This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

*Digital Object Identifier 10.1109/ACCESS.2017.Doi Number*

**Accelerating Nested Conditionals on CGRA with Tag-based Full Predication Method**

**JIANG SHA1, WENBO SONG1, YU GONG1, AND YINGYING ZHAO2**

1National ASIC System Engineering Research Center, Southeast University, Nanjing 210096, China

2School of Microelectronics, Southeast University, Wuxi, 214061, China

Corresponding author: Jiang Sha (e-mail: 1312sj@seu.edu.cn)

**ABSTRACT** CGRA (Coarse-grained Reconfigurable Architecture) has been widely considered as one of themost promising computing architectures to exploit spatial parallelism. Compared with the typical general-purpose architectures which are instruction-driven, most of the state-of-art CGRAs are designed following the data-driven strategy, leading to difficulties while dealing with control-flow (nested if-then-else structures, NITE). Tackling with this problem, existing techniques such as partial predication and full predication introduce extra conditional *move* and *select* operations, while state-based full predication (SFP) introduces *sleep* and *awake* operations to correctly implement the basic function of NITE. Meanwhile, performancedegradation is also incurred by these redundant operations. In this paper, a novel tag-based full predication (TFP) strategy is proposed, trying to eliminate redundant operations and thus accelerate NITE on CGRAs. The extra tag field is added to each instruction word to implement distributed nullification and parallel tag register (TReg) overwriting. Hardware support for TFP is present, and experimental validation is based on RTL-level simulation with manual mapping. Results show that our method achieves over 30% performance gain on average compared with SFP at the expense of around 5% additional power consumption and ignorable area overhead.

**INDEX TERMS** Reconfigurable architectures, control-flow, nested branch, full predication.

**I. INTRODUCTION**

CGRAs have been considered as one of the most promising architectures, which fill the gap between the General-Purpose Processors (GPPs) and the Application Specific Integrated Circuits (ASICs). Classic GPPs maximize flexibility with purely instruction-driven execution mechanism, while ASICs are commonly data-driven, maximizing performance and power efficiency. CGRAs, featured by data-driven and moderate flexibility at the same time, try to reach a balance between ASICs and GPPs. Previous works pointed out that, CGRAs are more power-efficient compared with its fine-grain reconfigurable counterpart, Field Programmable Gate Arrays (FPGAs) due to word-level reconfiguration granularity. FPGAs are considered to have bit-level reconfiguration granularity relying on look-up tables (LUTs), bringing about performance decay and waste of power. On the contrast, CGRAs are considered to have comparable performance and power efficiency as ASICs [1].

CGRAs benefit from a larger number of processing elements (PEs) compared with GPPs, and thus exploit more data-level and instruction-level parallelism. Compute-intensive loops can be mapped onto CGRAs to achieve better performance. H.264 encoder/decoder, large scale MIMO detection, deep learning and most of the image processing algorithms are popular target applications for CGRAs [2]-[6]. Among them, quite a lot of compute-intensive loops are not free of conditionals. After inspecting benchmarks including SPEC CPU 2006 [7] and some digital signal processing applications, more than 40% compute-intensive loops contain if-then-else (ITE) structure [8]. Another research [9] further declared that, in the SPEC CPU 2006 benchmark suite, more than 70% of conditionals presented in compute-intensive loops are NITE structures. According to the famous Amdahl’s law [10], when the acceleration of compute-intensive parts in a loop has reached a relatively extreme extend, the performance bottleneck will be the control-intensive part. In this case, NITE is the performance bottleneck for CGRAs considering minor acceleration.

|  |  |
| --- | --- |
| VOLUME XX, 2017 | 1 |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



To relieve this problem, partial predication (PP) [11], [12] was proposed and is quite popular nowadays because of its moderate modification to original architecture design. Drawbacks of PP are a) extra operations as conditional *move* and *select* must be introduced, which results in waste of cycles and thus degrades performance; b) large register pressure may occur and register spill [13] may happen because computed results in a single ITE must be kept in registers for the final select operation, which aggravates performance degradation. Dual issue scheme [13] was proposed to reduce register pressure and accelerate ITE execution, but it cannot execute NITE [9]. SFP [14] employed *sleep* and *awake* operation to change PE state and minimize instruction decoding and execution for gaining a better energy profile. SFP is capable of executing both ITE and NITE, but there is still some redundancy in the extra operations introduced by state switching that can be removed.

In order to accelerate the execution of NITE on CGRAs, we propose a novel full predication scheme based on tag scheme. Because the inefficiency of SFP lies in the state-based indirect predication information transfer mechanism, we abandon the state concept. Instead, tags directly transfer predication information and thus remove all the extra operations in SFP and PP, resulting in better performance. At the same time, it can retain the power-saving advantages of SFP.

The rest of this paper is organized as follows. Section II introduces the background and existing methods. Section III describes the proposed method and detailed theoretical performance analysis. Section IV shows the hardware implementation to support our method. Section V shows the results of performance, power consumption, and area overhead compared to the other methods. Finally, section VI states the conclusions.

**II. BACKGROUND AND MOTIVATION**

CGRA is featured by its data-driven nature, which implies that the pending input data-flow is independent of the operation type. Typically, operands delivered to each PE can be divided into three parts: external input data-flow, constant parameters and routing data. External data-flow is often pre-stored in external off-chip memory and buffered in an input FIFO, varying among different iterations of a loop. Constant parameters are usually cached in an on-chip RAM due to the limited data quantity and the requirement of rapid access. Routing data is an intermediate result computed by the adjacent PEs. According to the data-driven scheme, external data-flow and constant parameters are sequentially fed into ALUs in each PE, and the sequence cannot be changed by the configuration information. Data routing direction can be controlled by configuration information in which way the data path can be constructed. Thus, the only dynamic data-flow transferring procedure is data routing among PEs. The other data-flow sequence is statically arranged, removing address-based dynamic load-store operations. In this way,

time and power consumption of unpredictable random memory accesses can be avoided by the data-driven scheme.

Although data-driven feature endues CGRAs with high performance and power efficiency, it brings about the predicament that control-flow becomes the main performance bottleneck of CGRAs. Control-flow often requires dynamic data-flow variation, which is a conflict with the data-driven scheme. Considering traditional GPPs, the instruction-driven feature allows GPPs to dynamically access memory to randomly fetch data from any address. Program counter (PC) further makes GPPs capable of dynamically changing instruction fetch point, resulting in the simplification while dealing with control-flow. CGRA not only does not have PC, but also is constrained by its relatively static and sequential data-flow, making it suffers from the intrinsically poor capability of processing control-flow.

CGRAs targeting compute-intensive applications have been designed in past years [15], [16], [17]. Normally, the computing portion of an application is mapped to CGRA, i.e., the loop body. Therefore, some studies [18], [19] focus on improving the performance of loops and even nested loops by exploring the parallelism. There is also a performance model [20] established to optimize the mapping of loop nests. In addition to control-flow such as nested loops, according to recent works [8] and [9], ITE and NITE structures widely exist in compute-intensive loops, becoming barriers which prevent the further utilization of CGRAs.

Although an efficient mapping method of CDFG (Control Data Flow Graph) based on register allocation [21] is proposed to process the data-flow and control-flow at the same time, it does not help to improve the processing performance. Several other methods have been proposed to endow CGRAs with the capability of processing ITE and NITE structures correctly and effectively.

* 1. [11] is one of the most popular mechanisms featured by modest modification of original architectures. The basic idea of PP is simultaneously executing both the true path and false path of an ITE structure, and employing a *select* operation to conditionally output the results based on predication. With enough PE resources, both paths can be executed parallelly with conditionals, or even speculatively executed before conditionals. PP aggressively takes advantage of a large amount of PE resources but may incur several problems that degrade performance. Firstly, although PP can theoretically resolve deep NITE with structure flattening technique [9], the quantity of conditional *select*/*move* operations must be added. This inevitablyconsumes extra time and PE resources. Secondly, because intermediate results must be reserved until the last *select* operation, register pressure will be unacceptable and may further incur a register spill, severely harming performance.

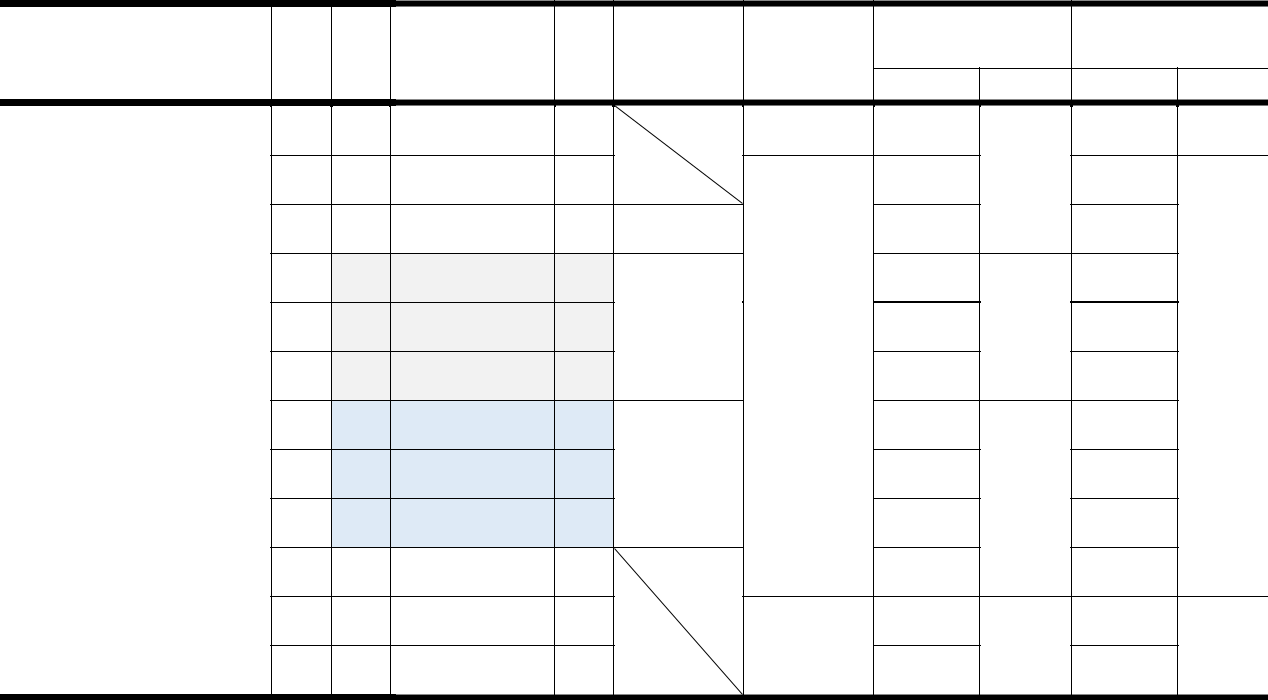
|  |  |
| --- | --- |
| VOLUME XX, 2017 | 9 |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



**TABLE 1. The basic mechanism of TFP. The tag transfer procedure of the TReg is shown in two different kinds of path taken conditions.**



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | **A!=0 and** | | **A==0** | |  |
| **Original C code** | **No.** | **CT** | **TFP** | | **NT** | **Inner Path** | **Outer Path** | **N<0** |  |  |
|  |  |  |  |
|  |  |  |  |  |  |  |  | **TReg** | **Nullify** | **TReg** | **Nullify** |  |
| for(i=0; i<50;i++){ | 1 | T1 | cmp A 0, eq T2 | | T2 |  | Conditional | T1 |  | T1 -> T2 | N |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| if (A != 0){ | 2 | T1 | add N D 1 | | T1 |  |  | T1 | N | T2 |  |  |
| N=D+1; | 3 | T1 | cmp N 0, lt T3 | | T3 | Conditional |  | T1 -> T3 |  | T2 |  |  |
| if(N >= 0) |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| P=(Q-7)\*(Q-8); | 4 | T1 | add P1 Q -7 | | T1 |  |  | T3 |  | T2 |  |  |
| else | 5 | T1 | add P2 Q -8 | | T1 | True Path |  | T3 | Y | T2 |  |  |
|  |  |  |  |
| P=(Q+7)\*(Q+8); | (Tag T1) |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| M=N-1; | 6 | T1 | mul P P1 P2 | | T1 |  | True Path | T3 |  | T2 | Y |  |
| } |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| else{ | 7 | T3 | add P1 Q 7 | | T3 |  |  | T3 |  | T2 |  |  |
| P=1; | 8 | T3 | add P2 Q 8 | | T3 | False Path |  | T3 |  | T2 |  |  |
| M=N+1; | (Tag T3) |  |  |  |  |
|  |  |  |  |  |  |  | N |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  |  |
| 9 | T3 | mul P P1 P2 | | T1 |  |  | T3 -> T1 |  | T2 |  |  |
| } | 10 | T1 | add M N -1 | | T1 |  |  | T1 |  | T2 |  |  |
|  |  |  |  |  |  |
|  | 11 | T2 | mov P | 1 | T2 |  | False Path | T1 |  | T2 |  |  |
|  |  |  |  |  |  |  |  | Y |  | N |  |
|  |  |  |  |  |  |  | (Tag T2) |  |  |  |
|  | 12 | T2 | add M N 1 | | T1 |  | T1 |  | T2 -> T1 |  |  |
|  |  |  |  |  |  |

Full predication (FP) is another mechanism inherited from SIMD architectures [11], [22], [23]. The original FP applies a predication register to each PE and a predication field for each instruction. The predication register (PReg) stores the latest predication value computed by a conditional. Operations in both true and false paths will be sequentially fetched and normally executed when the PReg equals the predication field, otherwise, operations will be nullified. In this way, register pressure is relieved in that only taken paths will be executed and no intermediate results need to be kept for selection. FP cannot resolve NITE, the predication register can only record the latest predication result. This results in an incorrect nullification when the inner ITE is finished and returns back to the outer ITE, because the predication register has been overwritten by the predication value of the inner ITE conditionals.

Dealing with NITE, SFP [14], [23] was proposed as an advanced version of FP, capable of executing NITE structure. Two states marked as *sleep* and *awake* are introduced to decide whether or not to execute instructions on a certain path. A 1-bit state register is embedded in each PE to indicate the current state. State registers are dynamically overwritten depending on the latest predication. Two different SFP schemes were proposed to decide the awake timing. Pseudo branch SFP (PSFP) method [23] is to mimic the branch behavior using a predication register to control the awake timing of each PE, introducing extra *awake* operation. Counter-based SFP (CSFP) [14] method employs a counter to implement an automatic awake scheme, removing *awake* operation. However, both of them need an *unconditional*

*sleep (US)* operation to mimic the unconditional jumpinstruction in traditional GPPs.

According to [23], the key performance overhead of SFP is caused by state switching operations, which is not needed in FP. To further accelerate ITE and NITE structures, we try to propose a novel TFP scheme that implements distributed instruction nullification and parallel TReg overwriting to remove redundant state switching operations.

**III.** **TAG-BASED FULL PREDICATION**

Redundant operations in SFP are mostly caused by the indirect predication transfer between conditional judgement and operations in each path. The state is employed as intermediate information. With regard to hardware support, the state-changing module is often designed as an independent component to sleep or wake up the PEs, which makes *sleep* and *awake* operations unavoidable.

On the other hand, it is interesting to observe that PSFP employs multi-bit tag registers (TReg) which store the target awake points, resulting in the better capability of resolving NITE. TFP is literally inspired by this multi-bit tag idea.

***A. BASIC IDEA OF OUR APPROACH***

In PSFP, state has the same function as predication in FP scheme. The state is changed to *sleep* by additional sleep instruction. When the condition is met, the target instruction *tag* is also loaded into the TReg. The PE stays in *sleep* stateaccording to the difference of the TReg and instruction *tag* field. A *NOP* operation with the target *tag* is inserted, responsible for awaking the PE as well as resetting the TReg.

|  |  |
| --- | --- |
| VOLUME XX, 2017 | 9 |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



With our novel TFP scheme, state is removed and the function of *tag* is redefined. Unlike PSPF where tags are only used to decide the awake point, TFP makes *tag* also function as the state at the same time.

An example of codes containing a 2-level NITE structure is illustrated in Table 1, in which pseudocodes and NITE processing procedure of TFP are shown. T1, T2, and T3 separately represent three different tag values to support 2-level NITE. Each instruction has a *current tag (CT)* field and a *next tag (NT)* field. The bit width of the tag field determines the number of NITE levels that TFP can support. N bits correspond to 2N-1 nested levels. In our cases, we assigned 3-bit to each tag field. Theoretically, it can support up to 7-level NITE. *CT* field indicates whether true or false path the operation belongs to, while *NT* field indicates the next tag that should be loaded into the TReg. It is worth to mention that, in TFP each executed instruction can change TReg according to the *NT* field. If an instruction is nullified, it will not be executed and the value of TReg will not be changed. This ensures the elimination of paired *US* and *awake* operations. To correctly and efficiently set *tags* for instructions in the different path and ITE level, the rule of thumb is:

1. In the same path, instructions share the same *CT*. *NT* is also the same except for the last instruction in the ITE, because of the need for changing the *tag* to normally execute instructions outside this ITE level.
2. Instructions involved in the same ITE level should share the same *CT*. This only contains conditional instructions that compute predications but does not contain the paths of inner ITE structure.
3. True paths should share the same *CT* with conditional instructions to save instruction words so that there is no need to involve two *NT* fields. If the true path is taken, the only thing to do for the TReg is to keep the current value.

To introduce our approach in detail, it is divided into the following three situations for discussion.

*Situation 1*: when *A!=0* and *N>=0*, the true path is takenboth times. TReg stays unchanged and after that, following true path instructions will be normally executed in that the TReg and *CT* field share the same value. And instructions in the false path can be automatically nullified according to the difference between *CT* and TReg, which also means that *NT* field does not affect the TReg value. This situation is relatively simple, and the TReg value remains the same, so it is not shown in Table 1.

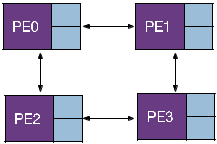
*Situation 2*: when *A!=0* and *N<0*, the inner false path istaken. *NT* field of instruction 3 will be loaded into TReg, and TReg changes from T1 to T3, which is the same as the *CT* field of instructions in the inner false path. Similarly, the following true path is nullified, and the false path will be executed. It should be noted that the last instruction executed in the ITE is responsible for modifying the TReg to the same value as the *CT* field of the first instruction outside of the

VOLUME XX, 2017

ITE. As shown in Table 1, *NT* field of instruction 9 is the same as the *CT* field of instruction 10.

*Situation 3*: when *A==0*, the outer false path is taken.TReg changes from T1 to T2 at the beginning, causing all instructions in the outer true path to be nullified. Unlike PSFP, no additional *awake* operation is needed, the false path will be automatically executed when PEs discover the consistency of *CT* field and the TReg.

Also, because the TReg can load *NT* field immediately after the PE finishes compute predication, we do not need to set counters as CSFP did, and *US* operations in PSFP can be removed just the same way. From Table 1 we can clearly discover that, with TFP scheme, redundant state-changing operations have been removed.



**FIGURE 1. 2x2 example PE array. Each PE has two data registers, and PEs are vertically and horizontally routed to each other.**

***B.*** ***THEORETICAL PERFORMANCE ANALYSIS***

To better illustrate the performance gain, we map this example code onto a 2x2 PE array as shown in Fig. 1. We try to discuss the theoretical performance gain of TFP compared with PSFP and CSFP in terms of three kinds of NITE: long path shallow NITE, short path multi-level and else-free NITE. Some tricks of TFP are also introduced. 2x2 PE array depicted by Fig. 1 is used to map each kind of NITE, in which each PE has two data registers. All of the NITE structures are wrapped into a loop, so that we can use the interval between the start of successive iterations, which is the initiation interval (II), to evaluate the performance [24]. The smaller II corresponds to the shorter execution time, as well as the better performance.

**TABLE 2. Pseudocodes of shallow NITE.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Original C code** | **Assembly code** | **PSFP** | **CSFP** |  | **TFP** |  |  |
|  |  |  |  |  |  |  |  | |
|  | for(i=0; i<50;i++){ | cmp A 0 | cmp A 0, **sleep neq T1** | cmp A 0 | **T1** | **cmp A 0,** | **neq T2** | |
|  | if (A != 0){ | **bneq PC +13** | add N D 1 | **sleep neq C11** | T1 | add N D 1, |  | T1 |
|  | N=D+1; | add N D 1 | cmp N 0, **sleep lt T2** | add N D 1 | **T1** | **cmp N 0,** | **lt** | **T3** |
|  | if(N >= 0) | cmp N 0 | add P1 Q -7 | cmp N 0 | **T1** | add P1 Q -7, | | T1 |
|  | P= (Q-7)\*(Q-8); | **blt PC+5** | add P2 Q -8 | **sleep lt C3** | **T1** | add P2 Q -8, | | T1 |
|  | else | add P1 Q -7 | mul P P1 P2; | add P1 Q -7 | **T1** | mul P P1 P2, | | **T1** |
|  | P= (Q+7)\*(Q+8); | add P2 Q -8 | **uc sleep T3** | add P2 Q -8 | **T3** | add P1 Q 7, | | T3 |
|  | M=N-1; | mul P P1 P2; | **awake T2** | mul P P1 P2; | **T3** | add P2 Q 8, | | T3 |
|  | } | **buc PC+5** | add P1 Q 7 | **uc sleep C3** | **T3** | mul P P1 P2, | | **T1** |
|  | else{ | add P1 Q 7 | add P2 Q 8 | add P1 Q 7 | T1 | add M N -1, | | T1 |
|  | P=1; | add P2 Q 8 | mul P P1 P2; | add P2 Q 8 | **T2** | mov P 1, |  | T2 |
|  | M=N+1; | mul P P1 P2; | **awake T3** | mul P P1 P2; | **T2** | add M N 1, | | **T1** |
|  | } | add M N -1 | add M N -1 | add M N -1 |  |  |  |  |
|  | } | **buc PC+3** | **uc sleep T4** | **uc sleep C1** |  |  |  |  |
|  |  | mov P 1 | **awake T1** | mov P 1 |  |  |  |  |
|  |  | add M N 1 | mov P 1 | add M N 1 |  |  |  |  |
|  |  |  | add M N 1 |  |  |  |  |  |
|  |  |  | **awake T4** |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

9

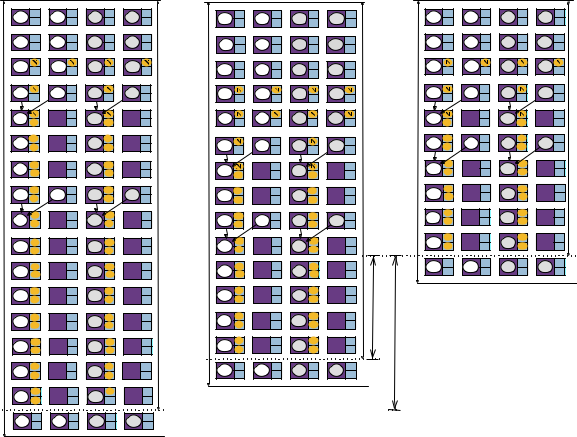
This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



PE0 PE1 PE2 PE3 PE0 PE1 PE2 PE3 PE0 PE1 PE2 PE3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C ycle |  |  |  |  |  |  | C ycle |  |  |  |  |  |  | C ycle |  |  |  |  |  |  |  |
| 1 | **hA** |  | **hA** | **hA** |  | **hA** | 1 | **hA** |  | **hA** | **hA** |  | **hA** | 1 | **hA** |  | **hA** | **hA** |  | **hA** |  |
| 2 | **N** |  | **N** | **N** |  | **N** | 2 | **SC** |  | **SC** | **SC** |  | **SC** | 2 | **N** |  | **N** | **N** |  | **N** |  |
| 3 | **hN** |  | **hN** | **hN** |  | **hN** | 3 | **N** |  | **N** | **N** |  | **N** | 3 | **hN** |  | **hN** | **hN** |  | **hN** |  |
| 4 | **P1** |  | **P2** | **P1** |  | **P2** | 4 | **hN** |  | **hN** | **hN** |  | **hN** | 4 | **P1** |  | **P2** | **P1** |  | **P2** |  |
| 5 | **P** |  |  | **P** |  |  | 5 | **SC** |  | **SC** | **SC** |  | **SC** | 5 | **P** |  |  | **P** |  |  |  |
|  | **P1** |  | **P1** |  |  |  | II= 14 | **P1** |  | **P1** | II= 10 |  |
|  |  |  |  |  |  |  | II= 16 |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 |  | **N** |  |  | **N** |  |  |  |  |  |  |  | 6 |  | **N** |  |  | **N** |  |  |
| **US** |  | **US** |  | 6 |  |  |  |  |  |  | **P1** | **P2** | **P1** | **P2** |  |
| **P** |  | **P** |  | **P1** |  | **P2** | **P1** |  | **P2** | **P** | **P** |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 | **AW** | **N** |  | **AW** | **N** |  | 7 | **P** |  |  | **P** |  |  | 7 | **P** | **N** |  | **P** | **N** |  |  |
|  |  | **P** |  |  | **P** |  | **P1** |  | **P1** |  |  | **P1** |  | **P1** |  |  |
| 8 | **P1** | **N** | **P2** | **P1** | **N** | **P2** | 8 | **US** | **N** |  | **US** | **N** |  | 8 | **M** | **N** |  | **M** | **N** |  |  |
|  |  | **P** |  | **P** |  |  |
|  |  | **P** |  |  | **P** |  |  |  | **P** |  |  | **P** |  |  |  |  |  |  |  |  |  |
| 9 | **P** | **N** |  | **P** | **N** |  | 9 | **P1** | **N** | **P2** | **P1** | **N** | **P2** | 9 | **P** | **M** |  | **P** | **M** |  |  |
| **P1** |  | **P1** |  | **P** | **P** |  | **P** |  | **P** |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  | 10 |  | **N** |  |  | **N** |  | 10 |  | **M** |  |  | **M** |  |  |
| 10 | **AW** | **N** |  | **AW** | **N** |  | **P** |  | **P** |  | **M P** | |  | **M P** | |  |  |
|  |  | **P1** |  | **P1** |  |  |  |  |  |
|  |  | **P** |  |  | **P** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 11 | **M** | **N** |  | **M** | **N** |  | 11 | **M** | **N** |  | **M** | **N** |  | 11 | **hA** |  | **hA** | **hA** |  | **hA** |  |
|  | **P** |  | **P** |  |  | **P** |  | **P** |  |  |  |  |  |  |  |  |  |



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12**US** | **M** | **US** | **M** | 12 | **US** | **M** | **US** | **M** |
|  | **P** |  | **P** |  |  | **P** |  | **P** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | **AW** | **M** | **AW** | **M** | 13 | **P** | **M** | **P** | **M** |  |  |  |  |
| **P** | **P** | **P** | **P** |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
| 14 | **P** | **M** | **P** | **M** | 14 | **M** | **M** | **M** | **M** |  |  |  |  |
|  | **P** | **P** |  | **P** | **P** |  |  |  |  |
| 15 | **M** | **M** | **M** | **M** | 15 | **hA** | **hA** | **hA** | **hA** | 29% |  |  |  |
|  | **P** | **P** |  |  |  |  |  |  |  |  |  |
| 16 | **AW** | **M** | **AW** | **M** |  |  |  |  |  |  | **OP** | O perations in the 1st iteration |  |
| **P** |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 17 | **hA** | **hA** | **hA** |  | **hA** |  |  |  |  | 38% | **OP** | O perations in the 2nd iteration |  |



(a) (b) (c)

**FIGURE 2. Mapping results of a loop with shallow NITE separately based on (a) PSFP, (b) CSFP and (c) TFP.**

1) SHALLOW NITE

The modified pseudocodes of a loop containing a shallow NITE structure are illustrated in Table 2. There are only two levels in the NITE structure, and data dependence, as well as data-level parallelism, are presented among them.

Based on the assembly code, we try to map the loop on the toy PE array mentioned before. The grey and white operations are from two different iterations based on loop unrolling technique [25] since there is no inter-iteration dependence. The mapping results are shown in Fig. 2 in which (a) to (c) separately shows the mapping results of PSFP, CSFP, and TFP. PSFP achieves an II of 16 cycles, while CSFP achieves an II of 14 cycles, both are worse than the II of 10 cycles achieved by TFP. The main consumption of cycles in SFP is introduced by *US* and *awake* operation. Before the false path in each ITE, PSFP should insert an *awake* operation in the case that the false path is taken. At the same time, *US* operation is inserted to ensure the correct nullification of the false path when the

resulting in correspondingly 38% and 29% II reduction compared with PSFP and CSFP.

2) DEEP NITE

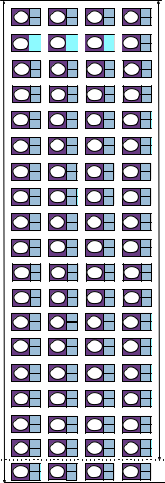
Now we consider the case of a deeper NITE structure, which is shown in Table 3. Different from the example before, in this case, NITE has 4-level NITE, which is quite common in several benchmarks such as *Susan* in Mibench [26]. Instructions in each ITE path are compute-intensive and have high parallelism, thus they can be simultaneously mapped to different PEs at the same cycle and the state of 4 PEs need to be synchronized. To simplify the routing problem and maximum performance, we make each PE compute the conditionals at the same time so that instructions can be nullified correctly and easily. From the mapping results in Fig. 3, it is clear that TFP reflects even more advantages, achieving near 50% performance gain.

**TABLE 3. Pseudocodes of deep NITE.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Original C code** | **Assembly code** | | **PSFP** | **CSFP** |  | **TFP** |  |
|  |  |  | |  |  |  |  | |
|  | for(i=0; i<50;i++){ | cmp h1 0 | | cmp h1 0, **sleep blt T1** | cmp h1 0 | **T1** | **cmp h1 0, lt T2** | |
|  | if (h1 >= 0){ | **blt** | **PC+6** | add A A 1 | **sleep lt C4** | T1 | add A A 1, | T1 |
|  | A+=1; B+=1; | add A A 1 | | … | add A A 1 | T1 | …, | T1 |
|  | C+=1; D+=1;} | … |  | **uc sleep T5** | … | **T2** | **cmp h2 0, lt T3** | |
|  | else{ | **buc PC+27** | | **awake T1** | **uc sleep C25** | T2 | add A A 2, | T1 |
|  | if(h2 >= 0){ | cmp h2 0 | | cmp h2 0, **sleep blt T2** | cmp h2 0 | T2 | …, | T1 |
|  | A+=2; B+=2; | **blt** | **PC+6** | add A A 2 | **sleep lt C4** | **T3** | **cmp h3 0, lt T4** | |
|  | C+=2; D+=2;} | add A A 2 | | … | add A A 2 | T3 | add A A 3, | T1 |
|  | else{ | … |  | **uc sleep T5** | … | T3 | …, | T1 |
|  | if(h3 >= 0){ | **buc PC+19** | | **awake T2** | **uc sleep C17** | **T4** | **cmp h4 0, lt T5** | |
|  | A+=3; B+=3; | cmp h3 0 | | cmp h3 0, **sleep blt T3** | cmp h3 0 | T4 | add A A 4, | T1 |
|  | C+=3; D+=3;} | **blt** | **PC+6** | add A A 3 | **sleep lt C4** | T4 | …, | T1 |
|  | else{ | add A A 3 | | … | add A A 3 | T5 | add A A 5, | T1 |
|  | if(h4 >= 0){ | … |  | **uc sleep T5** | … | T5 | …, | T1 |
|  | A+=4;B+=4; | **buc PC+12** | | **awake T3** | **uc sleep C10** |  |  |  |
|  | C+=4; D+=4;} | cmp h4 0 | | cmp h4 0, **sleep blt T4** | cmp h4 0 |  |  |  |
|  | else{ | **blt PC+6** | | add A A 4 | **sleep lt C4** |  |  |  |
|  | A+=5; B+=5; | add A A 4 | | … | add A A 4 |  |  |  |
|  | C+=5; D+=5;} | … |  | **uc sleep T5** | … |  |  |  |
|  | }}}} | **buc PC+5** | | **awake T4** | **uc sleep C3** |  |  |  |
|  |  | add A A 5 | | add A A 5 | add A A 5 |  |  |  |
|  |  | … |  | … | … |  |  |  |
|  |  |  |  | **awake T5** |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

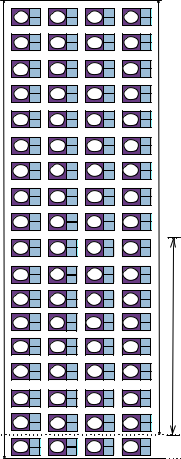
true path is taken. *US* operation requires another *awake* at the end of each ITE to normally execute instructions out of the ITE structure. In total, PSFP should insert 1 *US* and 2 *awake*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | P E0 | P E1 | P E2 | P E3 |
| C ycle |  |  |  |  |
| 1 | **h1** | **h1** | **h1** | **h1** |



2 **A1**  **B1**  **C1**  **D1**

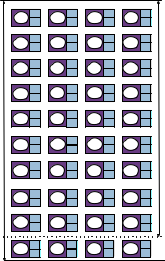
PE0 PE1 PE2 PE3



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C ycle |  |  |  |  |  |
| 1 | **h1** | **h1** | **h1** | **h1** |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | **SC** | **SC** | **SC** | **SC** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | P E0 | P E1 | P E2 | P E3 |  |
| C ycle |  |  |  |  |  |
| 1 | **h1** | **h1** | **h1** | **h1** |  |
| 2 | **A1** | **B1** | **C1** | **D1** |  |
|  |  |  |  |  |



operations for each ITE structure.

Instead of inserting *awake* operation to change back to the *awake* state, CSFP employs a counter for each PE, which isset once the PE needs to switch to *sleep* mode. Once the counter counts down to zero, PE will wake up itself automatically. In this way, all the *awake* operation in SFP can be eliminated. However, because of the need for initializing counters, extra *sleep and set the counter (SC)* operation needs to be inserted, which counteracts part of the benefit brought about by counters. Also, *US* operation cannot be removed. CSFP still has a performance gain in that, for each ITE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | **US** | **US** | **US** |  |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | **AW** | **AW** | **AW** |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 5 | **h2** | **h2** | **h2** |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | **A2** | **B2** | **C2** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | **US** | **US** | **US** |  |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | **AW** | **AW** | **AW** |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | **h3** | **h3** | **h3** |

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | **A3** | **B3** | **C3** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **US** | **US** | **US** |  |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | **AW** | **AW** | **AW** |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 13 | **h4** | **h4** | **h4** |
| 14 | **A4** | **B4** | **C4** |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 | **US** | **US** | **US** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 16 | **AW** | **AW** | **AW** |  |
|  |  |  |  |

**US**

**AW**

**h2**

**D2**

**US**

**AW**

**h3**

**D3**

**US**

**AW**

**h4**

**D4**

**US**

**AW**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | **A1** | **B1** | **C1** | **D1** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 | **US** | **US** | **US** | **US** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| II= 18 5 | **h2** | **h2** | **h2** | **h2** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | **SC** | **SC** | **SC** | **SC** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | **A2** | **B2** | **C2** | **D2** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | **US** | **US** | **US** | **US** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | **h3** | **h3** | **h3** | **h3** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | **SC** | **SC** | **SC** | **SC** |
| 11 | **A3** | **B3** | **C3** | **D3** |
| 12 | **US** | **US** | **US** | **US** |
| 13 | **h4** | **h4** | **h4** | **h4** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 14 | **SC** | **SC** | **SC** | **SC** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 15 | **A4** | **B4** | **C4** | **D4** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 16 | **US** | **US** | **US** | **US** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | **h2** | **h2** | **h2** | **h2** |
| 4 | **A2** | **B2** | **C2** | **D2** |
| II= 17 |  |  |  | II= 9 |
| 5 | **h3** | **h3** | **h3** | **h3** |
| 6 | **A3** | **B3** | **C3** | **D3** |
| 7 | **h4** | **h4** | **h4** | **h4** |
| 8 | **A4** | **B4** | **C4** | **D4** |
| 9 | **A5** | **B5** | **C5** | **D5** |
| 10 | **h1** | **h1** | **h1** | **h1** |



48%

50%

structure, 1 *SC* and 1 *US* operations should be inserted, which is less redundant than PSFP. TFP completely cancels the state transfer operation, and instructions are nullified purely by tags.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 17 | **A5** | **B5** | **C5** | **D5** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 18 | **AW** | **AW** | **AW** | **AW** |  |
|  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 19 | **h1** | **h1** | **h1** | **h1** |  |
|  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 17 | **A5** | **B5** | **C5** | **D5** |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 18 | **h1** | **h1** | **h1** | **h1** |



Benefitting from the tag transfer operation embedded in each instruction, *sleep*, *awake* and *US* can be all removed,

VOLUME XX, 2017

**FIGURE 3. Mapping results of a loop with deep NITE separately based on (a) PSFP, (b) CSFP and (c) TFP.**

9

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



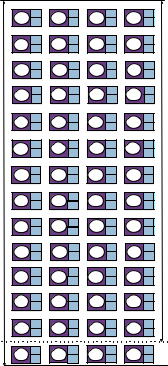
**TABLE 4. Pseudocodes of else-free NITE.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Original C code** | **Assembly** | | **PSFP** | **CSFP** |  | **TFP** |  |  |
|  |  | **code** |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  | |  |  |  |  |  |  |
|  | for(i=0; i<50;i++){ | cmp h1 0 | | cmp h1 0, **sleep blt T1** | cmp h1 0 | **T1** | **cmp h1 0,** | **lt T2** |  |
|  | if (h1 >= 0){ | **blt** | **PC+9** | add A A 1 | **sleep lt C7** | T1 | add A A 1, | T1 |  |
|  | A+=1; B+=1; | add A A 1 | | … | add A A 1 | T1 | …, | T2 |  |
|  | C+=1; D+=1;} | … |  | **awake T1** | … | T2 | add A A 2, | T2 |  |
|  | A+=2; B+=2; | add A A 2 | | add A A 2 | add A A 2 | **T2** | …, | **T1** |  |
|  | C+=2; D+=2; | … |  | … | … | **T1** | **cmp h2 0,** | **lt T2** |  |
|  | if(h2 >= 0){ | cmp h2 0 | | cmp h2 0, **sleep blt T1** | cmp h2 0 | T1 | add A A 3, | T1 |  |
|  | A+=3; B+=3; | **blt** | **PC+5** | add A A 3 | **sleep lt C3** | **T1** | …, | **T2** |  |
|  | C+=3; D+=3;} | add A A 3 | | … | add A A 3 | **T2** | **cmp h3 0,** | **lt T1** |  |
|  | if(h3 >= 0){ | … |  | **awake T1** | … | T2 | add A A 3, | T2 |  |
|  | A+=4; B+=4; | cmp h3 0 | | cmp h3 0, **sleep blt T1** | cmp h3 0 | **T2** | …, | **T1** |  |
|  | C+=4; D+=4;} | **blt** | **PC+5** | add A A 4 | **sleep lt C3** | **T1** | **cmp h4 0,** | **lt T2** |  |
|  | if(h4 >= 0){ | add A A 4 | | … | add A A 4 | **T1** | add A A 4, | T1 |  |
|  | A+=5; B+=5; | … |  | **awake T1** | … | **T1** | …, | **T2** |  |
|  | C+=5; D+=5;} | cmp h4 0 | | cmp h4 0, **sleep blt T1** | cmp h4 0 |  |  |  |  |
|  | } | **blt** | **PC+5** | add A A 5 | **sleep lt C3** |  |  |  |  |
|  |  | add A A 5 | | … | add A A 5 |  |  |  |  |
|  |  | … |  | **awake T1** | … |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

case as else-free ITE. Typically, we arrange the same tag to the conditional and true path instructions, and the instructions out of the ITE still have the same *CT* with the conditional, because they are in the same ITE level. The problem occurs in that, if the conditional is false, the tag should be changed to a value which is different from instructions out of the ITE. After nullifying the true path, tag difference makes PEs continue to incorrectly nullify the common instructions out of the ITE.

One intuitive solution is to mimic PSFP *awake* operation and insert a NOP instruction to reset the TReg, which will inevitably increase the II and do harm to performance. In essence, this NOP operation is only used to complete tag transfer, making PE have the same tag value with the *CT* field of following common instructions. Instead of making TReg reset and consume extra time, we can directly consider the following instructions as a pseudo false path, as shown in the

PE0 PE1 PE2 PE3



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C ycle |  |  |  |  |  |
| 1 | **h1** | **h1** | **h1** | **h1** |  |
| 2 | **A1** | **B1** | **C1** | **D1** |  |
| 3 | **AW** | **AW** | **AW** | **AW** |  |
|  |  |  |  |  |
| 4 | **A2** | **B2** | **C2** | **D2** |  |
| 5 | **h2** | **h2** | **h2** | **h2** |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | **A3** | **B3** | **C3** | **D3** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 7 | **AW** | **AW** | **AW** | **AW** |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | **h3** | **h3** | **h3** | **h3** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | **A4** | **B4** | **C4** | **D4** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | **AW** | **AW** | **AW** | **AW** |  |
|  |  |  |  |  |

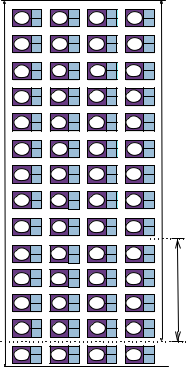
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **h4** | **h4** | **h4** | **h4** |
| 12 | **A5** | **B5** | **C5** | **D5** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 13 | **AW** | **AW** | **AW** | **AW** |  |
|  |  |  |  |  |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 14 | **h1** | **h1** | **h1** | **h1** |

PE0 PE1 PE2 PE3



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C ycle |  |  |  |  |  |
| 1 | **h1** | **h1** | **h1** | **h1** |  |
|  |  |  |  |  |
| 2 | **SC** | **SC** | **SC** | **SC** |  |
| 3 | **A1** | **B1** | **C1** | **D1** |  |
| 4 | **A2** | **B2** | **C2** | **D2** |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| II= 13 5 | **h2** | **h2** | **h2** | **h2** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | **SC** | **SC** | **SC** | **SC** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | **A3** | **B3** | **C3** | **D3** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | **h3** | **h3** | **h3** | **h3** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | **SC** | **SC** | **SC** | **SC** |  |
|  |  |  |  |  |

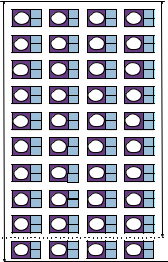
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | **A4** | **B4** | **C4** | **D4** |
| 11 | **h4** | **h4** | **h4** | **h4** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | **SC** | **SC** | **SC** | **SC** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 13 | **A5** | **B5** | **C5** | **D5** |  |
|  |  |  |  |  |
| 14 | **h1** | **h1** | **h1** | **h1** |  |



PE0 PE1 PE2 PE3



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| C ycle |  |  |  |  |  |
| 1 | **h1** | **h1** | **h1** | **h1** |  |
| 2 | **A1** | **B1** | **C1** | **D1** |  |
|  |  |  |  |  |
| 3 | **A2** | **B2** | **C2** | **D2** |  |
| 4 | **h2** | **h2** | **h2** | **h2** |  |
| II= 13 |  |  |  | II= 9 |  |
| 5 | **A3** | **B3** | **C3** | **D3** |  |
| 6 | **h3** | **h3** | **h3** | **h3** |  |
| 7 | **A4** | **B4** | **C4** | **D4** |  |
| 8 | **h4** | **h4** | **h4** | **h4** |  |
| 9 | **A5** | **B5** | **C5** | **D5** |  |
| 10 | **h1** | **h1** | **h1** | **h1** |  |
| 31% |  |  |  |  |  |



last column of Table 4. If h1 is smaller than zero, then the TReg will load T2 and nullify the true path. Because we pretend that the following A+=2 is in the false path, it is arranged with tag T2, and thus will be normally executed. The TReg is reset to T1 afterward to save *CT* bits.

For the example else-free code, TFP arranges *CT* for each ITE in an interleaved style, and no extra NOP instruction needs to be added to reset the TReg. From Fig. 4 we observe that TFP still achieves 31% performance improvement compared with PSFP and CSFP.

**IV. HARDWARE IMPLEMENTATION**

**FIGURE 4. Mapping results of a loop with else-free NITE separately based on (a) PSFP, (b) CSFP and (c) TFP.**

One reason for this relatively large gain percentage is that the example deep NITE has only a short path for each PE. Whereas as far as we observe, long NITE structures that are compute-intensive often do not have a long path for each PE. mainly due to the high parallelism present. The redundant state-changing operations weigh even more percentage with PSFP and CSFP. TFP eliminates all these operations and thus achieves a larger performance gain percentage.

3) ELSE-FREE ITE

The last case is relatively special, and we will do some tricks for TFP to further reduce the II. The example code is shown in Table 4, in which no “else” path exits in each so-called ITE structure, in another word no false path is present and only true paths exist. This is very friendly to PSFP because no extra *US*-*awake* pair is needed to nullify the false path, thus only 1 *awake* operation is needed for each PE. However, CSFPachieves relatively low performance because of additional *SC* operations are still needed to set counters. As shown in Fig. 4, PSFP and CSFP literately have the same cycles, which indicates CSFP has no performance gain at all in this else-free case.

Remembering that in former cases, TFP conceals the paired *US*-*awake* operations by the tag transfer finished by the lastinstruction in each ITE structure. However, it is not the same

VOLUME XX, 2017

***A. HARDWARE SUPPORT FOR TFP***

Unlike PP, which is dedicated to the minimum modification to architecture, TFP needs more additional hardware support. According to the mechanism described in Section III-A, the key component that should be planted in each PE is the TReg, indicating which path the PE needs to nullify. The instruction word fed into each PE is composed of three parts: *CT*, operation code and *NT*. In this design, we separately arrange 3 bits to *CT* and *NT*, while 4 bits to operation code. The overview hardware schematic is shown in Fig. 5.

When the CMP determines the TReg and the *CT* have the same value and the operation code field indicates a conditional, the predication result will be chosen to be delivered to the CE port. If the predication is “true”, the TReg will be disabled, realizing the function of keeping the true path the same *CT* with the conditional. If the predication is “false”, the TReg will be enabled and NT will be loaded, making the TReg share the same *CT* with the false path. An additional MUX logic is inserted between the operation code and the DReg CE port to make the DReg keep unchanged in case that a conditional is fetched. When the operation code indicates a non-conditional operation, the DReg will just load the computing result. Default “1” will be delivered to the CE port of the TReg and NT field will be loaded, endowing common operation with the capability of changing the TReg.

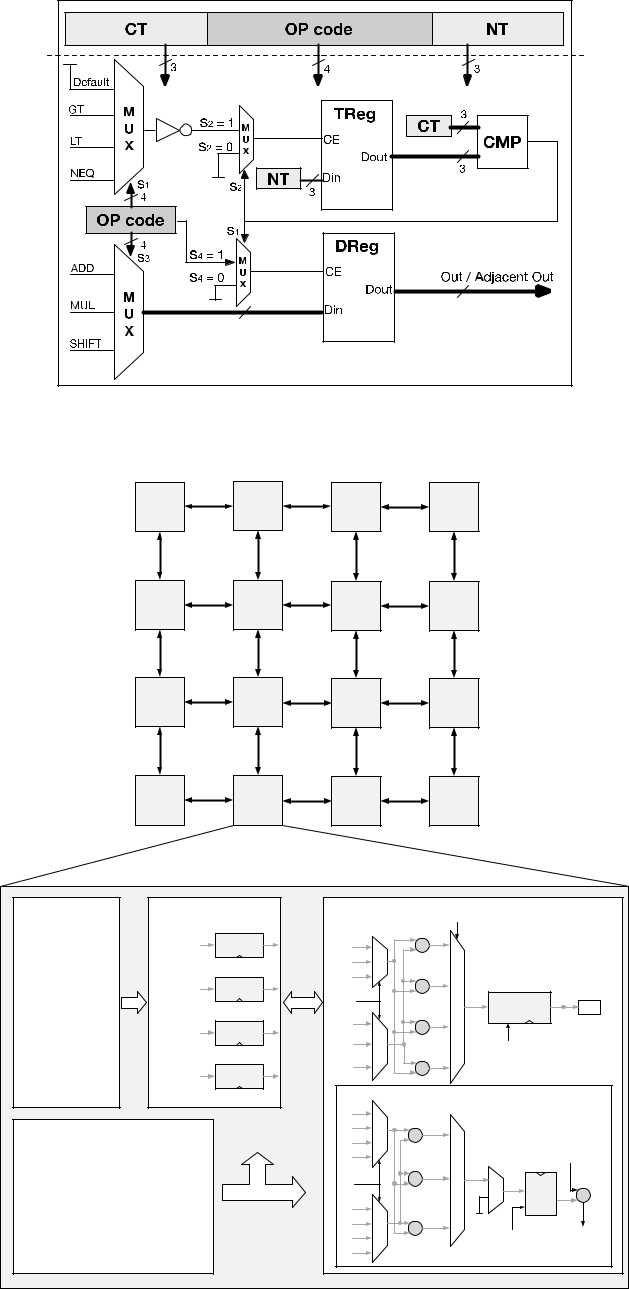
9

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



**FIGURE 5. Hardware support for TFP. CE ports of TReg and DReg are used to realize nullification. MUX S2 is employed to conditional change the TReg.**



|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **PE0** |  | **PE1** | **PE2** |  |  | **PE3** |  |  |  |  |
|  | **PE4** |  | **PE5** | **PE6** |  |  | **PE7** |  |  |  |  |
|  | **PE8** |  | **PE9** | **PE10** | |  | **PE11** |  |  |  |  |
|  | **PE12** |  | **PE13** | **PE14** | |  | **PE15** |  |  |  |  |
|  | **Register** | |  | **Reconfigurable ALU** | | | | |  |  |  |
|  |  |  |  | Config[6:3] (OP) | |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  | PE result |  | **REG1** | R1 |  | **+** |  |  |  |  |  |
|  | Din CLK Dout | | R2 | **M** |  |  |  |  |  |
|  |  |  |  |  | **U** |  |  |  |  |  |  |
|  |  |  |  | R3 | **X** |  |  |  |  |  |  |
|  | Neighbor | **REG2** | | Config[6:3] |  | **-** |  |  |  |  |  |
|  | PE result | Din | Dout |  |  |  |  |  |  |
| **Data RAM** |  | CLK | (OP) |  |  | **M** | **Output REG** | |  |  |
|  |  |  |  |  |  | **U** | Din | Dout | **OUT** |  |
|  | External |  |  | R2 |  | **×** | **X** | CE | CLK |  |  |
|  |  | **REG3** |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  | DATA | Din | Dout |  | **M** |  |  |  |  |  |
|  |  | CLK | R3 |  |  | Config[13] | |  |  |
|  |  |  |  |  | **U** |  |  |  |  |
|  |  |  |  | R4 | **X** |  |  |  |  |  |  |
|  | Local | **REG4** | |  | **/** |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  | DRAM | Din | Dout |  |  |  |  |  |  |  |  |
|  |  | CLK | **TFP hardware unit** | | | | |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  | R1 |  |  |  |  |  |  |  |
|  |  |  |  | R2 | **M** |  |  |  |  |  |  |
|  |  |  |  |  | **>** |  |  |  |  |  |
|  |  |  |  | R3 | **U** |  |  |  |  |  |
|  |  |  |  | **X** |  |  |  |  |  |
|  |  |  |  | R4 |  |  |  |  | Config[9:7] | |  |
|  |  |  |  | Config[6:3] | |  |  |  |  | (CT) |  |
|  |  |  |  | **<** | **M** |  |  |  |  |
|  |  |  |  | (OP) |  | **U** | **M** | **TREG** |  |  |
| **Configuration RAM** | |  |  |  |  |  | **X** | **U** | CE | **==** |  |
|  |  | R1 |  |  |  | **X** | Dout |  |
|  |  |  |  |  |  |  |  |  | Din |  |  |
|  |  |  |  | R2 | **M** |  |  |  |  |  |  |
|  |  |  |  | R3 | **U** | **==** |  | Config[2:0] | | CLK\_en |  |
|  |  |  |  | **X** |  |  |  |
|  |  |  |  |  |  |  | (NT) |  |  |
|  |  |  |  | R4 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |



**Data**  **Control** 

**FIGURE 6. Baseline CGRA architecture. Each PE is equipped with local data and configuration RAM to achieve more control-flow flexibility.**

The nullification procedure is realized by keeping the TReg and the internal result data register (DReg) unchanged, which means CE ports should be given a “0”. The CMP component compares the TReg with *CT*, and if they are tested to be different, the output signal of CMP should be “false” which will choose “0” to deliver to the CE port of the TReg, making the TReg disabled and keep the current value. For the DReg, the same mechanism is adopted to nullify the computing result of common operation. In practice, the CMP will control the enable signal of the ALU and registers through the clock

gating signal, as shown in Fig. 6. So the nullified instructions will not output results as well as they will not be decoded and executed.

***B.*** ***ARCHITECTURE FRAMEWORK***

The baseline architecture is a coarse-grain reconfigurable architecture containing a 4×4 PE array. As the taxonomy proposed in [1], this experimental architecture can be reconfigured temporally in a fine grain style. The overall architecture is shown in Fig. 6. Each PE has a local configuration RAM as well as a local data RAM, of which the sizes are both 512Bytes (128 ×32-bit). The local configuration RAM loads all of the instructions needed by the target loop from off-chip memory at initialization time, and can be sequentially fetched by each PE cycle-by-cycle once the PE array begins to execute the loop. The local data RAM is responsible for storing some constant parameters which will be sequentially fed into ALUs as operands. The reconfigurable ALU currently supports only a few basic integer operations, including addition, subtraction, multiplication, shift, and of course, nested conditional branch. This kind of design has been widely adopted in [27], [28], [29], proven to be efficiently dealing with control flow due to flexibility supplied by the cycle-based reconfiguration.

The routing methodology is in mesh style, allowing vertical and horizontal data transfer to neighbor PEs and stored in four related registers. Because the architecture is designed for verification of TFP, no host processor is introduced, and the input dataflow and instructions are stored in a ROM before the initialization of CGRA.

**V. EXPERIMENTAL RESULTS**

***A.*** ***EXPERIMENTAL METHODOLOGY***

In order to verify the performance improvement of TFP, we design a simple CGRA following the framework mentioned in section IV. The design is written in Verilog, and RTL simulation is conducted with Vivado suite on Xilinx Aritx-7 FPGA to evaluate the performance, then synthesized using Design Compiler (DC) in TSMC 40nm technology to evaluate power and area. The simulation clock frequency is set to 50 MHz. At the same time, we also realize the PSFP and CSFP on the same baseline architecture to make a comparison. According to the previous work [14] and [23], we extract some relatively control-intensive loops with typical NITE structures from SPEC CPU2006 [7] and Mibench [26] to be the target test cases, as shown in Table 5. The binary instructions are generated in a semi-automatic way, which is obtained by replacing the assembly codes with a script. According to the different complexity of the configuration information, PE can be dynamically reconfigured within 10-15 cycles. We manually map the test loops on the CGRAs separately, and the CGRAs are designed after the three different NITE processing schemes. The performance simulation results are shown in Fig. 7, while the results normalized to PSFP are shown in Fig. 8.

|  |  |
| --- | --- |
| VOLUME XX, 2017 | 9 |

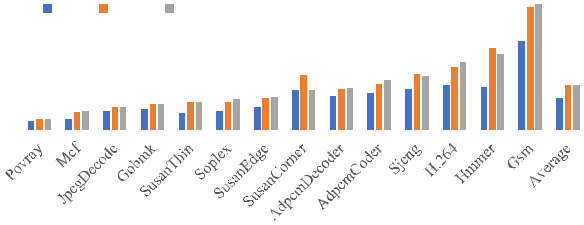
This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



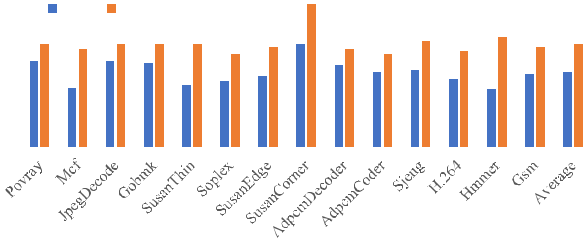
**TABLE 5. Benchmarks used for evaluation.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Benchmark Suite** | | **Program** | **Domain** |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Susan\_edge |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Susan\_corner | Image Processing |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Susan\_thin |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  | Mibench | Jpeg\_decode | Image Compression |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Adpcm\_code | Audio Compression |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  | Adpcm\_decode |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Gsm | Mobile Telecom |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | H.264 | Video Compression |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Hmmer | Gene Search |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Sjeng | Pattern Recognition |  |  |
|  |  |  |  |  |  |  |  |
|  |  | SPEC CPU2006 | | Soplex | Linear Program |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Gobmk | Artificial Intelligence |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Mcf | Optimization |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  | Povray | Computer Visualization |  |  |
|  |  |  |  |  |  |  |  |
|  |  | 150 |  |  |  |  |  |
| Interval | | TFPCSFP | PSFP |  |  |  |
|  |  |  |  |
| 100 |  |  |  |  |  |
| Initiation | |  |  |  |  |  |
| 50 |  |  |  |  |  |
|  |  |  |  |  |
| 0 |  |  |  |  |  |
|  |  |  |  |  |



**FIGURE 7. II of TFP, CSFP, and PSFP. TFP achieves the best performance.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1.5 |  |  |  |
| Normalized | TFP | CSFP |  |
|  |  |
| 0.5 |  |  |  |
|  | 1 |  |  |  |
|  |  |  |  |
|  | 0 |  |  |  |
|  |  |  |  |
|  |  |  |  |



**FIGURE 8. II of TFP and CSFP normalized to PSFP.**

***B.*** ***COMPARISON OF PERFORMANCE***

Generally speaking, TFP achieves the smallest II among the three strategies and yielding to a 32% performance gain over PSFP according to the normalized results shown in Fig. 8. Fig.7 presents the II of each strategy, from which we can observe that multiple kinds of path length are considered.

It can be observed that CSFP only achieves a comparable performance with PSFP in most of the cases. One reason is the counteracting effect brought about by the extra *SC* operation. The other reason is that, although CSFP could remove the *awake* operation at the end of each ITE, in some of the deepNITE structures multiple conditionals can share the same *awake* operation in PSFP, which further diminishes theperformance gain obtained by CSFP. The essence of awake-share phenomenon is that conditionals in the same NITE

VOLUME XX, 2017

structure could branch the same target address, which is the end of the NITE. Because CSFP should introduce an extra *SC* operation whenever coming across with a conditional, and if most of the conditional could share the same *awake* operation such as *Susan\_corner*, CSFP may result in a worse performance than PSFP. In common cases, CSFP achieves nearly 2 cycles reduce according to our experimental results.

Compared with CSFP, TFP achieves larger performance gain over PSFP. The main reason for the improvement is the elimination of all the redundant operations. The gain percentage can reach to nearly 50% when a high branch percentage is presented, such as *Susan\_thin* and *Hmmer*. TFP obtains nearly no improvement with *Susan\_corner* over PSFP only because all of the ITE structures involved in the NITE are else-free thus no redundant *US* and *awake* operation introduced due to false paths. Only one *awake* operation is added at the end of the NITE, shared by all of the inner ITE structures. In these extreme cases, the only improvement can be achieved is to remove the last *awake* operation. TFP obtains 32% performance gain over CSFP on average, indicating *Susan\_corner* is only a special case.

1. ***COMPARISON OF POWER CONSUMPTION AND AREA OVERHEAD***

Although performance improvement is the main purpose of this work, power consumption and area overhead are equally crucial to be considered in our experiments. The power result of the 4x4 PE array is 268.5mW, and the power usage of modules is listed in Table 6. Besides, the power consumption of CSFP and PSFP are also given for comparison. Since the actual execution path of the extracted loop is unknown, that is, it is not known whether the true path or the false path is taken, the length of the invalid path is also uncertain. To verify as comprehensive as possible, we show the results of the short invalid path and the long invalid path in Fig. 9 and Fig. 10 respectively.

The results show that the power reduction of CSFP relative to PSFP is 2.5% when the short path is invalid, and 8.8% when the long path is invalid. And our TFP method has almost the same result as PSFP due to similar strategies, obtaining an average 5% additional power consumption compared to CSFP. The reason may be that CSFP is designed to use counters instead to reduce the hardware resources for state monitoring in PSFP, the corresponding power consumption is also saved. In general, the difference is mainly determined by the length of the invalid path. The longer it is, the more obvious the power consumption savings of CSFP relative to PSFP and TFP.

**TABLE 6. The power usage of modules.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Power Group** | **Interval Power** | **Switching Power** | **Leakage Power** |
|  | Register | 263.52 | 0.17 | 1.12 |
|  | Combinational | 0.88 | 2.01 | 0.8 |
|  | Total (mW) | 264.4 | 2.18 | 1.92 |
|  |  |  |  |  |

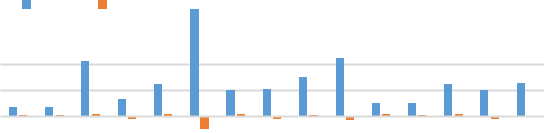
9

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



|  |  |  |  |
| --- | --- | --- | --- |
| 10.00% |  |  |  |
| CSFP | TFP |  |
| 8.00% |  |
|  |  |  |
| 6.00% |  |  |  |
|  |  |  |



4.00%

2.00%

0.00%

-2.00%

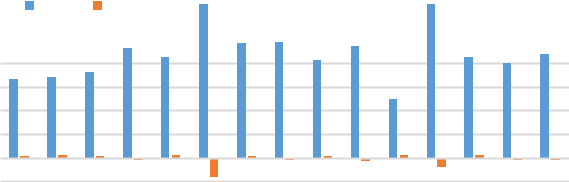


**TABLE 7. The area overhead of PE.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Power Group** | **TFP** | **CSFP** | **PSFP** |
|  | Area (μm2) | 8,927 | 8,965 | 8,925 |
|  | Ratio | - | 0.43% | -0.02% |

**FIGURE 9. Power reduction of CSFP and TFP relative to PSFP when the short path is invalid.**

|  |  |  |  |
| --- | --- | --- | --- |
| 14.00% |  |  |  |
| CSFP | TFP |  |
| 12.00% |  |
|  |  |  |
| 10.00% |  |  |  |
|  |  |  |



8.00%

6.00%

4.00%

2.00%

0.00%

-2.00%

-4.00%



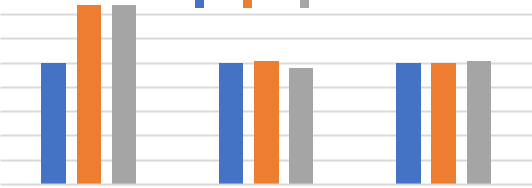
**FIGURE 10. Power reduction of CSFP and TFP relative to PSFP when the long path is invalid.**

Since our experimental architecture is relatively simple and not suitable for a fair comparison of the overall area, we compare the additional hardware added in each PE to reflect the area difference between TFP and other methods. Table 7 shows the synthesis results of the area overhead. Compared to PSFP, the proposed TFP only adds negligible hardware. The additional area overhead mainly comes from the extended instruction tag field (from 3 bits to 6 bits) and the corresponding tag register. For CSFP, a sleep counter (8 bits) is needed to automatically wake up the PE, but the instruction tag field (3 bits) and the corresponding TReg in PSFP and TFP can be removed. Resulting in only an insignificant 0.43% extra area overhead.

At last, we show a comprehensive comparison of performance (II), power consumption, and area overhead in Fig. 11. It can be seen that our purpose is to accelerate the processing of NITE. For CGRA, which is known for its energy-efficiency architecture, performance improvement is more meaningful than power and area savings.

1.6

TFP PSFP CSFP



1.4

1.2

1

0.8

0.6

0.4

0.2

0

Performance (II) Power Area

**FIGURE 11. Comprehensive comparison (normalized to TFP).**

VOLUME XX, 2017

**VI. CONCLUSION**

This paper proposes a tag-based full predication strategy to accelerate NITE structures. The TFP strategy is based on full predication and inspired by the tag idea introduced by PSFP. TFP inherits the advantage of full predication, which has no register pressure and at the same time, capable of dealing with NITE. Furthermore, TFP eliminates the redundant operations introduced by PSFP and CSFP with the idea that using tags to directly transfer predication information to operations in each path, obtaining 32% performance gain over PSFP and CSFP without sacrificing much extra power and area.

**REFERENCES**

1. M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: overview and classification," in *Proc. SAMOS,* 2016, pp. 235-244.
2. M. Balasubramanian, S. Dave, A. Shrivastava, and R. Jeyapaul, "LASER: A hardware/software approach to accelerate complicated loops on CGRAs," in *Proc. DATE,* 2018, pp. 1069-1074.
3. X. Chen, A. Minwegen, S. B. Hussain, A. Chattopadhyay, G. Ascheid and R. Leupers, "Flexible, efficient multimode MIMO detection by using reconfigurable ASIP," *IEEE TVLSI*, vol. 23, no. 10, pp. 2173-2186, Oct. 2015.
4. M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao and Y. Nakashima, "A CGRA-Based Approach for Accelerating Convolutional Neural Networks," in *IEEE MCSoC*, Turin, Italy, 2015, pp. 73-80.
5. Y. Yuttakonkit and Y. Nakashima, "Performance Comparison of CGRA and Mobile GPU for Light-Field Image Processing," in 2016 4th Int. Symp. Comput. Network. (CANDAR), Hiroshima, Japan, 2016, pp. 174-180.
6. B. Liu, H. Qin, Y. Gong, W. Ge, M. Xia and L. Shi, "EERA-ASR: An Energy-Efficient Reconfigurable Architecture for Automatic Speech Recognition With Hybrid DNN and Approximate Computing," *IEEE* *Access*, vol. 6, pp. 52227-52237, 2018.
7. J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH* *Comput. Archit. News,* vol. 34, no. 4, pp. 1-17, 2006.
8. M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-aware loop mapping on CGRAs," in *DAC*, 2014, pp. 1-6.
9. S. Yin, P. Zhou, L. Liu, and S. Wei, "Acceleration of nested conditionals on CGRAs via trigger scheme," in *ICCAD*, 2015, pp. 597-604.
10. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. SJCC*, Atlantic City, NJ, USA, 1967.
11. K. Han, J. Ahn, and K. Choi, "Power-efficient predication techniques for acceleration of control flow execution on CGRA," *ACM Trans. Archit.* *Code Optim.,* vol. 10, no. 2, pp. 1-25, May 2013.
12. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *MICRO*, Portland, OR, USA, 1992, pp. 45-54.
13. K. Han, J. K. Paek, and K. Choi, "Acceleration of control flow on CGRA using advanced predicated execution," in *FPT*, Beijing, China, 2010, pp. 429-432.
14. K. Han, S. Park, and K. Choi, "State-based full predication for low power coarse-grained reconfigurable architecture," in *DATE*, Dresden, Germany, 2012, pp. 1367-1372.

9

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3001220, IEEE Access



1. R. A. Bittner, P. M. Athanas, and M. D. Musgrove, "Colt: An experiment in wormhole run-time reconfiguration," in *Proc. SPIE* *–* *Int. Soc. Opt.* *Eng.,* vol. 2914, pp. 187-194, 1996.
2. B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *FPL,* Lisbon, Portugal, 2003,
   1. 61-70.
3. K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, C. R. Moore, "TRIPS:A polymorphous architecture for exploiting ILP, TLP, and DLP," *ACM Trans. Archit. Code Optim.,* vol. 1, no. 1, pp. 62-93, 2004.
4. S. Yin, D. Liu, Y. Peng, L. Liu and S. Wei, "Improving Nested Loop Pipelining on Coarse-Grained Reconfigurable Architectures," *IEEE* *TVLSI*, vol. 24, no. 2, pp. 507-520, Feb. 2016.
5. S. Yin, X. Lin, L. Liu and S. Wei, "Exploiting Parallelism of Imperfect Nested Loops on Coarse-Grained Reconfigurable Architectures," *IEEE* *TPDS*, vol. 27, no. 11, pp. 3199-3213, 1 Nov. 2016.
6. D. Liu, S. Yin, L. Liu, and S. Wei, "Polyhedral model-based mapping optimization of loop nests for cgras," In *DAC*, Austin, TX, USA, 2013,
   1. 1–8.
7. S. Das, K. J. M. Martin, P. Coussy, D. Rossi, and L. Benini, "Efficient mapping of CDFG onto coarse-grained reconfigurable array architectures," In *ASP-DAC*, Chiba, Japan, 2017, pp. 127–132.
8. K. Han, K. Choi, and J. Lee, "Compiling control-intensive loops for CGRAs with state-based full predication," in *DATE*, Grenoble, France, 2013, pp. 1579-1582.
9. 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. Euromicro Symp. Digital Syst. Design.* *Archit. Methods Tools*, 2002, pp. 148-155.
10. B. R. Rau, “Iterative modulo scheduling: an algorithm for software pipelining loops,” in *MICRO*, San Jose, CA, USA, 1994, pp. 63–74.
11. R. Gnanaolivu, T. Norvell, and R. Venkatesan, “Modulo scheduling for loops with conditional branches onto coarse-grained reconfigurable architectures,” in *NECEC*, St. John's, NL, Canada, 2010.
12. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE WWC-4*, Austin, TX, USA, 2001, pp. 3-14.
13. V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP-A Self-reconfigurable data processing architecture," *J, Supercomputing,* vol. 26, no. 2, pp. 167-184, Sep. 2003.
14. R. David, D. Chillet, S. Pillement, and O. Sentieys, “DART: A dynamically reconfigurable architecture dealing with future mobile telecommunications constraints,” in *IPDPS*, Fort Lauderdale, FL, USA, 2002.
15. G. Cichon, P. Robelly, H. Seidel, E. Matúš, M. Bronzel, and G. Fettweis, "Synchronous Transfer Architecture (STA)," in *Int. Workshop* *Embedded Comp. Syst.*, Berlin, Germany, 2004, pp. 343-352.

VOLUME XX, 2017

**Jiang Sha** received the B.S. and M.S. degree inElectronic Engineering from Shenyang University of Technology in 2011 and 2014. He is currently a Ph.D. candidate in School of Electronic Science and Engineering, Southeast University. His research interests mainly include system workload modeling, reconfigurable computing and SoC design technology.



**Wenbo Song** received his B.S. degree fromXidian University, Xi’an, China, in 2016, major in electronic engineering. He is currently pursuing the M.S. degree in Southeast University. His research interests include coarse-grain reconfigurable architecture and general-purpose architecture design.



**Yu Gong** received the B.S. degree in mathematicsfrom Southeast University in 2013 and the M.S. degree in Integrated Circuits from Southeast University in 2016. He is now pursuing his Ph.D. degree in Electronic Science and Engineering form Southeast University since 2016. His research interests including approximate computing,



reconfigurable computing, deep learning accelerator and related VLSI design.



**Yingying Zhao** received her B.S degree fromHefei University of Technology, Hefei, China, in 2017, major in Microelectronics Science and Engineering. She is currently pursuing the M.S. degree in Southeast University. Her research interests include coarse-grain reconfigurable architecture and general-purpose architecture design.

9

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/.