# State-Based Full Predication for Low Power Coarse-Grained Reconfigurable Architecture

Kyuseung Han, Seongsik Park, and Kiyoung Choi
School of Electrical Engineering and Computer Science
Seoul National University, Seoul, Korea
{ darprin, pss015, kchoi }@dal.snu.ac.kr

***Abstract*—It has been one of the most fundamental challenges in architecture design to achieve high performance with low power while maintaining flexibility. Parallel architectures such as coarse-grained reconfigurable architecture, where multiple PEs are tightly coupled with each other, can be a viable solution to the problem. However, the PEs are typically controlled by a centralized control unit, which makes it hard to parallelize programs requiring different control of each PE. To overcome this limitation, it is essential to convert control flows into data flows by adopting the predicated execution technique, but it may incur additional power consumption. This paper reveals power issues in the predicated execution and proposes a novel technique to mitigate power overhead of predicated execution. Contrary to the conventional approach, the proposed mechanism can decide whether to suppress instruction execution or not without decoding the instructions and does not require additional instruction bits, thereby resulting in energy savings. Experimental results show that energy consumed by the reconfigurable array and its configuration memory is reduced by up to 23.9%.**

***Index Terms*—CGRA; reconfigurable architecture; predication; predicated execution; low power design;**

## I. Introduction

Achieving high performance with low power, while maintaining flexibility, has been considered as one of the most fundamental challenges in architecture design. High performance is the number one goal at all time, and low power becomes essential to make the system feasible. Especially, as high-end mobile devices like smartphones become popular, low power design is more emphasized to prolong battery lifetime.

Coarse-grained reconfigurable architecture (CGRA), which has a tightly coupled cluster of processing elements (PEs), can be a viable solution to meet both requirements at the same time. It has simple architecture design and allows efficient parallel processing through structuring PEs tightly coupled with each other and controlled by a centralized unit. Although these characteristics make CGRA hard to exploit task-level parallelism (TLP), they lead to better performance-to-power ratio even compared to GPU when exploiting ILP/DLP. This is because GPU architecture needs complicated mechanism for collaboration of active PEs. Especially for embedded systems, where type of parallelism is simple and power consumption is important, CGRA can be an effective solution.

However, in spite of the relative effectiveness, CGRA still consumes considerable power compared to other solutions for embedded systems like ASIC. Though it has a great merit in flexibility and time-to-market compared to ASIC, high power consumption makes it difficult to be integrated into embedded systems. Thus, power reduction is one of the most important challenges that CGRA is facing.

There have been several attempts to reduce power in CGRA [1]–[3]. To the best of our knowledge, however, no one has tried to reduce power related to predicated execution. It has been taken for granted to adopt predicated execution technique into CGRA regardless of how much power is increased since CGRA has a great limitation in handling control flow and predication is the only solution known so far. Thus, in this paper, we reveal power issues in applying the predicated execution and propose a novel technique to reduce power consumption. Following are the main contributions of the paper.

- We reveal the power issues of predicated execution not only in the domain of CGRA, but in all domains related to predicated execution. Most previous researches on predicated execution have concentrated only on performance [4], compiler [5], [6], and/or architecture [7], [8], and no one has considered power.
- We propose a novel predication mechanism to mitigate power overhead of predicated execution. Conventional full predication technique requires additional instruction bits for condition field and has inefficient predication mechanism always requiring decoding of instruction, which results in significant power overhead. Our approach does not incur such wastes and thus can save power in the reconfigurable array as well as in the memory that stores the configuration code.
- We implement both the conventional full predication and the proposed one into CGRA at the register transfer level (RTL) and measure the power at the gate-level after synthesis.

## II. Background

### A. Predicated execution technique

Predicated execution is a technique to convert control flows to data flows. Conventional predicated execution techniques can be classified into two types: *full predication* and *partial predication* [4]. The goal of partial predication is to manage control flows with negligible modifications to the original design. On the other hand, full predication can achieve more speedup at the cost of additional hardware. Partial predication is based on speculation, so unnecessary memory operation and register pressure can degrade the performance significantly compared to full predication. Thus it is appropriate to use partial predication only for short if-clauses. Full predication can overcome these problems by suppression mechanism, which is the essence of full predication. To support full predication, every instruction has a *condition* as an additional operand, which defines the predicate of the instruction. That is, the instruction is executed if the predicate is true (i.e., the condition is met); otherwise, the instruction is suppressed. Full predication is free from the overhead of speculation and heavy register pressure, resulting in better performance compared to partial predication. Therefore, it may be inevitable to choose full predication for high performance. The downside of it is that the number of bits for the instruction encoding should be increased to hold the condition within the instruction and the instruction set architecture (ISA) should be modified significantly to implement the suppression mechanism.

### *B. Role of predicated execution on CGRA*

CGRA has a critical limitation in exploiting parallelism in the presence of control flow due to passive characteristic of PEs. Since each PE cannot select its own instruction flow, but the centralized unit controls all the PEs, only one control flow can be handled at a time. The same problem exists in SIMD machines due to the same reason, and it is actually a more vital problem to SIMD machines since instructions are always broadcast to all PEs. SIMD machines cannot exploit any DLP if there is a control flow. Hence, predication is usually adopted to CGRAs and SIMD machines to tackle the bottleneck due to control flow [5], [8], [9]. It allows each PE to have its own execution even if the common instructions are supplied so that it can maximize the parallelism regardless of the existence of control flow.

## III. Proposed Predicated Execution Technique

### *A. Motivation*

Conventional full predication wastes power in two aspects. One is that the increased number of bits for instruction encoding incurs more power consumption in configuration memory even when the processor does not execute control flow. The other is that a PE fetches and executes instructions on *unnecessary paths*, which are nullified by the predication mechanism. The PE should decode even the instructions on an unnecessary path since the condition can be checked only after decoding. To avoid such wastes, we propose a novel mechanism based on state instead of condition. In the rest of the paper, we call conventional full predication as *condition-based full predication* (CFP) and our proposed one as *state-based full predication* (SFP) according to the criterion for determining predication.

### *B. Mechanism*

The proposed SFP technique is based on two states: *awake* and *sleep*. As mentioned above, the states play the role of predicates. That is, a PE executes instructions normally in the awake state and suppresses the execution in the sleep state. By controlling the state of each PE using a special instruction, we can execute either if-part or else-part selectively.

To change the state of a PE from the awake state to the sleep state, we use a *sleep* instruction, which has a structure similar to a branch instruction. It holds a condition and the offset as operands. The offset is the difference between the current address and the target address. The sleep instruction is executed when the PE is awake and the condition is met. It is different from a branch instruction in that a sleep instruction makes the PE sleep during the period corresponding to the address difference (*sleep period*) instead of branching to the target address. This mechanism can be implemented easily by using a counter. When going into the sleep state, the counter initializes its count value with the sleep period specified in the instruction and then counts down every cycle until the value becomes zero. Right after the value becomes zero, the PE wakes up automatically.

| | | |
|---|---|---|
| `for(i=0; i<8; i++)` | `load R0 cond0[i]` | `load R0 cond0[i]` |
| `{` | `cmp R0 #1` | `cmp R0 #1` |
| ` if(cond0[i]==1)` | `b neq pc+11` | **`sleep neq #9`** |
| `  if(cond1[i]==1){` | `load R1 cond1[i]` | `load R1 cond1[i]` |
| `   x[i] = a;` | `cmp R1 #1` | `cmp R1 #1` |
| `   y[i] = a;` | `b neq pc+5` | **`sleep neq #3`** |
| `   z[i] = a;` | `store a x[i]` | `store a x[i]` |
| `  }` | `store a y[i]` | `store a y[i]` |
| `  else{` | `store a z[i]` | `store a z[i]` |
| `   x[i] = b;` | `b uc pc+3` | **`sleep uc #1`** |
| `   y[i] = b;` | `store b x[i]` | `store b x[i]` |
| `  }` | `store b y[i]` | `store b y[i]` |
| ` else` | `b uc pc+2` | **`sleep uc #0`** |
| `  x[i] = c;` | `store c x[i]` | `store c x[i]` |
| `}` | | |
| (a) | (b) | (c) |

Fig. 1. (a) An example of loop code with nested if-else structure, (b) Assembly-level code of the loop body for a scalar processor, and (c) Assembly-level code of the loop body using the proposed technique.

Fig. 1 and Fig. 2 take an example to show how this mechanism works. Fig. 1(a) is a simple example of a loop with nested if-structure written in the C language. Fig. 1(b) shows the assembly code obtained by compiling the C code using branch operations. The `cmp` instruction compares two operands and modifies the flag values to be used for the following predicated instruction execution. The `b` instruction is a branch instruction. The `eq` and `neq` represent "equal" and "not equal" condition code, and the `uc` represents "unconditional" branch. For example, the third instruction '`b neq pc+11`' means that if the "not equal" flag (evaluated in the second instruction '`cmp R0 #1`') is set, then the processor jumps to the target address of *current pc*+11 (`store c x[i]`). If the condition is not satisfied, the very next instruction (`load R1 cond1[i]`) is executed.

However, since branch operation is not acceptable for CGRA, we convert it to Fig. 1(c) by applying our technique. Each branch instruction is replaced by a *sleep instruction*, which specifies the sleep period in the number of cycles to be in the sleep mode. Actually, the sleep period is calculated statically by the compiler[1] as (target offset − 2). This is because the actual sleep period is (target offset − 1) and the sleep state lasts until the counter value reaches zero, not one. In the case of '`b neq pc+11`', PE should sleep for 10 cycles since target offset is 11 so 10-1 is inserted to sleep instruction as an operand.

Fig. 2 shows the detailed operations of two PEs executing the same code under different conditions when the proposed technique is applied. Suppose that the value of `cond0` is 1 and `cond1` is 0 for the $m^{th}$ PE. Under these conditions, the sleep instruction at cycle 2 is suppressed. However, the sleep instruction at cycle 5 is executed, and the PE changes its state to the sleep state and initializes the counter value to 3 at the next cycle. The PE remains in the sleep state and the counter is decremented every cycle until cycle 9. At cycle 9, the counter

[1]The proposed technique requires only two simple operations added to conventional compilers. One is the replacement of branch instructions by sleep instructions and the other is the calculation of sleep period from offset to the target address. They are simple and cause negligible modification to the compilers, which makes the SFP technique very compiler-friendly.

| cycle | | m th processing element cond0[m] = 1, cond1[m] = 0 | | | n th processing element cond0[n] = 0, cond1[n] = 1 | | |
|---|---|---|---|---|---|---|---|
| | | flag | state | Counter | flag | state | counter |
| 0 | `load R0 cond0[i]` | | awake | | | awake | |
| 1 | `cmp R0 #1` | | awake | | | awake | |
| 2 | `sleep neq #9` | eq | awake | | neq | awake | |
| 3 | `load R1 cond1[i]` | eq | awake | | neq | sleep | 9 |
| 4 | `cmp R1 #1` | eq | awake | | neq | sleep | 8 |
| 5 | `sleep neq #3` | neq | awake | | neq | sleep | 7 |
| 6 | `store a x[i]` | neq | sleep | 3 | neq | sleep | 6 |
| 7 | `store b y[i]` | neq | sleep | 2 | neq | sleep | 5 |
| 8 | `store c z[i]` | neq | sleep | 1 | neq | sleep | 4 |
| 9 | `sleep uc #1` | neq | sleep | 0 | neq | sleep | 3 |
| 10 | `store d x[i]` | neq | awake | | neq | sleep | 2 |
| 11 | `store e y[i]` | neq | awake | | neq | sleep | 1 |
| 12 | `sleep uc #0` | neq | awake | | neq | sleep | 0 |
| 13 | `store c x[i]` | neq | sleep | 0 | neq | awake | |
| 14 | … | neq | awake | | neq | awake | |

Fig. 2. The execution sequence of different processing elements when adopting the proposed technique.

value becomes 0, and so the PE wakes itself up at the next cycle to resume normal execution. Another example is shown on the next column of Fig. 2 for the $n^{th}$ PE running the same code with condition values of `cond0` = 0 and `cond1` = 1, which can be followed easily.

### C. Capability for low power design

The SFP technique has several good characteristics for low power design. As shown in the examples, the SFP executes control flows as if all instructions were predicated. Different from CFP, however, SFP can achieve such effect without adding any additional field to the instruction word, thereby leading to energy savings in the configuration memory.

Another low power feature comes from the predication mechanism. A PE knows a *priori* whether the next instruction will be executed or not before decoding the instruction, since the predicate does not depend on the condition in the instruction but depends on the state provided by the counter. Thus, when the PE is in the sleep state, it does not need to do anything but just count down until it wakes up. It means that we can block any changes of values in all registers and all combinational logics except some small modules related to the counter; we can block modifying the content of the instruction register as well as decoding the instruction. In particular, if the circuit for supporting this feature is implemented through clock gating, then we can reduce dynamic power dramatically. It is impossible in CFP since the PE can recognize the predicate only after decoding the instruction and checking the condition.

### D. Hybrid approach

The proposed SFP technique has some performance overhead due to the insertion of sleep instruction to control if-structure, which in turn will affect energy consumption. If the if-clause is long, the overhead can be compensated by large power savings on the unnecessary path. However, it may not be the case when the if-clause is short. To mitigate this overhead, we propose to hybridize SFP with partial predication (PP). As explained in Section II-A, PP provides good performance for short if-clause, and thus the hybrid approach handling short if-clauses using PP and long ones using SFP can provide a robust way to reduce power without incurring large performance degradation.

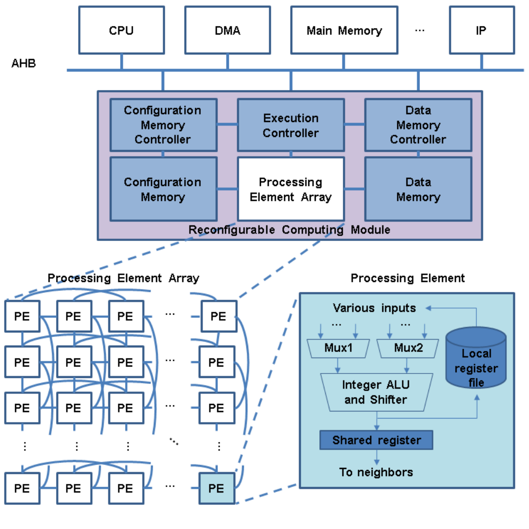


Fig. 3. Overall FloRA architecture.

## IV. Hardware Implementation

### A. Baseline architecture

We have implemented our proposed techniques on a coarse-grained reconfigurable array architecture called FloRA [10], [11]. The overall architecture is shown in Fig. 3. It has an 8x8 PE array with configuration memory and data memory. Each PE is comprised of an integer ALU, a shifter, and a local register file, and can be dynamically reconfigured every cycle if needed. The CGRA has been implemented on a chip and its functionality and performance have been verified [10].

FloRA is an attractive architecture for exploiting both ILP and DLP. The array processing was originally intended to exploit ILP, but the loop pipelining technique [11] allows the CGRA to exploit DLP also. The technique reduces configuration memory size and power dramatically in a way similar to SIMD and enables efficient resource sharing at the same time. Although FloRA has little support for control flows, it has abundant resources for ILP and DLP. This makes

TABLE I
SYNTHESIS RESULTS OF RECONFIGURABLE ARRAY

| | CFP | SFP | SFP+PP |
|---|---|---|---|
| Area (μm$^2$) | 291,285 | 294,401 | 294,581 |
| Ratio | - | 1.07% | 1.13% |

the architecture very appropriate for implementing predicated execution.

## B. Implementation details

To compare the proposed SFP approach with CFP on power reduction, we have implemented both approaches using Verilog at the RTL. For CFP, the instruction word length has been increased from 20 bits to 23 bits for holding three bits of condition and the suppression mechanism has been implemented in such way that the decoder outputs disable writing into the registers and latches. For SFP, the instruction word length remains to be 20 bits, but an 8-bit sleep counter and an 1-bit register indicating current state of a PE (awake or sleep)[2] are added. Since each PE already has a 3-bit counter to support multi-cycle operations like multiplication, we have just increased the bit width of the counter to use it as the sleep counter during the sleep mode. For low power design, the state value is used for the clock gating signal to all registers except the counter register that stores the sleep period. For hybrid approach, we implement *conditional move* instruction to support partial predication [4].

We have adopted two basic low power techniques for the base architecture. First, we have used clock gating all over the architecture. Secondly, to reduce the switching in the FUs, we have added latches to their inputs such that the input values do not change when the FUs are not used.

From the RTL descriptions, gate-level circuits for three approaches (CFP, SFP, and SFP+PP) have been synthesized using Synopsys Design Compiler. We have used TSMC 45 nm technology library and set the target clock frequency to 500MHz. We have verified its functional correctness through gate-level simulation using Mentor Graphics ModelSim.

Table I shows the area overhead. Compared to CFP, the proposed SFP and SFP+PP require only about 1% more hardware. The area overhead of SFP comes mainly from the extension of the counters from three bits to eight bits, but it seems to be compensated by the reduction of instruction fetch unit (from 23 bits to 20 bits), resulting in ignorable area overhead.

## V. EXPERIMENTAL RESULT

## A. Experimental setup

To evaluate the effectiveness of the proposed SFP, we have measured the power consumption of the reconfigurable array and configuration memory for two techniques, CFP and SFP. CFP is considered as a baseline and we will show how much the proposed SFP reduces power consumption.

We have measured the power consumption in the reconfigurable array at the gate-level using Synopsys Design Compiler and Mentor Graphics ModelSim. For the configurable memory, we have used CACTI 6.5 [12] for SRAM power estimation since SRAM cell library was not available. Both target 500MHz at 45nm technology.

[2]The 1-bit state register is needed because the non-zero value of the counter does not always indicate the sleep state. The counter is also used to support multi-cycle operations.

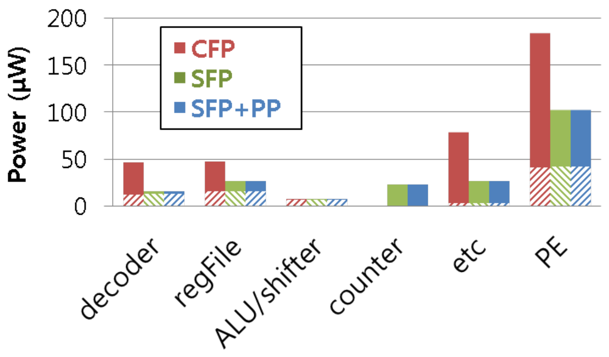


Fig. 4. Comparison of power consumption in one PE. The solid and striped parts mean dynamic and static power, relatively.

For comparison, we have experimented with six examples; `dct_clip`, `chromakey`, `finding_max`, `secded_decoding`, `deblocking`, and `interpolation`. `Dct_clip`, one of compute-intensive parts in JPEC decoder, performs discrete cosine transform and clipping of values into some ranges. `Chromakey` is a technique for compositing two images and `finding_max` is just finding maximum value from a list. `Secded_decoding`, where SECDED means single-error-correction-and-double-error-detection, is error-correcting method used for communication. Its decoding process corrects errors if they occur, and thus has relatively long if-clause compared to the other three examples above. There are various ways for SECDED method, but we use Hamming(8,4). `Deblocking` and `interpolation` are kernels from H.264 video decoder. `Deblocking` is a process to reduce blocking effect on pixels. The pixels are handled differently according to the strength of blocking effect, which makes up heavy control flows. `Interpolation` is a process to interpolate intermediate values between pixels. There are 16 ways of interpolation and control flow is needed to select one of them.

## B. Reconfigurable array

*1) The effects of predication mechanism:* The main power reduction in the reconfigurable array comes from the differences in predication mechanism. To show the effect of sleep (SFP) instead of nullification (CFP), we have measured the power consumed by one PE for both cases. The results are shown in Fig. 4. We have analyzed power consumption for major components of a PE; 'etc' means the rest parts of a PE including state registers and wires between components. The figure shows that there is no big difference in the static power of different approaches regardless of the parts examined. Static power of the counter may be increased, but the actual amount seems ignorable. On the other hand, dynamic power consumption of the decoder, register file, and the 'etc' part in SFP+PP is greatly reduced by 92.9%, 65.8%, and 69.2%, respectively, since SFP does not need instruction decoding during the sleep state so does not incur switching activities in combinational logic gates and wires. Dynamic power consumption of counters is increased due to counting sleep cycles, but the reduction from other parts is much greater, so the total power consumption is reduced by 44.1%.

*2) Examples with short if-clause:* The tendency of result varies according to the length of if-clause (or else-clause)

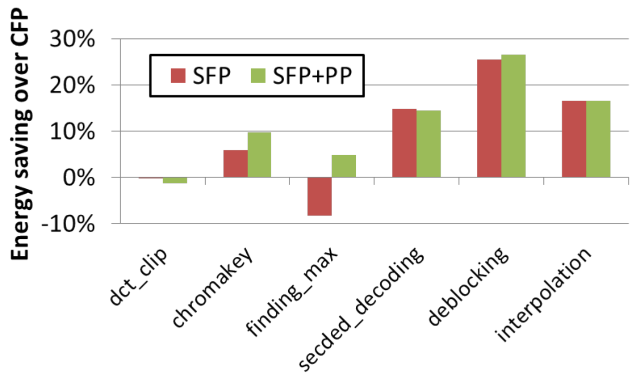


Fig. 5. Energy savings in PE array.

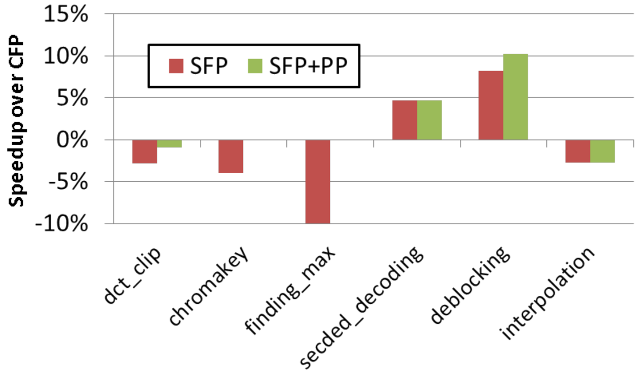


Fig. 6. Performance improvements.

since power consumption depends on whether unnecessary paths are executed or not. First three examples, `dct_clip`, `chromakey`, and `finding_max`, among the above six examples have short if-clauses, so we cannot expect large power reduction. As shown in Fig. 5, energy consumption for some examples is increased due to longer execution time (see Fig. 6). To overcome this drawback, we propose the hybrid approach (SFP+PP) in Section III-D, and we can see from green bars of Fig. 6 that it works. In all three cases, the hybrid approach improves performance compared to the original SFP. As a result, energy consumption of SFP+PP is improved over CFP for `chromakey` and `finding_max`. However, energy consumption of `dct_clip` is still slightly worse. It is because handling control flow by PP has power overhead compared to SFP.

*3) Examples with long if-clause:* The other three examples, `secded_decoding`, `deblocking`, and `interpolation`, have long if-clauses, so they are expected to have significant power reduction. Since its amount is dominant and thus the results of SFP and SFP+PP are almost the same, we explain only comparison between CFP and SFP+PP.

Fig. 5 shows that SFP+PP saves about 14.5% to 26.5% of energy compared to that of CFP. In the cases of `secded_decoding` and `deblocking`, the reduction partially comes from the better performance shown in Fig. 6. It is because SFP deals with nested if-structures without *flattening*, as shown in Section III-B, but CFP cannot handle it directly [8]. CFP should *flatten* them into non-nested structures, which causes performance degradation because of the register pressure for keeping condition values and/or the recalculation of the condition values. On the other hand, the performance of `interpolation` is degraded by SFP+PP because the example does not have a nested-if structure but has a simple *switch* structure. However, it can save 16.5% of energy due to long unnecessary paths.

## C. Configuration Memory

CFP incurs power overhead in configuration memory due to the extension of instruction bit width. Our CGRA architecture has two different types of configuration memory: temporal cache and spatial cache [1]. We assume that the CGRA has 256 entries for temporal cache and 16 entries for spatial cache in both CFP and SFP designs. Thus, total capacity of configuration memory is 8.625kB for CFP and 7.5kB for SFP.

The results for static, dynamic, and total energy savings are shown in Fig. 7. Dynamic energy is determined by multiplying the size of data per read by the number of read operations. The former is determined by the bit width of instructions and the latter is related to the execution time. Basically SFP has a merit in reading instructions due to its narrower instruction bit width, so it consumes less energy in most cases. However, `finding_max` consumes more energy because the number of fetched instructions is greatly increased as shown in Fig. 6. Since SFP+PP overcomes performance overhead in short if-clause, it enhances the results of SFP in the examples having short if-clause, always guaranteeing dynamic energy reduction (4.7% to 23.2%) in all cases.

Static energy seems to have similar tendency. It is affected by area and the execution time. Due to instruction bit width, SFP outperforms CFP in all cases, and SFP+PP improves more on short if-else examples.

The sum of the two is depicted in Fig. 7(c). The amount of total energy reduction ranges from 8.2% to 20.9%. The energy reduction for `interpolation` seems relatively small, but it has large energy saving in the reconfigurable array part.

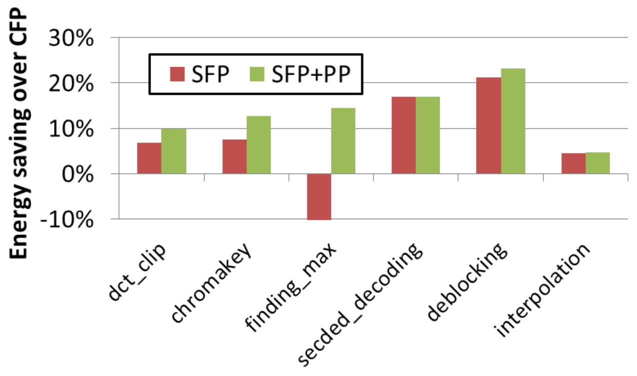
## D. Total energy consumption

Total energy consumption in the reconfigurable array and configuration memory is compared in Fig. 8. We can see that the amount of energy saving is all positive for the hybrid approach. As a result, the energy consumption is reduced by 3.3% to 23.9% and its geometric mean value is 12.6%.

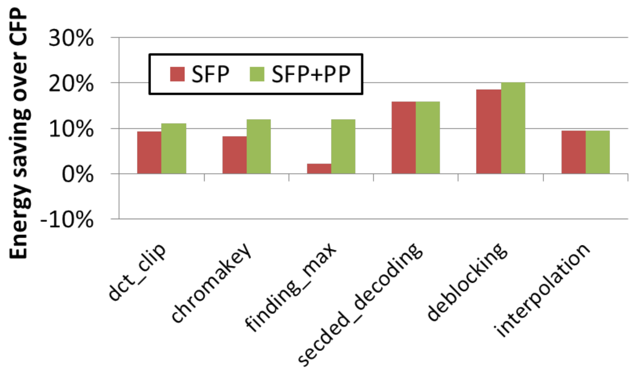
## VI. Related Work

There have been several researches on predication techniques that belong to the category of SFP [7], [8], [13]. [7] has revealed that SFP can virtually implement full predicated execution only at the cost of partial predication. The work in [8] has emphasized that SFP can handle nested-if structures naturally, whereas the conventional full predication cannot. SFP has performance overhead due to the insertion of instructions to change the state according to the control flow, which is not needed in full predication. To mitigate the overhead, [13] has suggested an efficient mechanism for controlling the state using a sleep counter. However, [7], [8], [13] do not consider any power issue in their implementations.

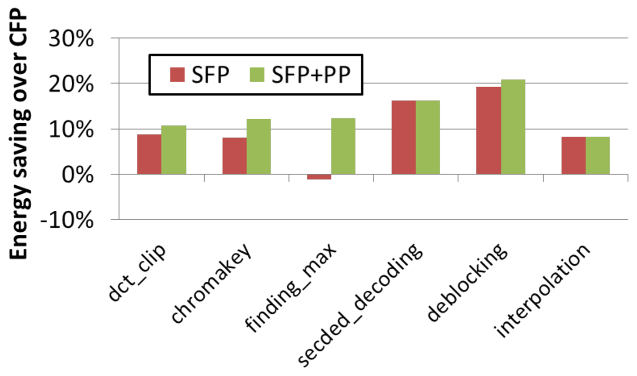
Moreover, to the best of our knowledge, there have been no previous approaches proposed to reduce power consumption related with predicated execution. Most approaches have concentrated only on performance issues involving compiler [4]–[6], the effect on branch prediction [14], etc. [7] and [8]



(a) Dynamic energy



(b) Static energy



(c) Total energy

Fig. 7. Energy savings in configuration memory.

deal with the architecture and the implementation, but there is no attempt to reduce power consumption.

Related to CGRA, there are several attempts to reduce power consumption [1]–[3]. [1] and [2] suggest the way of reducing configuration power and [3] researches on interconnection related to power. However, none of them considers the effect of predication in CGRA.

## VII. Conclusion

We have presented a new type of predication, which we call state-based full predication. It performs like full prediction, but has a more efficient predication mechanism in terms of power consumption compared to conventional full predication.

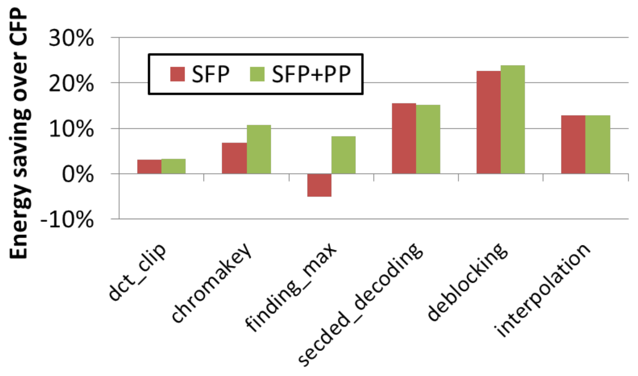


Fig. 8. Total energy savings in reconfigurable array and configuration memory.

When executing unnecessary paths in control flows, processing elements consume 44.1% less power by not decoding the instructions through clock gating. Moreover, by applying the hybrid approach with partial predication, the proposed approach can overcome the most critical drawback of the state-based full predication for the case of short if-clauses, and so improves energy consumption in reconfigurable array and configuration memory. As a result, our proposed mechanism achieves up to 23.9% energy reduction in the reconfigurable array and configuration memory.

## Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2011-0018640).

## References

[1] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in *Proc. ISLPED*, 2006.
[2] T. Nishimura, K. Hirai, Y. Saito, T. Nakamura, Y. Hasegawa, S. Tsutsusmi, V. Tunbunheng, and H. Amano, "Power reduction techniques for dynamically reconfigurable processor arrays," in *Proc. FPL*, 2008.
[3] A. Lambrechts, P. Raghavan, M. Jayapala, F. Catthoor, and D. Verkest, "Energy-aware interconnect optimization for a coarse grained reconfigurable processor," in *Proc. VLSI Design*, 2008.
[4] 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 *Proc. ISCA*, 1995.
[5] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *Proc. CGO*, 2005.
[6] W. Chuang, B. Calder, and J. Ferrante, "Phi-predication for light-weight if-conversion," in *Proc. PLDI*, 2003.
[7] L. Huang, L. Shen, S. Ma, N. Xiao, and Z. Wang, "DM-SIMD: a new SIMD predication mechanism for exploiting superword level parallelism," in *Proc. ASICON*, 2009.
[8] M. L. Anido, A. Paar, and N. Bagherzadeh, "Improving the operation autonomy of SIMD processing elements by using guarded instructions and pseudo branches," in *Proc. DSD*, 2002.
[9] 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 *Proc. DATE*, 2007.
[10] D. Lee, M. Jo, K. Han, and K. Choi, "FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability," in *Proc. FPT*, 2009.
[11] 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 *Proc. DATE*, 2005.
[12] *http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html*.
[13] K. Han, J. K. Paek, and K. Choi, "Acceleration of control flow on CGRA using advanced predicated execution," in *Proc. FPT*, 2010.
[14] 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 *Proc. MICRO*, 2001.