in some cases) over unnecessary paths which do not need to be executed at all.

Based on this analysis, this thesis has proposed power-efficient and high-performance mechanisms for predicated execution. In particular, state-based full predication (StateFull) has been developed to eliminate the need for decoding, executing, and committing instructions on unnecessary paths as well as to fully support *nested-if* structures in an efficient way. Moreover, Dual-

Issue-Single-Execution (DISE) has been proposed to accelerate the execution of *if-else* structures. On top of that, hybrid mechanisms have been developed to further improve the proposed techniques in terms of both performance and energy consumption.

Experimental results obtained by gate-level simulation of RTL design have shown that the proposed techniques successfully improved both performance and power consumption at the same time. More specifically, compared to the conventional mechanisms, the hybrid mechanism combining StateFull, partial predication (Partial), and DISE improved energy-delay product by 11.9% on average over all applications used for the experiments compared to Partial, 14.7% compared to condition-based full predication (CondFull), and 23.8% compared to another state-based approach (PseudoBranch). Especially on H.264 applications (`deblock` and `itpl`), energy-delay product is improved by 38.3%, 39.6%, and 32.7% compared to Partial, CondFull, and PseudoBranch, respectively. We believe that the proposed mechanisms could be competitive candidates for a "universal predication technique" that makes better use of data-level parallelism in CGRAs.

This thesis also has presented a mapping framework for CGRAs that relies on StateFull for conditional execution. While StateFull can remove wasteful power consumption by introducing the sleep state into the processing elements, mapping becomes more challenging when handling multiple conditionals. The proposed framework uses a new CDFG structure that can succinctly capture the parallelism existing between conditionals and makes it possible to be integrated with conventional mapping algorithms by separating the handling of control flow and data flow. Experimental results demonstrate that our framework successfully finds and exploits parallelism between multiple conditionals, thereby leading to 2.21 times higher performance than the naïve approach.

7.2 Applicable Scope and Future Work

The contribution of this thesis can be divided into two categories, predication techniques and mapping framework. Since predication is a PE-based approach, the proposed predications can be used for any DLP processors including SIMD units and CGRAs, where branch is not possible and the use of predication is mandatory to handle control flow. On the other hand, the mapping problems occur only when one iteration of a loop is mapped onto several PEs. It is natural to other CGRAs but not to SIMD units. It means that the proposed mapping framework is adaptable only to CGRAs. However, as [35] recently revealed, conventional SIMD can be inefficient and it is needed to exploit ILP partially in some applications, and thus it is expected that our framework will be used for SIMD units in the future.

There remains some work not fully covered in this thesis. First, the optimality of the separation algorithm is not fully addressed. As a future research, we may be able to analyze the optimality and refine the algorithms for better results. Second, in JPEG decoder and H.264 decoder, although we cover only some parts related to our topic in order to focus on the effect of predications, accelerating full applications can also be valuable work to prove the competitiveness of CGRAs. Lastly, since predication is a general approach as stated in the previous paragraph, we can adopt the proposed ones into general SIMD units or GPUs, which makes our work more influential.

Appendix

Execution Time (μs)

Application	Partial	CondFull	PseudoBranch	StateFull	StateFull + Partial	StateFull + Partial +DISE
dct	0.210	0.208	0.220	0.214	0.210	0.210
chroma	0.056	0.056	0.060	0.058	0.056	0.056
max	0.134	0.134	0.162	0.148	0.134	0.134
sad	0.056	0.054	0.058	0.056	0.056	0.056
shift	0.036	0.034	0.038	0.036	0.036	0.036
round	0.078	0.076	0.084	0.080	0.080	0.068
secded	0.186	0.186	0.194	0.178	0.178	0.178
deblock	0.202	0.246	0.242	0.218	0.214	0.172
itpl	-	1.006	0.998	0.938	0.936	0.84
epic	0.066	0.068	0.078	0.068	0.068	0.058

Dynamic Energy Consumption on Reconfigurable Array (pJ)

Application	Partial	CondFull	PseudoBranch	StateFull	StateFull + Partial	StateFull + Partial +DISE
dct	4206	4232	4566	4269	4246	4408
chroma	914	944	1035	946	932	995
max	1594	1638	2165	1747	1556	1624
sad	630	541	616	571	568	608
shift	528	446	528	466	480	507
round	1332	1090	1263	1176	1166	1158
secded	3485	3273	3226	2746	2760	3040
deblock	3790	3754	3505	2527	2551	2424
itpl	-	16488	18012	12852	12867	13151
epic	1268	1152	1350	1058	1053	1037

Static Energy Consumption on Reconfigurable Array (pJ)

Application	Partial	CondFull	PseudoBranch	StateFull	StateFull + Partial	StateFull + Partial +DISE
dct	663	665	704	689	674	699
chroma	177	179	192	187	180	186
max	378	383	472	431	385	398
sad	178	173	186	181	181	187
shift	113	108	120	115	115	119
round	246	242	269	258	257	226
secded	588	595	623	575	573	595
deblock	640	787	777	704	691	573
itpl	-	3239	3214	3058	3033	2822
epic	209	218	250	220	219	193

Number of Fetched Instructions

Application	Partial	CondFull	PseudoBranch	StateFull	StateFull + Partial	StateFull + Partial +DISE
dct	544	536	584	560	544	544
chroma	136	136	136	136	136	136
max	264	264	408	336	264	264
sad	144	136	152	144	144	144
shift	568	560	576	568	568	568
round	120	112	144	128	128	136
secded	624	656	688	624	624	624
deblock	624	736	720	624	608	608
itpl	-	4024	4448	4208	4200	4216
epic	168	176	232	176	176	176

Dynamic Energy Consumption on Configuration Memory (pJ)

Application	Partial	CondFull	PseudoBranch	StateFull	StateFull + Partial	StateFull + Partial +DISE
dct	458	517	495	473	458	403
chroma	126	145	126	126	126	108
max	222	254	355	288	222	195
sad	110	118	118	110	110	100
shift	341	383	348	341	341	334
round	111	119	134	119	119	108
secded	579	698	638	579	579	496
deblock	579	783	668	579	564	484
itpl	-	3565	3499	3276	3269	2954
epic	156	187	215	163	163	140

Static Energy Consumption on Configuration Memory (pJ)

Application	Partial	CondFull	PseudoBranch	StateFull	StateFull + Partial	StateFull + Partial +DISE
dct	1246	1409	1305	1270	1246	1261
chroma	332	379	356	344	332	336
max	795	908	961	878	795	805
sad	332	366	344	332	332	336
shift	214	230	225	214	214	216
round	463	515	498	475	475	408
secded	1104	1260	1151	1056	1056	1069
deblock	1199	1667	1436	1294	1270	1033
itpl	-	6817	5922	5566	5554	5044
epic	392	461	463	404	404	348

Bibliography

[1] D. Chen and J. Rabaey, “A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high speed DSP data paths,” *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 1895–1904, Dec. 1992.

[2] C. Ebeling, D. C. Cronquist, and P. Franklin, “Configurable computing: the catalyst for high-performance architectures,” in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 1997.

[3] T. Miyamori and K. Olukotun, “REMARC: reconfigurable multimedia array coprocessor,” *IEICE Transactions on Information and Systems*, pp. 389–397, 1998.

[4] S. C. Goldstein, H. Schmit, M. Moe, M. Budiuy, S. Cadambi, R. R. Taylor, and R. Laufer, “PipeRench: a coprocessor for streaming multimedia acceleration,” in *Proceedings of the International Symposium on Computer Architecture*, 1999.

[5] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and M. C. Filho, “MorphoSys: an integrated reconfigurable system for data-parallel

and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, pp. 465–481, May 2000.

[6] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP–A self-reconfigurable data processing architecture," *Journal of Supercomputing*, vol. 26, pp. 167–184, Sep. 2003.

[7] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the International Conference on Field Programmable Logic and Application*, 2003.

[8] F. Garzia, W. Hussain, and J. Nurmi, "CREMA: a coarse-grain reconfigurable array with mapping adaptiveness," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2009.

[9] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2005.

[10] Y. Saito, T. Sano, M. Kato, V. Tunbunheng, Y. Yasuda, M. Kimura, and H. Amano, "MuCCRA-3: a low power dynamically reconfigurable processor array," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2010.

[11] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2001.

[12] K. Choi, "Coarse-grained reconfigurable array: architecture and application mapping," *IPSJ Transactions on System LSI Design Methodology*, vol. 4, pp. 31–46, Feb. 2011.

[13] "ARM development suite," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.ads/index.html.

[14] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983.

[15] P. Dang, "An efficient implementation of in-loop deblocking filters for H.264 using VLIW architecture and predication," in *International Conference on Consumer Electronics Digest of Technical Papers*, 2005.

[16] C. Arbelo, A. Kanstein, S. Lopez, J. Lopez, M. Berekovic, R. Sarmiento, and J. Y. Mignolet, "Mapping control-intensive video kernels onto a coarse-grain reconfigurable architecture: the H.264/AVC deblocking filter," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2007.

[17] W. Chuang, B. Calder, and J. Ferrante, "Phi-predication for light-weight if-conversion," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

[18] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[19] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," in *Proceedings on the International Workshop on Field Programmable Logic*, 1998.

[20] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a Raw machine," in *Proceedings on the international Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[21] K. Wu, A. Kanstein, J. Madsen, and M. Berekovic, "MT-ADRES: multithreading on coarse-grained reconfigurable architecture," in *Proceedings of the International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, 2007.

[22] T. V. Aa, M. Palkovic, M. Hartmann, P. Raghavan, A. Dejonghe, and L. V. der Perre, "A multi-threaded coarse-grained array processor for wireless baseband," in *Proceedings of the IEEE Symposium on Application Specific Processors*, 2011.

[23] H. Singh, G. Lu, F. Kurdahi, N. Bagherzadeh, E. Filho, and R. Maestre, "MorphoSys: case study of a reconfigurable computing system targeting multimedia applications," in *Proceedings of the Design Automation Conference*, 2000.

[24] F. Veredas, M. Scheppler, W. Moffat, and B. Mei, "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia

purposes," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2005.

[25] S. Purohit, S. R. Chalamalasetti, M. Margala, and W. Vanderbauwhede, "Throughput/resource-efficient reconfigurable processor for multimedia applications," *IEEE Transactions on Very Large Scale Integration Systems*, 2012, in press.

[26] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture*, 2009.

[27] M. Jo, G. Lee, K. Chang, K. Han, K. Choi, H. Yang, and K. Yoon, "Coarse-grained reconfigurable architecture for multiple application domains: a case study," in *Proceedings of the International Conference on Hybrid Information Technology*, 2009.

[28] C. Brunelli, F. Garzia, D. Rossi, and J. Nurmi, "A coarse-grain reconfigurable architecture for multimedia applications supporting subword and floating-point calculations," vol. 56, pp. 38–47, Jan. 2010.

[29] D. Novo, W. Moffat, V. Derudder, and B. Bougard, "Mapping a multiple antenna SDM-OFDM receiver on the ADRES coarse-grained reconfigurable processor," in *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation*, 2005.

[30] H. Parizi, A. Niktash, A. Kamalizad, and N. Bagherzadeh, "A reconfigurable architecture for wireless communication systems," in *Processing*

on the International Conference on Information Technology: New Generations, 2006.

[31] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid, "FLEXDET: flexible, efficient multi-mode MIMO detection using reconfigurable ASIP," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2012.

[32] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proceedings of the International Symposium on Computer Architecture*, 1983.

[33] http://www.intel.com.

[34] "The ARM® NEON™ general-purpose SIMD engine," http://www.arm.com/products/processors/technologies/neon.php.

[35] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: tailoring SIMD execution using heterogeneous hardware and dynamic configurability," in *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture*, 2012.

[36] http://www.xilinx.com/.

[37] http://www.altera.com/.

[38] B. Mei, S. Vemaldet, D. Verkestt, H. D. Man, and R. Lauwereins, "DRESC: a retargetable compiler for coarse-grained reconfigurable architectures," in

Proceedings of the International Conference on Field-Programmable Technology, 2002.

[39] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[40] T. Toi, N. Nakamura, Y. Kato, T. Awashima, and K. Wakabayashi, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *Proceedings of the International Conference on Computer-Aided Design*, 2008.

[41] G. Lee, S. Lee, and K. Choi, "Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques," in *Proceedings of the International SoC Design Conference*, 2008.

[42] B. D. Sutter, P. Coene, T. V. Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, 2008.

[43] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek, "A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 17, pp. 1565–1578, Nov. 2009.

[44] S. Friedman, A. Carroll, B. Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2009.

[45] G. Lee, S. Lee, K. Choi, and N. Dutt, "Routing-aware application mapping considering steiner points for coarse-grained reconfigurable architecture," in *Proceedings of the International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, 2010.

[46] Y. Kim, J. Lee, and A. Shrivastava, "Operation and data mapping for CGRAs with multi-bank memory," in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2012.

[47] G. Lee, K. Choi, and N. Dutt, "Mapping multi-domain applications onto coarse-grained reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 637–650, May 2011.

[48] D. Lee, M. Jo, K. Han, and K. Choi, "FloRA: coarse-grained reconfigurable architecture with floating-point operation capability," in *Proceedings of the International Conference on Field-Programmable Technology*, 2009.

[49] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2006.

[50] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the International Symposium on Computer Architecture*, 1995.

[51] M. L. Anido, A. Paar, and N. Bagherzadeh, "Improving the operation autonomy of SIMD processing elements by using guarded instructions and pseudo branches," in *Proceedings of the Euromicro Symposium on Digital System Design*, 2002.

[52] L. Huang, L. Shen, S. Ma, N. Xiao, and Z. Wang, "DM-SIMD: a new SIMD predication mechanism for exploiting superword level parallelism," in *Proceedings of the International Conference on ASIC*, 2009.

[53] K. Han, J. K. Paek, and K. Choi, "Acceleration of control flow on CGRA using advanced predicated execution," in *Proceedings of the International Conference on Field-Programmable Technology*, 2010.

[54] K. Han, S. Park, and K. Choi, "State-based full predication for low power coarse-grained reconfigurable architecture," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2012.

[55] Y. Choi, A. Knies, L. Gerke, and T. Ngai, "The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor," in *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture*, 2001.

[56] E. Quinones, J. M. Parcerisa, and A. Gonzalez, "Improving branch prediction and predicated execution in out-of-order processors," in *Proceedings*

of the International Symposium on High Performance Computer Architecture, 2007.

[57] J. Lee, Y. Kim, J. Jung, S. Kand, and K. Choi, "Reconfigurable ALU array architecture with conditional execution," in *Proceedings on the international SoC Design Conference*, 2004.

[58] K. Chang and K. Choi, "Mapping control intensive kernels onto coarse-grained reconfigurable array architecture," in *Proceedings of the International SoC Design Conference*, 2008.

[59] G. Lee, K. Chang, and K. Choi, "Automatic mapping of control-intensive kernels onto coarse-grained reconfigurable array architecture with speculative execution," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, 2010.

[60] S. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture*, 1992.

[61] G. Dasika, M. Woh, S. Seo, N. Clark, T. Mudge, and S. Mahlke, "Mighty-morphing power-SIMD," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2010.

[62] A. Fijany and F. Hosseini, "Image processing applications on a low power highly parallel SIMD architecture," in *Proceedings of the IEEE Aerospace Conference*, 2011.

[63] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: a tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[64] K. Han and K. Choi, "Library-based mapping of application to reconfigurable array architecture," *Jounal of Semiconductor Technology and Science*, vol. 9, pp. 209–215, Dec. 2009.

[65] K. Chang and K. Choi, "Memory-centric communication architecture for reconfigurable computing," in *Proceedings of the International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, 2010.

[66] J. Paek, J. Lee, and K. Choi, "CRM: configurable range memory for fast reconfigurable computing," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum*, 2010.

[67] Y. Kim, K. Han, and K. Choi, "A host-accelerator communication architecture design for efficient binary acceleration," in *Proceedings of the International SoC Design Conference*, 2011.

[68] T. X. Mai and J. Lee, "Software-managed automatic data sharing for coarse-grained reconfigurable coprocessors," in *Proceedings of the International Conference on Field-Programmable Technology*, 2012.

[69] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "EPIMap: using epimorphism to map applications on CGRAs," in *Proceedings of the Design Automation Conference*, 2012.

[70] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[71] http://clang.llvm.org/.

국문초록

재구성형 구조는 연산량이 많은 프로그램을 내장형 시스템에서 가속시키는데 적합한 방법 중 하나이다. 이는 일반적으로 많은 연산유닛들과 하나의 컨트롤러로 구성되어 고성능, 유연성, 저전력을 동시에 달성할 수 있도록 해준다. 많은 연산유닛을 바탕으로 한 병렬처리는 응용프로그램의 실행속도를 빠르게 하며, 재구성 기능은 다양한 응용프로그램에의 활용을 가능하게 해준다. 또한, 명령어와 데이터에 대한 스케쥴을 미리 정해놓음으로써 제어구조를 단순화시킬 수 있으며 이는 연산량 대비 전력소모를 최소한으로 줄여준다.

하지만 응용프로그램이 복잡해짐에 따라 연산량이 많은 부분들에 분기문이 생기게 되었으며 이는 재구성형 구조를 사용함에 있어 큰 위협이 되고 있다. 분기문을 다룰 수 있는 컨트롤러가 하나이기 때문에 컨트롤러에 병목현상이 발생하거나 동시에 서로 다른 제어를 요구하게 되면 해당 프로그램은 가속이 불가능해진다. 조건실행이라는 기술을 사용할 경우 이를 부분적으로 해소할 수 있지만 기존에 개발되어 있는 조건실행 기술들은 재구성형 구조에 성능 및 전력소모 면에서 부정적인 영향을 끼친다.

따라서 본 논문에서는 연산량이 많지만 분기문을 가진 응용프로그램에서 조건실행이 성능과 전력 면에서 어떠한 영향을 미치는지 밝히며 이를 바탕으로 고성능과 저전력을 가진 조건실행 방법을 제안한다. 실험 결과에 따르면 제안한 방식은 기존의 세가지 방식보다 성능과 전력소모를 곱으로 표현한 수치에 있어서 11.9%, 14.7%, 23.8% 만큼의 이득을 보였다. 또한, 제안한 조건실행 방법에 적합한 컴파일 체계도 제안하였다. 제안한 조건

실행은 절전모드를 사용함에 따라 전력을 아낄 수 있지만 기존의 컴파일 방식으로는 여러 조건문을 병렬적으로 수행하도록 컴파일할 수 없는 문제가 생긴다. 따라서 본 논문에서는 이런 문제를 밝히고 조건문들을 서로 다른 연산유닛에 할당함으로써 문제를 해결하는 방식을 제안하고 있다. 제안한 방식을 사용할 경우 단순하고 직관적인 방법에 비하여 평균적으로 2.21배의 높은 성능을 얻을 수 있었다.

주요어: 재구성형 구조, 재구성형, 조건실행, 저전력, 고성능

학번: 2008-21002

감사의 글

학위 논문을 완성하기까지 그리고 박사학위를 받기까지 부족한 저에게 도움을 주신 많은 분들께 감사의 마음을 표현하고자 합니다.

우선 누구보다도 부모님께 감사의 인사를 드리고 싶습니다. 제가 연구에 전념할 수 있도록 물심양면으로 지원을 아끼시지 않으신 부모님이 계셨기에 지금의 결과를 얻을 수 있었다고 생각합니다. 비단 대학원 생활에서 뿐만이 아니라 지금까지 살면서 맞닥뜨린 여러 어려운 순간들에도 저를 믿고 지원해주신 점, 감사드립니다.

학문적으로 저를 이끌어주신 최기영 교수님께 깊은 감사를 드립니다. 자유롭게 생각을 펼칠 수 있도록 도와주시고 그러한 생각이 발전적인 방향으로 나아가도록 지도해주신 덕에 좋은 논문들을 쓸 수 있었습니다. 학문에 대한 진실됨과 연구에 대한 끊임없는 열의는 학자로서의 본보기가 되어주셨으며 학생들을 인격적으로 존중해주시는 인자하신 성품은 인간적인 면에 있어서도 큰 귀감이 되었습니다.

같은 연구실 선후배, 동기에게도 감사한 마음을 전합니다. 학부생 때 졸업프로젝트를 맡아준 석형이형과 강희형, 같은 팀으로 연구 주제의 기반을 닦을 수 있도록 도와준 만휘형과 경욱이, 칩 테스트를 같이 진행한 동욱이형, 연구를 도와준 종경이, 성식이, 양수, 준환이, 그리고 언제든 아이디어 논의 대상이 되어준 준희형과 진호에게 감사의 마음을 전합니다. 이외에도 지식적으로 도움을 주시고 부족한 제 성격을 받아준 연구실 선배들인 기성이형, 임용이형, 현직이형, 한민이형에게도 감사합니다. 영철이형, 동엽이형, 혁중이형, 영배형, 우디, 밍양, 재훈이형, 경훈이형, 학림이, 동우,

재민이, 선욱이, 성주, 남형이, 지현이도 연구실 생활을 함께 할 수 있어서 즐거웠습니다.

바쁘신 와중에도 논문지도를 기꺼이 허락해주신 채수익 교수님, 백윤흥 교수님, 이종은 교수님, 김윤진 교수님께도 감사의 말씀을 드리며, 오랜 시간을 함께한 중학교 친구들, 대학교 생활을 즐겁게 할 수 있도록 해준 컴반04학번들, 긴 시간을 함께하지 못 하였지만 어려울 때 힘이 되어준 시그마 동아리 사람들에게도 감사합니다. 형으로써 많은 신경을 써주진 못한 동생 규문이에게는 미안한 마음을 전하며, 곁에서 힘이 되어준 여자친구에게도 감사의 마음을 전합니다.

저를 도와주신 모든 분들도 원하시는 바를 이루시기 바라며, 끝은 새로운 시작이라는 마음으로 앞으로도 열심히 살도록 하겠습니다.