Power-Efficient Predication Techniques for Acceleration of Control Flow Execution on CGRA

KYUSEUNG HAN, JUNWHAN AHN, and KIYOUNG CHOI, Seoul National University

Coarse-grained reconfigurable architecture typically has an array of processing elements which are controlled by a centralized unit. This makes it difficult to execute programs having control divergence among PEs without predication. However, 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. This article reveals performance and power issues in predicated execution which have not been well-addressed yet. Furthermore, it proposes fast and power-efficient predication mechanisms. Experiments conducted through gate-level simulation show that our mechanism improves energy-delay product by 11.9% to 23.8% on average.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: CGRA, predication, predicated, low power, architecture, reconfigurable, control flow

ACM Reference Format:

Han, K., Ahn, J., and Choi, K. 2013. Power-efficient predication techniques for acceleration of control flow execution on CGRA. *ACM Trans. Archit. Code Optim.* 10, 2, Article 8 (May 2013), 25 pages.

DOI: <http://dx.doi.org/10.1145/2459316.2459319>

1. INTRODUCTION

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 both performance and flexibility. It can achieve high performance through parallel array processing and execute various applications by reconfiguring them at runtime. However, relatively high power consumption compared to ASIC is one of hurdles for CGRA to be integrated into embedded systems, so reducing power consumption is one of the most important challenges that CGRA is facing.

The power problem could be exacerbated when handling control flows. Due to the architectural limitation, CGRA can execute control flows only if they are transformed into data flows by predicated execution techniques. However, conventional predication

This work is an extension of a conference paper previously presented in FPT 2010 [Han et al. 2010] and in DATE 2012 [Han et al. 2012].

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Korea government (MEST) under grant no. 2012-0006272 and in part by Ministry of Knowledge Economy and IDEC Platform center at Hanyang University.

Authors' address: K. Han, J. Ahn, and K. Choi (corresponding author), Design Automation Lab, Department of Electrical Engineering and Computer Science, Seoul National University, Seoul, Republic of Korea; email: kchoi@snu.ac.kr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/05-ART8 \$15.00

DOI: <http://dx.doi.org/10.1145/2459316.2459319>

can threaten the competitiveness of CGRA since it causes serious overhead in both performance and power consumption. Nonetheless, to the best of our knowledge, these aspects have not been well-addressed yet.

This article reveals performance and power issues of predicated execution and proposes a novel mechanism to overcome drawbacks of the conventional techniques. The main contributions of the article include the following.

- We investigate power consumption related with predicated execution techniques for the first time not only in the domain of CGRA, 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 [Arbelo et al. 2007; Dang 2005] and/or compiler-level [Chuang et al. 2003; Shin et al. 2005] 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 13.8% on the average over the main target applications.
- 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 had 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 12.2% on average.
- We compare several predication techniques in terms of both performance and power consumption at gate level. The previous research done by Mahlke et al. [1995] was limited to comparing only performance of two techniques. We implemented several predication techniques at register-transfer level and measured performance and power consumption through gate-level simulation after logic synthesis.

This article is an extension of our previous work presented in Han et al. [2010, 2012]. The extension includes hybrid architectures combining DISE and partial predication with state-based full predication and an in-depth comparison among various predication schemes with a wide variety of examples.

2. BACKGROUND

2.1. Coarse-Grained Reconfigurable Architecture

Coarse-Grained Reconfigurable Architecture (CGRA) is generally composed of tens of PEs, which are tightly connected with each other in the form of a 1D or 2D array. PEs are equipped with several functional units such as ALUs, shifters, and/or multipliers, and control signals reconfigure them at runtime to change their functionality. Although the concept of reconfiguration is similar to FPGA, its configuration has word-level granularity rather than bit-level ones as in FPGA, thereby reducing the amount of configuration information and thus the time overhead of reconfiguration. Due to these benefits, there has been much research proposing various kinds of CGRAs [Becker and Vorbach 2003; Choi 2011; Garzia et al. 2009; Hartenstein 2001; Lee et al. 2009; Mei et al. 2004; Saito et al. 2010].

However, CGRAs have a limitation in handling control flows. Multiple PEs in a CGRA are typically controlled by a single, centralized control unit, and thus each PE cannot select its own instruction flow. In other words, the entire PEs should take the

same branch (either if- or else-path) at each conditional branching point. This limitation prevents programs requiring different control for each PE from being parallelized. 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;
}
```

To exploit data-level parallelism, each loop iteration is mapped to a different set of PEs so that overall performance can be improved. However, a single control unit of the entire CGRA cannot handle the case where some PEs should execute if-path ($x[i] = 0$) and some others should do else-path ($x[i] = 1$) at the same time. This problem, called *control divergence*, prevents such programs from being accelerated by CGRA through utilizing data-level parallelism. This problem is not limited to CGRA but exists in other well-known parallel architectures such as VLIW or SIMD machines [Anido et al. 2002; Arbelo et al. 2007; Dang 2005; Han et al. 2010; Huang et al. 2009; Mahlke et al. 1995; Shin et al. 2005].

2.2. Predicated Execution Techniques and their Role in Parallel Architectures

Predicated execution is a technique to remove control flows with architectural support. 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*. This technique is essential for architectures exploiting Data-Level Parallelism (DLP) since the architectures cannot parallelize loops whose body has control flows due to the lack of resources handling branches. Thus, SIMD processors and CGRAs typically adopt predicated execution to leverage on DLP for loops with control flows [Anido et al. 2002; Arbelo et al. 2007; Han et al. 2010; Huang et al. 2009; Shin et al. 2005].

3. CONVENTIONAL PREDICATED EXECUTION

Through this and next sections, we present detailed comparison of two 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 CGRA. The characteristics of CGRA 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 1(a) 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* 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

Predication is classically divided into two categories, *partial* and *full*, according to the range of predicated instruction [Mahlke et al. 1995]. *Partial predication* (PARTIAL) simply adds a number of special predicated instructions such as *conditional mov* to

<pre> 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 } </pre> <p style="text-align: center;">(a) C code</p>	<pre> 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: </pre> <p style="text-align: center;">(b) branch</p>
--	--

Fig. 1. An example C code and its branch equivalent.

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	1t	Yes	eq
6 cmov eq R1 R3	No	1t	Yes	eq
7 cmov eq R2 R4	No	1t	Yes	eq
8 cmov neq R1 R5	Yes	1t	No	eq
9 cmov neq R2 R6	Yes	1t	No	eq

Fig. 2. Partial predication (PARTIAL).

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 1(a) is converted to two instructions as shown in Figure 2: one for storing the result of a normal addition to a temporary register R3 (line 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 [Mahlke et al. 1995].

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 kernels¹, 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 rely on software-level approaches, which generally degrade performance.

¹CGRAs 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 [ARM NEON 2013; Dasika et al. 2010; Fijany and Hosseini 2011; Mei et al. 2004].

Assembly Code			R0 == 0		R0 == 1	
			Executed?	Flag	Executed?	Flag
1	uc	cmp R0 #1	Yes		Yes	
2	eq	add R1 R1 #1	No	1t	Yes	eq
3	eq	add R2 R2 #1	No	1t	Yes	eq
4	neq	sub R1 R1 #1	Yes	1t	No	eq
5	neq	sub R2 R1 #1	Yes	1t	No	eq

Fig. 3. Condition-based full predication (CONDFULL).

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, our 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 our experiments, average 1.75 more registers are required per PE, and one example (itp1) cannot be accelerated by 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 8.1. 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

Unlike partial predication, full predication makes all instructions predicated. *Condition-based full predication* (CONDFULL) is a type of full predication that 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 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 [Mahlke et al. 1995]. 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.

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

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	1t
x = 0;	No	3	eq mov R2 #0	No	1t
else		4			
x = 1;	No	5	neq mov R2 #1	Yes	1t

(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	1t
x = 0;	No	5	eq mov R2 #0	No	1t
if (a == 1 && b != 1)	Yes	6	uc testneq R4 R1 #1	Yes	1t
		7	uc and R4 R3 R4	Yes	1t
		8	uc cmp R4 #0	Yes	1t
		9	neq mov R2 #1	No	1t
x = 1;	No				

(b) flattened

Fig. 4. An example program that naïve conversion to CONDFULL produces incorrect assembly code due to its *nested-if* structure.

is determined only by the most recent comparison instruction, and that makes CONDFULL hard to handle *nested-if* structures. As an example, Figure 4(a) 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 4(a) could be converted to the one in Figure 4(b). 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 our predication technique that can support nested structures.

4. PROPOSED PREDICATED EXECUTION

This section introduces two novel predication techniques, one for low-power and *nested-if* structures and the other for high performance. We use the same example (Figure 1(a)) as the previous section for the explanation.

4.1. State-Based Full Predication

State-based full predication is yet another type of full predication proposed in Han et al. [2012]. 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

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

Fig. 5. Counter-based state transition technique for state-based full predication (STATEFULL).

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, we can execute either the if-path or else-path selectively.

There can be several ways to control the state of each PE, but we propose *state-based full predication using counter* (STATEFULL), which uses *per-PE counters* and a *sleep* instruction. The sleep instruction changes the state of a PE from *AWAKE* state to *SLEEP* state, which has two operands: *condition* and *offset*. When a sleep instruction is invoked, the PE checks the status flag and enters into *SLEEP* state only if the flag value satisfies the condition. Then 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 5 shows the mechanism of STATEFULL. “csleep neq 3” denotes a sleep instruction that changes the state of the PE to *SLEEP* state and keeps the state 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, we assume that no operations require varying number of cycles (e.g., data-dependent operation cycles) since such operations can be hardly supported by CGRA 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-multicyle 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 multicyle operations at all. Lastly, the architecture can be stalled dynamically due to runtime conditions. For example, ADRES architecture [Mei et al. 2004] 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.

For programs that do not have any *nested-if* structures, STATEFULL could incur more performance overhead compared to CONDFULL. This is mainly because it needs to insert special sleep instructions to control the states of PEs. This performance overhead can be relatively large for *short-if* statements (*short-if* means the body of

Fig. 6. The concept of the DISE technique.

the *if* structure is short; refer to Section 8.2 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 or 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 we implement it through clock-gating techniques. Note that it is impossible for CONDFULL since the PEs recognize the predicate only after the instruction has been decoded and the corresponding condition has been evaluated. As a result, our approach reduces power by 43.4% compared to CONDFULL (refer to Section 8.1).

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 configuration memory as well.

4.2. Dual-Issue-Single-Execution (DISE)

We propose another novel approach called Dual-Issue-Single-Execution (DISE) to accelerate execution of control flows. Figure 6 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

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

Fig. 7. Dual-Issue-Single-Execution (DISE).

configuration memory space. In case of *normal code* that has no control flow in it², 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 our 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 6.3.2 for the implementation detail).

Figure 7 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 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 [Han et al. 2010]. 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 [Chang and Choi 2008]. 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 4.1, 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

²Hereafter, we call the portions of the program corresponding to if-else paths as *if-else code* and all others as *normal code*.

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	

Fig. 8. Additional nop instructions inserted for DISE to balance the lengths of if- and else-paths.

(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 8.6. 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 8. 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 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.

5. HYBRID PREDICATION

5.1. Motivation

The first four rows of Table I summarize the characteristics of predication techniques introduced in the previous section. Although CONDFULL shows relatively good performance in both *short-if* and *long-if* structures, 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 characteristics are rather specialized to specific kinds of programs (*short-if* structures for PARTIAL and balanced *if-else* structures for DISE).

Table I. 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

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

Fig. 9. An example program that PARTIAL executes with better performance than STATEFULL.

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 I. 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.

5.2. 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 9.

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 8.3.

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_csleeeep uc 2	AWAKE		FALSE	lt
5	add R3 R3 #1		SLEEP	2	TRUE	lt
6	add R4 R4 #1		SLEEP	1	TRUE	lt

Fig. 10. A solution to eliminate the need for nop instructions (applied to the example code shown in Figure 8).

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 [Mahlke et al. 1995]. 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 [Mahlke et al. 1995]).

5.3. 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. Our solution is adding a predicated instruction that has the capability of changing the state of a PE to SLEEP state and, at the same time, alternating between true-path and false-path. This special instruction, `changepath_csleeeep`, eliminates the need for extra nop instructions completely, thereby reducing the size of code so dynamic power consumption as well. Figure 10 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_csleeeep` 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 subinstruction in Figure 10 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 11 shows such an example. Figure 11(a) shows the C code and Figure 11(b) shows the case of applying DISE to the outermost *if-else* structure while Figure 11(c) shows the case of applying it to the innermost one. Figure 11(d) 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

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

(a) example

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

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

(c) applying DISE into the innermost *if-else* structure(d) applying DISE into both *if-else* structuresFig. 11. Applying DISE and STATEFULL together into *nested-if* structures for better performance.

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 candidates³.

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.

6. IMPLEMENTATION DETAILS

6.1. Baseline Architecture

We implemented our proposed techniques on a CGRA called FloRA [Kim et al. 2005; Lee et al. 2009]. It mainly targets the embedded system applications having ILP or 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 [Lee et al. 2009].

³Identifying the best candidate efficiently remains to be a problem to be solved. In our 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.

Fig. 12. The target architecture FloRA.

The overall architecture is shown in Figure 12. 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 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.

One of the main features of FloRA is the loop pipelining technique as depicted in Figure 13 [Kim et al. 2005, 2006]. 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, and thus saves power consumption in accessing the configuration memory. Also, it simplifies the programming model for shared area-critical resources, such as multipliers, by allowing each PE in the same row to use the shared resources in a round-robin manner.

6.2. Conventional Predicated Execution Techniques

6.2.1. Partial Predication. 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 into our architecture because the select instruction had four operands, whereas our ISA allowed only three-address form.

Fig. 13. Architecture for loop pipelining technique for FloRA.

6.2.2. Condition-Based Full Predication. 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 7.5KB ($=3072 \times 20\text{bits}$) to 8.625KB ($=3072 \times 23\text{bits}$).

6.2.3. Another State-Based Full Predication (PSEUDOBRANCH). There is an approach that can be classified into state-based full predication (PSEUDOBRANCH) [Anido et al. 2002] in that it also uses the concept of a *state*. They propose to use an explicit instruction to terminate SLEEP state, which is called the *awake* instruction. In order to support *nested-if* structures, they associate each sleep instruction with a tag. Each PE stores the tag of the most recently executed sleep instruction into its own *tag register*, and the PE in SLEEP state wakes up only when an awake instruction with the same tag is invoked. Although this mechanism enables efficient handling of *nested-if* structures, it hinders PEs from saving power on unnecessary paths since the PEs need to decode every instruction even in SLEEP state to check for an awake instruction.

To show the difference from our approach, we also implement PSEUDOBRANCH as proposed in Anido et al. [2002]. 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.

6.3. Proposed Predicated Execution Techniques

6.3.1. 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 multicyle 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 our ten applications, the longest *if* structure consists of 143 instructions. Note that this does not imply that our 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.

Fig. 14. Modification of architecture to support DISE.

6.3.2. 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 14 shows how it is combined with our feature shown in Figure 13. 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 $(i \bmod 2)$ -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.

6.3.3. 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 5.3.

7. EXPERIMENTAL SETUP

To evaluate and compare the mentioned predication techniques accurately, we implemented all of them on our architecture at register-transfer level using Verilog HDL. From the RTL description, gate-level circuits were synthesized with 500MHz of target clock frequency using Synopsys Design Compiler with TSMC 45nm 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 [Muralimanohar et al. 2009] to estimate power consumption of the configuration memory also with 500MHz of target clock frequency at 45nm technology library.

Our experiment is conducted on the reconfigurable array and configuration memory but not on data memory since data memory access is affected minimally in our experiment. Predication could possibly affect data memory access in two ways. According to Mahlke et al. [1995], 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 our 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 [Han and Choi 2009] 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 8.2) 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 II. 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

Table II. The Detailed Information of the Applications

	SHORT-IF						LONG-IF			
	idct	chroma	max	sad	shift	round	seceded	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%

measurement. The average execution cycles of SHORT-IF and LONG-IF are 45.80 and 172.75 cycles, respectively.

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).

8. EXPERIMENTAL RESULTS

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

8.1. 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 15 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.

Fig. 15. Power consumption of a PE on an unnecessary path. Here *etc.* includes power consumption of state registers and wires.

Fig. 16. Energy normalized to that of STATEFULL.

8.2. 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 5.2, 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 16. 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*.

8.3. Compilation Strategy in STATEFULL+PARTIAL

Figure 16 also shows why we use PARTIAL for only special cases where *short-if* structures are only composed of *mov* instructions in our combined approach of STATEFULL+PARTIAL. PARTIAL consumes much more energy than CONDFULL 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.

8.4. 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 our STATEFULL. Also, our 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.

Fig. 17. Execution time normalized to that of STATEFULL.

8.4.1. Performance. Figure 17 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 applications 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, our 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.

8.4.2. Energy Consumption of the Reconfigurable Array. Figure 18(a) shows energy consumption of the reconfigurable array. For SHORT-IF, PARTIAL and CONDFULL consumed less energy than PSEUDOBRANCH and 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 17) and power consumption of a PE (Figure 15).

8.4.3. Energy Consumption of Configuration Memory. Predication mechanisms also affect energy consumption of the configuration memory, which is shown in Figure 18(b). In most cases, PARTIAL and STATEFULL consumed less energy on the configuration memory than CONDFULL. 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 in the case of STATEFULL, which suffered from relatively high performance overhead (see Figure 17), and thus the increase in the number of fetched instructions.

Fig. 18. Energy consumption normalized to that of STATEFULL.

8.5. Proposed Hybrid Predication Techniques

We hybridize STATEFULL with PARTIAL and DISE to improve performance in *short-if* and *if-else* structures, respectively. Figure 19(a) 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 19(b) 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*). The difference in energy consumption between STATEFULL and the two hybrid mechanisms is not significant, but it means the hybrid approaches also significantly improve the energy consumption compared with the three existing approaches as STATEFULL does, which is proven in the previous subsections.

8.6. Putting Together

Table III and Table IV summarize all results discussed so far⁴. We define *improvement* as follows to reflect that less energy/delay/EDP (energy-delay product) is better.

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

⁴LONG-IF and Total for PARTIAL do not include the results for *itpl* since PARTIAL could not be used for it, as explained in Section 8.4.1.

Fig. 19. Comparison among STATEFULL and hybrid approaches. The values are normalized to the results of STATEFULL.

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 thus to get lower EDP. Also, compared to CONDFULL, it reduced much energy consumption and EDP in LONG-IF, which is our main target. 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%⁵ overhead in power consumption. As a result, it improved EDPs by 25.7%, 27.9%, and 31.4% compared to the conventional mechanisms of PARTIAL, CONDFULL, and PSEUDOBRANCH, respectively. Considering that there is a trade-off between performance

⁵This can be inferred from the comparison with STATEFULL+PARTIAL for SHORT-IF in Table IV because DISE is not used for SHORT-IF.

Table III. 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%

Table IV. 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%

and energy in general, this hybrid approach provides a unique merit that enables an energy-efficient acceleration of control flow execution.

9. RELATED WORK

Mahlke et al. [1995] classify predication techniques into partial and full predication and compare their effects on performance in the domain of ILP, not of DLP. Their 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* [Han et al. 2012].

There have been several works related to the state-based full predication. Huang et al. [2009] reveal that state-based full predication can virtually implement full predication only at the cost of partial predication. Anido et al. [2002] emphasize that state-based full predication can handle *nested-if* structures, whereas condition-based full predication cannot. Han et al. [2010] propose an efficient mechanism for controlling the state using a sleep counter, thereby reducing performance overhead compared to Anido et al. [2002].

There has been much research related to predicated execution so far including the papers mentioned previously. However, to the best of our knowledge, there has been no attempt to either analyze or reduce power consumption overhead of predicated execution. Instead, most architecture-level [Arbelo et al. 2007; Dang 2005] and compiler-level [Chuang et al. 2003; Shin et al. 2005] studies have concentrated on performance improvement and/or design automation. Other research by Choi et al. [2001] and Quinones et al. [2007] focus on the effect of predication on branch prediction, which also does not consider power consumption of predicated execution.

In the domain of CGRA, there have been several papers reducing power consumption of the architecture. Kim et al. [2006] and Nishimura et al. [2008] propose a technique to reduce power consumption in a configuration cache. Lambrechts et al. [2008] try to save power consumption through interconnection optimization. However, none of them considers the effect of predication in CGRA.

10. CONCLUSION

This article 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 article 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, and DISE improved energy-delay product by 11.9% to 23.8% on average over all applications used for the experiments, and by 25.7% to 31.4% on average over the main target applications. We believe that the proposed mechanisms could be competitive candidates for a “universal predication technique” that makes better use of data-level parallelism in CGRA.

REFERENCES

- Anido, M. L., Paar, A., and Bagherzadeh, N. 2002. 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*.
- Arbelo, C., Kanstein, A., Lopez, S., Lopez, J., Berekovic, M., Sarmiento, R., and Mignolet, J. Y. 2007. 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*.
- ARM NEON. 2013. The ARM[®] NEON[™] general-purpose SIMD engine.
<http://www.arm.com/products/processors/technologies/neon.php>.
- Becker, J. and Vorbach, M. 2003. Architecture, memory and interface technology integration of an industrial/academic configurable system-on-chip CSoC. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*.
- Chang, K. and Choi, K. 2008. Mapping control intensive kernels onto coarse-grained reconfigurable array architecture. In *Proceedings of the International SoC Design Conference*.
- Choi, K. 2011. Coarse-grained reconfigurable array: Architecture and application mapping. *IPSJ Trans. Syst. LSI Des. Methodol.* 4, 31–46.
- Choi, Y., Knies, A., Gerke, L., and Ngai, T. 2001. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the International Symposium on Microarchitecture*.
- Chuang, W., Calder, B., and Ferrante, J. 2003. Phi-predication for light-weight if-conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*.
- Dang, P. 2005. An efficient implementation of in-loop deblocking filters for H.264 using VLIW architecture and predication. In *Proceedings of the International Conference on Consumer Electronics (Digest of Technical Papers)*.

- Dasika, G., Woh, M., Seo, S., Clark, N., Mudge, T., and Mahlke, S. 2010. Mighty-morphing power-SIMD. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*.
- Fijany, A. and Hosseini, F. 2011. Image processing applications on a low power highly parallel SIMD architecture. In *Proceedings of the IEEE Aerospace Conference*.
- Garzia, F., Hussain, W., and Nurmi, J. 2009. CREMA: A coarse-grain reconfigurable array with mapping adaptiveness. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.
- Han, K. and Choi, K. 2009. Library-based mapping of application to reconfigurable array architecture. *J. Semiconductor Technol. Sci.* 9, 209–215.
- Han, K., Paek, J. K., and Choi, K. 2010. Acceleration of control flow on CGRA using advanced predicated execution. In *Proceedings of the International Conference on Field-Programmable Technology*.
- Han, K., Park, S., and Choi, K. 2012. State-based full predication for low power coarse-grained reconfigurable architecture. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*.
- Hartenstein, R. 2001. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*.
- Huang, L., Shen, L., Ma, S., Xiao, N., and Wang, Z. 2009. DM-SIMD: A new SIMD predication mechanism for exploiting superword level parallelism. In *Proceedings of the International Conference on ASIC*.
- Kim, Y., Kiemb, M., Park, C., Jung, J., and Choi, K. 2005. 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*.
- Kim, Y., Park, I., Choi, K., and Paek, Y. 2006. 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*.
- Lambrechts, A., Raghavan, P., Jayapala, M., Cathoor, F., and Verkest, D. 2008. Energy-aware interconnect optimization for a coarse grained reconfigurable processor. In *Proceedings of the International Conference on VLSI Design*.
- Lee, D., Jo, M., Han, K., and Choi, K. 2009. FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In *Proceedings of the International Conference on Field-Programmable Technology*.
- Mahlke, S. A., Hank, R. E., McCormick, J. E., August, D. I., and Hwu, W. W. 1995. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the International Symposium on Computer Architecture*.
- Mei, B., Vernalde, S., Verkest, D., and Lauwereins, R. 2004. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*.
- Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. 2009. CACTI 6.0: A tool to model large caches. Tech. rep. HPL-2009-85, HP Laboratories.
- Nishimura, T., Hirai, K., Saito, Y., Nakamura, T., Hasegawa, Y., Tsutsusmi, S., Tunbunheng, V., and Amano, H. 2008. Power reduction techniques for dynamically reconfigurable processor arrays. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.
- Quinones, E., Parcerisa, J. M., and Gonzalez, A. 2007. Improving branch prediction and predicated execution in out-of-order processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- Saito, Y., Sano, T., Kato, M., Tunbunheng, V., Yasuda, Y., Kimura, M., and Amano, H. 2010. MuCCRA-3: A low power dynamically reconfigurable processor array. In *Proceedings of the Asia and South Pacific Design Automation Conference*.
- Shin, J., Hall, M., and Chame, J. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization*.

Received June 2012; revised October 2012; accepted December 2012