A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications

LEIBO LIU, JIANFENG ZHU, ZHAOSHI LI, and YANAN LU, Institute of Microelectronics, Tsinghua University, Beijing, China
YANGDONG DENG, School of Software, Tsinghua University, Beijing, China
JIE HAN, Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada
SHOUYI YIN and SHAOJUN WEI, Institute of Microelectronics, Tsinghua University, Beijing, China

As general-purpose processors have hit the power wall and chip fabrication cost escalates alarmingly, coarse-grained reconfigurable architectures (CGRAs) are attracting increasing interest from both academia and industry, because they offer the performance and energy efficiency of hardware with the flexibility of software. However, CGRAs are not yet mature in terms of programmability, productivity, and adaptability. This article reviews the architecture and design of CGRAs thoroughly for the purpose of exploiting their full potential. First, a novel multidimensional taxonomy is proposed. Second, major challenges and the corresponding state-of-the-art techniques are surveyed and analyzed. Finally, the future development is discussed.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; • **Hardware** → **Reconfigurable logic and FPGAs**; • **Theory of computation** → *Models of computation*;

Additional Key Words and Phrases: CGRA, spatial architecture, reconfigurable computing, dataflow, scheduling

ACM Reference format:
Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* 52, 6, Article 118 (October 2019), 39 pages.
https://doi.org/10.1145/3357375

1 INTRODUCTION

In recent years, with the rapid developments in society and technology, the demand for performance, energy efficiency and flexibility has grown continuously in the field of computing chips. A

This work is supported in part by the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No. 2018ZX01028201), and in part by the National Natural Science Foundation of China (Grants No. 61672317 and No.61834002), and in part by National Science and Technology Major Project of the Ministry of Science and Technology of China # 2016ZX01012101.
Authors' addresses: L. Liu, J. Zhu (corresponding author), Z. Li, Y. Lu, S. Yin, and S. Wei, Institute of Microelectronics, Tsinghua University, Haidian District, Beijing, 100084, China; emails: {liulb, zhujianfeng, lizhaoshi}@tsinghua.edu.cn, lya13@mails.tsinghua.edu.cn, {yinsy, wsj}@tsinghua.edu.cn; Y. Deng, Institute of Microelectronics, Tsinghua University, Haidian District, Beijing, 100084, China; email: dengyd@tsinghua.edu.cn; J. Han, Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 1H9, Canada; email: jhan8@ualberta.ca.

0360-0300/2019/10-ART118 $15.00
https://doi.org/10.1145/3357375

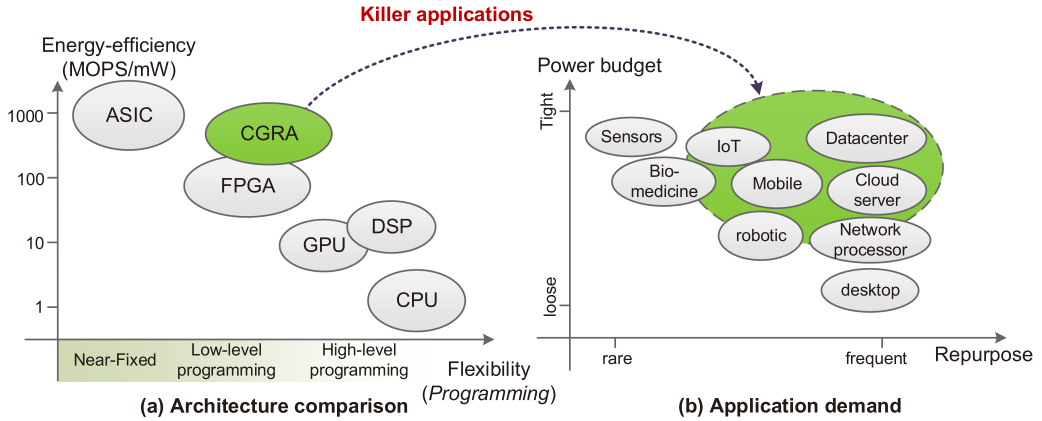

Fig. 1. Architecture comparison in terms of flexibility, performance, and energy efficiency.

large amount of popular or emerging applications (e.g., neural networks and bioinformatics) have led to unprecedented demand for computing power. Traditionally, computing fabrics have taken advantage of integrated circuit technology advances as a major measure to improve the computing power in the past decades. However, this measure becomes invalidate as Moore's Law and Dennard scaling are slowing down or even terminating. A well-known problem of the *power wall* arises: the power budget of integrated circuits becomes tighter in many applications, and worse still, the energy efficiency has a diminishing return with new technologies, resulting in a limitation on the feasible computing power [1–4]. Consequently, computer architecture designers have to shift their focus from performance to energy efficiency. However, flexibility has also become an important consideration in circuit design. As software is evolving rapidly with emerging applications, user needs, and scientific progress, the hardware that cannot adapt to software (e.g., application-specific integrated circuits, ASICs) will suffer from a short lifecycle and high nonrecurring engineering (NRE) cost. The situation becomes even worse for expensive new circuit technology. Overall, both energy efficiency and flexibility have become the main criteria for computing fabrics [5, 6].

Nevertheless, it is challenging for the mainstream computing fabric to meet this new demand. ASICs have extremely low flexibility, whereas Von Neumann processors, such as general-purpose processors (GPPs), graphics processing units (GPUs), and digital signal processors (DSPs), have extremely low energy efficiency. Field-programmable gate arrays (FPGAs) appear promising to some extent, but this architecture is more challenging to program than central processing units (CPUs) and is less energy-efficient than ASICs. Therefore, none of these options can achieve a satisfactory balance between the two criteria, which raises an urgent demand for novel architecture, as evidenced by industry's adoption of domain-specific accelerators in many important areas, such as machine learning and big data.

Coarse-grained reconfigurable architectures (CGRAs) are a natural coarse-grained implementation of the concept of reconfigurable computing proposed in 1960s [7]. This architecture originated in the 1990s [8, 9] and has been developing rapidly since the 2000s [10–13]. CGRAs continue to attract increasing interest because they possess near-ASIC energy efficiency and performance with post-fabrication software-like programmability [14–18]. The comprehensive comparison provided in Figure 1 compares CGRAs with ASICs, FPGAs, DSPs, GPUs and CPUs in terms of the energy efficiency, flexibility and performance [14, 18]. In academia, many researchers consider CGRAs a strong competitor for mainstream computing fabrics, as evidenced by the substantial works published at leading conferences [19, 20] and the important foundation supports by, e.g., the Defense Advanced Research Projects Agency (DARPA) [21]. The goal of the DARPA ERI (electronics

resurgence initiative) is software-defined hardware (SDH) that enables near-ASIC performance (within 10×) without sacrificing programmability for data-intensive algorithms. The DARPA has already thought of CGRAs as a potential direction for the SDH project. In industry also, CGRAs have gained increasing acceptance. For instance, Samsung integrated a CGRA accelerator into its 8K high-definition television (HDTV) and Exynos System-on-Chips (SoC) [22, 23]. PACT Inc. has had CGRA intellectual property (IP) cores applied to the satellite payload of Astrium [24]. Intel started a project to incorporate CGRAs into its Xeon processor in 2016 [25]. Many other companies also have related plans, prototypes or products, such as the DRP [26] and DAPDNA [27]. Despite these commercial applications, CGRAs have much more popularity in academia than in industry for the reason that their technology is still immature.

First, a definition of CGRAs is presented as the foundation of this work. We argue that *a CGRA is a computing fabric that has all the following characteristics*:

(1) ***Domain-Specific Flexibility***. CGRAs have a degree of post-fabrication flexibility between general purpose and fixed function. Their hardware can be defined by software at runtime, but their processing elements (PEs) are not as powerful as those of GPPs, and their interconnections are not as complex as those of FPGAs. This architecture is just flexible enough for specific domains. Although CGRAs are mostly reconfigurable at the coarse-grained level (as indicated by their name), they actually differ across applications. For instance, a CGRA for cryptographic algorithms might contain fine-grained components.

As opposed to general-purpose flexibility (e.g., FPGAs and GPPs), domain-specific flexibility tailors the hardware to target applications and keeps redundant resources minimized. As a result, for the target domain, CGRAs are typically 1–2 orders of magnitude more energy-efficient than FPGAs and more than 2–3 orders of magnitude more energy-efficient than GPPs [28–30]. For general applications, the advantage of CGRAs typically shrinks [16]. Therefore, domain-specific flexibility proves to be one of the critical reasons for CGRAs' balance between energy efficiency and flexibility.

(2) ***Combining Spatial and Temporal Computation.*** In spatial computation, CGRAs take advantage of parallel computing resources and data transferring channels to perform computation. In temporal computation, CGRAs take advantage of time-multiplexing resources to perform computation. Therefore, the mapping of a CGRA is actually equivalent to identifying the spatial and temporal coordinates of every node and arc in the control/data flow graph (CDFG). Compilers are responsible for making this arrangement.

The combination of spatial and temporal computation provides a more flexible and powerful implementation framework for applications. Relative to architectures that enable only temporal computation (e.g., GPPs), CGRAs can obviate costly deep pipelines and the centralized communication overhead. Relative to architectures that enable only spatial computation (e.g., conventional FPGAs, programmable array logic (PAL) architectures, and ASICs), CGRAs can improve area efficiency. Therefore, combining spatial and temporal computation is one of the critical reasons for CGRAs' high energy/area efficiency without reducing flexibility.

(3) ***Configuration- or Data-driven Execution.*** As opposed to sequential processors whose operations are driven by control flow (statically determined by compilers), CGRAs have their operations driven mostly by configuration flow or data flow. The configuration of CGRAs defines PE operations in addition to interconnections. All the PEs defined by one configuration execute in lockstep and under the same flow of control (thread). Although the configurations are also driven by control flow mostly, the operations in each configuration are in parallel or pipelined, which exploits compiler-directed parallelism. More importantly, configuration-driven CGRAs can exploit efficient explicit dataflow via interconnections, which is not supported in conventional instruction sets. A data-driven CGRA is an implementation of an explicit dataflow machine [31],

Table 1. Comparisons Between CGRAs and Important Computing Fabrics

Architecture	Flexibility	Computation form		Execution mechanism			
		Temporal	Spatial	Reconfiguration time(4)	Configuration-driven	Dataflow-driven	Instruction-driven
CGRA	Domain	√	√	ns-μs	√	√	×
FPGA	General	×(1)	√	ms-s	√	√	×
ASIC	Fixed	×(2)	√	×	×(3)	√	×
In-Order Processor/ VLIW	General	√	×	ns	×	×(5)	√
Out-of-Order Processor	General	√	×	ns	×	√	√
Multicore	General	√	√	ns	×	×(5)	√

Notes: (1) FPGAs can perform temporal computation, but it is not practical considering the overhead and effectiveness;
(2) ASICs support hardware resource sharing as temporal computation to some extent;
(3) ASICs do not support reconfiguration, but there might exist configuration codes;
(4) The reconfiguration time is not accurate. The data come from recent works with technologies below 90nm [21].
(5) Dataflow mechanisms can be supported in software at the task/thread level, e.g., data-triggered multi-threading [32, 33]. A thread of computation is initiated when its input data are ready, continuation is ready, or an address is changed.

which abandons control flow execution completely. With all operations in one configuration as candidates, any one that has its operands prepared will be executed. Here, data-driven CGRAs follow an explicit producer-consumer data-dependent relationship.

Compared to control-flow-driven or instruction-driven execution, as occurs in, e.g., multicore processors, configuration-/data-driven execution can avoid over-serialized execution of PEs, exploit fine-grained parallelism, and provide efficient synchronization among PEs. This execution style further supports explicit data communication, which can minimize the energy overhead of data movement. Therefore, configuration-/data-driven execution is one of the critical reasons for CGRAs' high performance and energy efficiency.

In summary, CGRAs are defined as domain-specific flexible hardware, on which the computation is performed both spatially and temporally and the execution is driven by configuration flow or data flow. This definition, a superset of previous definitions that are either controversial or unilateral [34–36], avoids ambiguity. As reported in Table 1, the three characteristics distinguish CGRAs from the other computing architectures. FPGAs are similar to CGRAs in terms of flexible spatial computing (reconfigurable computing). However, FPGAs are a fine-grained general-purpose flexible architecture and rarely support temporal computation because their reconfiguration process is much slower than that of CGRAs (ns vs. ms to s) and thus the reconfiguration is challenging to be pipelined with kernel-level computation. Although some commercial FPGAs support runtime reconfiguration (RTR), the broad usefulness of RTR remains an open question [35]. Multicore processors are similar to CGRAs in terms of structure, including multidimensional PE arrays and message-passing interconnection. However, their processing units are individual sequential cores driven by control flow or instructions. Moreover, this definition identifies the relationship between CGRAs and other classic concepts in computer architecture. CGRAs are a subset of spatial architecture because they essentially support spatial computation. Some overlap exists between CGRAs and dataflow architectures because some CGRA implementations adopt the dataflow mechanism.

The above characteristics also identify the key reasons for the advantages of CGRAs over other architectures. However, since CGRAs are still immature in terms of *programmability, productivity*, and *adaptability*, their commercial applications are limited today. First, CGRAs are challenging to program in high-level languages with desirable efficiency, and their automatic compiling is difficult

to surmount. Second, CGRAs vary greatly from one design to another, making them costly to incorporate in real systems. Third, CGRAs applications are strictly limited to purely computation- or data-intensive kernels, and their performance degrades severely in cases involving any irregular codes or complex control flow. Since these problems are not only related to microarchitecture, CGRAs must be thoroughly reviewed at all system abstraction levels to determine the root causes for these fatal problems.

In recent years, many surveys and books about reconfigurable computing have been published [15, 34–45]. Some of them concentrate on FPGAs and pay less attention to CGRAs [35, 37, 40, 42–45]. The literature focused on CGRAs has covered a lot of topics, including classifications, architectures, applications, compilation methods and tools, and challenges. However, several important problems are missing or require further discussion. First, most classifications are based on low-level characteristics or trivial details, which cannot reveal the inherent characteristics of CGRAs. Second, challenges have not been analyzed comprehensively. Third, the future development of architecture and applications has not yet been discussed in-depth. Therefore, a comprehensive survey is necessary.

In this survey, a comprehensive introduction to the architecture and design of CGRAs is presented in terms of the aspects of classification, challenges and applications. Relative to previous works [15, 36–39, 41], the main contributions are as follows:

(1) A novel CGRA classification method from the abstraction levels of programming, computation and execution.
(2) A complete top-down analysis of the challenges that are currently encountered by CGRAs and corresponding state-of-the-art measures and prospective solutions.
(3) An in-depth discussion of CGRAs' architecture and application development.

The rest of this article is organized as follows. Section 2 proposes a multidimensional taxonomy. Section 3 analyzes challenges in terms of the aforementioned aspects. Section 4 surveys state-of-the-art techniques and prospective solutions. Section 5 discusses the future trends of architecture and applications. Section 6 concludes the article.

2 A TAXONOMY OF CGRAS

As defined above, CGRAs are a class of reconfigurable computing processors that are specialized for coarse-grained kernels in applications. Their architecture design has been controversial till now. Classification opens a door to in-depth understanding of their architecture. This section reviews previous classification methods and then presents a novel multidimensional taxonomy.

2.1 Review of Previous Classification Methods

This part reviews previous classification methods for reconfigurable computing. Hartenstein et al. [34] introduced CGRAs in three major categories in terms of **interconnections and topology**: mesh-based architecture (the most common one is a 2D array with horizontal and vertical connections), linear array architecture (aiming at mapping pipelines), and cross-bar-based architecture (most powerful for routing with most overheads). Compton and Hauck [44] classified reconfigurable systems according to **the degree of coupling between the reconfigurable fabric and the host processor**. Four classifications of reconfigurable fabric were proposed: (1) stand-alone, (2) attached to the host processor without shared cache, (3) as a coprocessor with shared cache, and (4) as a data path in the host processor. Todman et al. [45] added an additional classification: (5) the processor is embedded in a reconfigurable fabric as a soft core or hard core. Chattopadhyay et al. [41] advocated a simplified three-classification system: (1) add-on reconfigurability, which incorporates reconfigurable fabric into the baseline GPP system; (2) add-on processing, which

allocates a part of the reconfigurable fabric to form processors; and (3) custom processing and reconfigurability, which combines customizable processors and custom-designed reconfigurable fabrics. Zain-ul-Abdin and Svensson [39] presented a classification based on **PE function and coupling** that divides CGRA into four categories: (1) hybrid architecture, in which the PE array is coupled with an ordinary processor; (2) array of function units, in which the control scheme is based on special PEs or modules; (3) array of processors, in which every PE is a simple processor with reconfigurable interconnections; and (4) array of soft processors, in which programmable logics are aggregated into multiple soft cores. Chattopadhyay et al. [41] also proposed another classification based on **application domains**: (1) general purpose, such as prototyping and demonstration, and (2) specific domains, such as digital signal processing, high-performance computing, multimedia processing and cryptography. Dehon et al. [46] and Wijtvliet et al. [36] adopted a more sophisticated classification method based on **design patterns** or **property categories**. A very large design space could be built on the basis of multiple properties or design patterns, such that any CGRA can be located in this design space. Dehon et al. [46] adopted the design patterns of area-time trade-off, parallelism expression, processor-FPGA integration, runtime partial reconfiguration, communication expression, synchronization, and so on. Wijtvliet et al. [36] adopted a simplified classification based on four properties: structure, control, integration and tools. However, since the adopted patterns and properties are often nonorthogonal and incomplete (such as system integration and tools) in defining architectures, these classifications cannot offer global or comprehensive perspective of CGRA architectures, somewhat like unfocused descriptions whose key information is drowned out by trivial details. For example, DySER has such an architecture with a 2D direct interconnection, scratch-pad memory, static scheduling, dynamic reconfiguring control scheme, tightly coupled integration, and a compiler using an imperative language [47].

Most of the previous classification methods are based on single dimensions of characteristics. Thus, they cannot reveal the entirety of the characteristics of CGRAs. Some classification methods use several dimensions of characteristics to characterize CGRAs, but they still include too many trivial design considerations that obscure the nature of CGRAs.

2.2 A Multidimensional Taxonomy for CGRAs

We propose a multidimensional taxonomy that classifies CGRAs according to the following abstraction levels of the system architecture:

(1) A programming model is an abstraction of an underlying computer system that allows for the expression of both algorithms and data structures [48]. This model bridges the gap between the underlying hardware and the supporting layers of software available to applications. In fact, all programming languages and application programming interfaces (APIs) are instantiations of programming models. The programming model can describe software applications as well as program hardware [49]. This model abstracts away the hardware details so that programmers can specify parallelism without worrying how this parallelism is implemented in hardware. The model determines which part of algorithmic parallelisms within applications can be explicitly expressed by the programmer. Thus, compiling the algorithms is simplified. For example, a multi-thread programming model, e.g., PThread, abstracts hardware resources as threads so that programmers can represent the parallelism of the application as coordinating threads.

(2) A computation model is an abstract representation of computational semantics that defines a high-level composition of an application [39]. This model provides a computation engine that is capable of producing solutions to any tasks described with the corresponding programming model. Such a model needs to reflect the salient computing

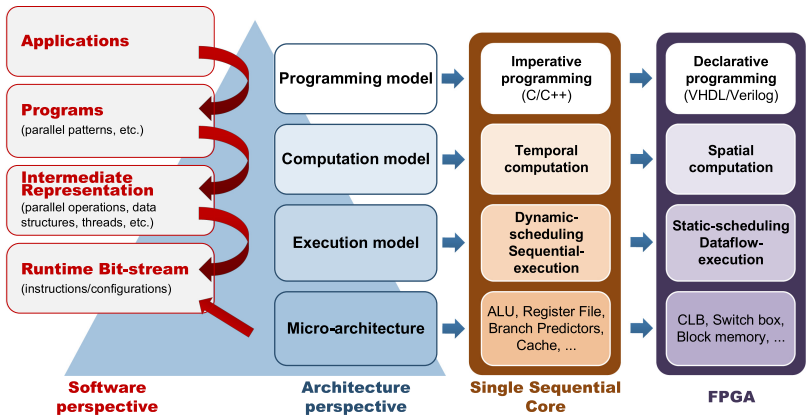

Fig. 2. Comparison of an FPGA and single sequential core under the guidance of the proposed multidimensional taxonomy: high-level abstractions help to improve the quality of classification.

characteristics of the physical computing platform [48]. A computation model actually determines which part of algorithmic parallelisms within applications can be supported and utilized by a specific computation substrate. For instance, thread-level parallelism typically cannot be exploited by a single-threaded computation model.

(3) An execution model is an abstract representation of microarchitectures that defines a scheduling scheme for the intermediate representations of computation models. This model needs to reflect the salient working mechanisms of hardware, such as the triggering, firing, execution and retiring of instructions/configurations. Therefore, the execution model provides a basic framework of microarchitecture design, which determines the runtime executing sequence and concurrency of the representations of computation models.

Figure 2 demonstrates the multidimensional taxonomy method from the perspectives of architecture as well as software on the left. The programming model layer transforms a target application into various programs with different explicit parallelisms. The computation model layer transforms the programs into intermediate representations that consist of operations, data sets or threads in parallel. The execution model layer maps these intermediate representations onto the underlying microarchitecture, generating (offline or on-the-fly) a runtime bitstream that is directly run by the hardware. Figure 2 further clarifies the proposed taxonomy with classic architectures on the right, which compares an in-order single-core processor with FPGA at the abstraction layers as described above. For the microarchitecture, a single sequential core and an FPGA have a large quantity of differences. It is challenging to identify which difference is essential. Moreover, the processors/FPGAs of different types vary a lot in terms of microarchitecture. In contrast, the higher-level abstraction helps differentiate these architectures clearly. An FPGA is typically programmed with a declarative programming model, while an in-order processor uses an imperative model. An FPGA adopts spatial computation, while an in-order processor adopts time-multiplexing computation. An FPGA is statically scheduled and driven by data flow, while an in-order processor is dynamically scheduled and executes sequentially. Therefore, in terms of clarity, the proposed multidimensional taxonomy performs better than previous ones based on microarchitectures. Moreover, this taxonomy help choose the architecture framework according to system-level requirements. Figure 3 illustrates this taxonomy, which is further explained in detail in the following subsections.

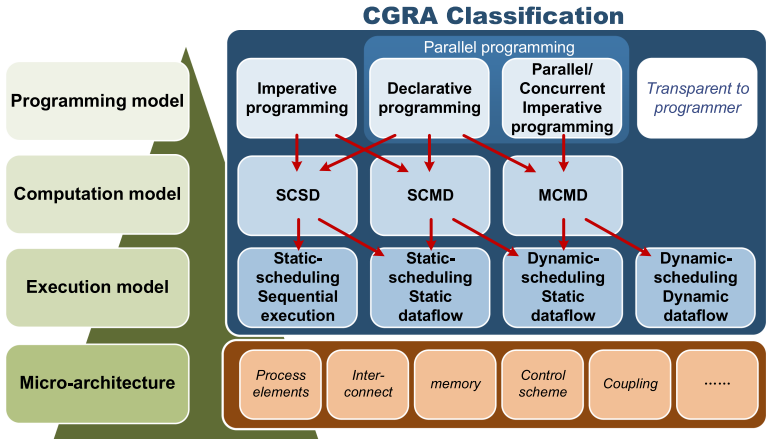

Fig. 3. Classification based on the system architecture.

2.2.1 Programming Model. CGRAs can be classified into two major categories based on the programming model.

The first major category uses an ***imperative programming model*** and imperative languages, such as C/C++. An imperative model uses an ordered sequence of statements or commands or instructions to control the system states and therefore cannot express any parallelism semantically (note that some extensions of imperative languages can express explicit parallelism, such as PThreads; these extensions fall into the category of parallel programming models). This model can be used by all imperative hardware, such as most processors. A CGRA's operation is controlled by its configuration sequence, so an imperative model can also be used. Because imperative languages are relatively easy for programmers and convenient to integrate with general processors, many CGRAs are programmed with imperative models [10, 13, 47].

The second major category uses a ***parallel programming model***. For simplicity, the concept of parallel programming model is used in this article, referring to the programming models that can express some parallelisms. This concept comprises a ***declarative programming model*** (such as functional languages, dataflow languages, and hardware description languages, HDLs), expressing parallelism implicitly, and a ***parallel/concurrent (imperative) programming model*** (such as OpenMP, PThreads, Massage Passing Interface (MPI), and CUDA C), expressing parallelism explicitly or partially explicitly with directives. The declarative programming model uses declarations or expressions instead of imperative statements to describe the logic of computation. This model does not describe any control flow, so computation parallelism is implicitly expressed. The concurrent programming model uses multiple concurrent imperative computations to build a program. Therefore, the model expresses computation parallelism explicitly. Relative to imperative models that rely on compilers and hardware to exploit parallelism, the major difference of parallel programming models is that programmers can express some computational parallelism with them, alleviating the burdens on compilers and hardware and consequently improving efficiency. Although there are fewer CGRAs adopting parallel programming models [20, 50], CGRAs are essentially suitable for these models. The reason is that CGRA hardware is not imperative but perform computations in parallel when it processes one configuration that contains multiple operations.

The third category, ***transparent programming***, does not have any static compilation for a specific CGRA architecture. This category requires dynamic compilation, relying on hardware to translate and optimize common program representations, such as instructions. Thus, its underlying computation models, execution models or microarchitectures could be totally transparent

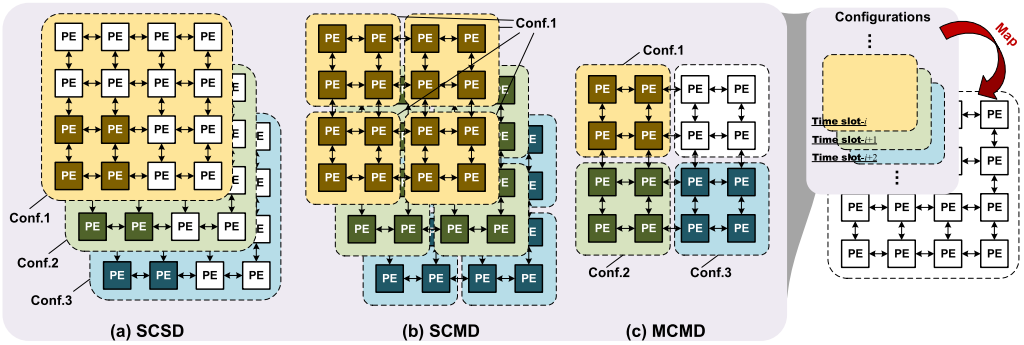

Fig. 4. Computation models of CGRAs: (a) SCSD, (b) SCMD, and (c) MCMD (in this figure, configuration-1 through configuration-3 are independent and asynchronous; rectangles with different colors represent different configurations, and blank ones represent idle).

to programmers. For example, DORA [51], CCA [52], and DynaSPAM [53] are tightly coupled with GPPs, and their configurations are generated according to runtime instruction streams. Remarkably, some CGRAs, e.g., PPA [54], have their runtime bitstream generated on-the-fly from the static compilation results, and thus they do not belong to this category. Transparent-programming CGRAs have two major advantages: productivity can be greatly improved with transparent programming, and optimization can be performed with runtime information that is impossible offline. The major drawback is insufficient performance and considerable energy overhead because additional hardware is responsible for all parallelism exploration at runtime.

2.2.2 Computation Model. All CGRAs belong to *multiple instructions, multiple data (MIMD)* computation according to Flynn's taxonomy [55]. For a more detailed division, considering that the concept of instructions cannot reflect the computing mechanism of CGRA, we introduce a configuration-based classification for computation models, as depicted in Figure 4.

The first category is the ***single configuration, single data (SCSD)*** model. This model refers to a spatial computation engine that executes a single configuration on a single data set. The model is a basic implementation of spatial computation. All the operations of an application or kernel are mapped onto the underlying hardware, so instruction-level parallelism can be extracted unlimitedly. SCSD is a general and powerful engine for different programming models, albeit one limited by the hardware scale. Pegasus is an intermediate representation of the SCSD model, and ASH is a microarchitecture template. Pegasus generates one configuration of ASH for an entire application [56, 57]. The SCSD model mainly exploits instruction-level parallelism. As shown in Figure 4(a), configuration-1 through configuration-3 must be mapped onto three different time slots in the SCSD model because this model does not support simultaneous multi-threading in a PE array.

The second category is the ***single configuration, multiple data (SCMD)*** model. This model refers to a spatial computation engine that can execute one configuration on multiple data sets (spread in spatial). The model can be viewed as a spatial implementation of the SIMD or SIMT model. The SCMD model is suitable for the popular stream-oriented and vector applications, such as multimedia and digital signal processing; hence, it is adopted by many CGRAs [10, 13]. This model mainly exploits data-level parallelism. As shown in Figure 4(b), the configurations of multiple threads in one time slot are identical in the SCMD model.

The third category is the ***multiple configuration, multiple data (MCMD)*** model. This model refers to a computation engine that can execute multiple configurations (i.e., from multiple programs or subprograms) on multiple data sets. Therefore, the model should support both

simultaneous multithreading (SMT) and temporal multithreading (TMT). The threads can communicate via the mechanisms of message passing or memory sharing. Because CGRAs comprise distributed interconnections, interthread communication is typically implemented on this interconnection instead of in shared memory (multiprocessors). Therefore, there are two common subcategories to describe CGRAs' process and communication mechanism. Communicating sequential processes (CSP) is a nondeterministic model for concurrent computation with interprocess communication managed by unbuffered messages passing channels in a blocking manner [58]. Processes can be synchronized through these communicating channels. The Kahn process network (KPN) is another model often adopted by CGRAs. The model comprises a group of deterministic processes that communicate with each other through unbounded first-in, first-out (FIFO) buffers [59]. The communication is asynchronous and nonblocking except when the FIFO buffer is empty. The CSP model, adopted by Tartan to some extent [60], implements asynchronous handshaking communication among PEs for energy efficiency. The KPN model has been applied to many dataflow CGRAs, such as triggered instruction architecture [61] and Wavescalar [62], which use FIFOs and CAMs as communicating channels among asynchronous PEs. The MCMD model mainly exploits thread-level parallelism. As shown in Figure 4(b), the MCMD model supports simultaneous multi-threading in a PE array.

2.2.3 Execution Model. CGRAs can be classified based on the execution model, specifically the scheduling and executing of configurations. (1) The scheduling of configurations refers to the mechanisms of fetching configurations from memory and mapping configurations onto hardware. To minimize hardware overhead, the fetching order and the place where configurations are mapped can be statically decided by compilers. For instance, FPGAs schedule configuration bitstreams totally through compilers. To maximize performance, configurations can be scheduled according to runtime system states (e.g., predicate and data token) and viable resources. For instance, superscalar processors schedule instructions dynamically through predictors and reservation stations. (2) The execution of configurations mainly refers to the mechanism of executing operations within a single configuration. If operations execute in an order determined by compilers, then the process is called sequential execution. If the operations whose operand data are ready takes precedence on execution, then the process is called dataflow execution. Dataflow execution can be further divided into static dataflow execution [63] and dynamic dataflow execution [64, 65]. The static dataflow requires the hardware to execute operations with operand data tokens ready. The dataflow model further requires the hardware to attach tags to tokens. Only if all the operand data tokens that are ready have identical tags, can the corresponding operation instance be executed. The static dataflow execution model does not allow multiple instances of the same routine to be executed simultaneously, whereas the dynamic dataflow execution model does. Therefore, CGRA can be classified into four major categories according to the execution model, as illustrated in Figure 5: (1) ***static-scheduling sequential execution (SSE)***, (2) ***static-scheduling static-dataflow (SSD) execution***, (3) ***dynamic-scheduling static-dataflow (DSD) execution***, and (4) ***dynamic-scheduling dynamic-dataflow (DDD) execution***. Note that the SSD model adopts static dataflow mechanism to schedule the operations within a configuration (at instruction level) while it fetches and maps configurations statically (at thread level). This case is different from that of SSE whose configuration is required to contain only parallel operations or statically sequenced operations. The CGRAs that adopt the former two execution models are more suitable as spatial accelerators, such as DySER and CCA [52], while the CGRAs that adopt the latter two execution models are more suitable as spatial dataflow machines, such as TRIPs and Wavescalar.

2.2.4 Microarchitecture Model. The microarchitectures of CGRAs have been extensively studied in previous works. There are many different classifications based on microarchitectural

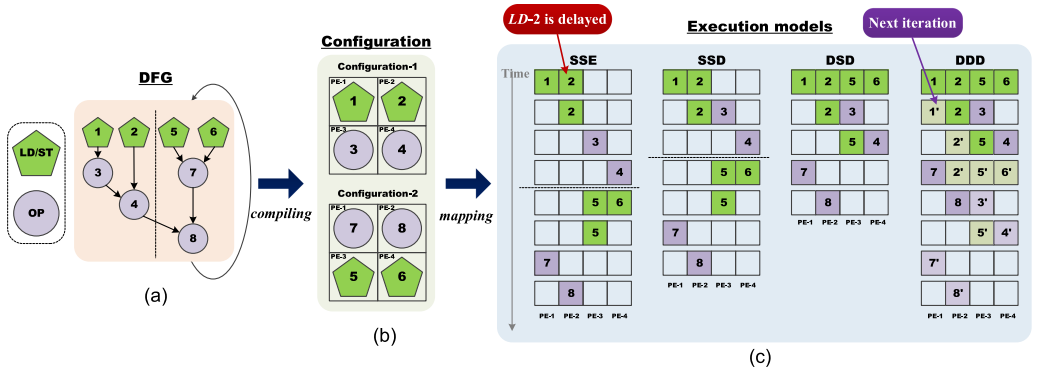

Fig. 5. Comparison between different execution models of CGRAs: (a) an example DFG (a loop body with operations 1–8), (b) spatially mapping configurations, (c) how the configurations execute with different models (operations 1'–8' from the next iteration). Note that DSD and DDD generally require a capability of partial reconfiguration (or MCMD computation).

characteristics, such as ***network/interconnect topology, data path granularity, reconfigurable logic function, memory hierarchy, operation scheduling, reconfiguration mechanism, custom operations, coupling with the host***, and ***resource sharing with the host***. At the microarchitectural level, it is easy to distinguish two CGRAs, but it is difficult to generate a complete classification with clear boundary. For example, a previous work performed an architecture exploration with different interconnect topologies on ADRES [66], implying that interconnect topologies are not essential to characterize ADRES. The situation with most microarchitectural characteristics is similar. An individual CGRA design could have a series of application-dependent variations with different granularity, reconfigurable logic functions, memory hierarchy or integration methods [17]. Therefore, it is not necessary to include these trivial microarchitecture characteristics in our multidimensional taxonomy.

2.2.5 Relations of Different Dimensions. The programming model, computation model and execution model are top-down approaches in hierarchical system design. Therefore, these models have some common corresponding relations, as illustrated in Figure 3. (1) The SCSD model can implement programs with imperative languages and hardware description languages. This model can be implemented on static-scheduling sequential or dataflow execution models. (2) The SCMD model can implement programs with imperative languages and dataflow and functional languages. This model can be implemented on the dynamic-scheduling static dataflow execution model. (3) The MCMD model can implement programs with concurrent programming models. This model can be implemented on the dynamic-scheduling dynamic dataflow execution model. These correspondences expose the intrinsic difference from one CGRA design to another, proving the reasonableness of our taxonomy. Since these correspondences are not one-to-one, it is a challenge to determine which one generates a more efficient CGRA system. The following sections discuss this problem.

Furthermore, the correspondence between the execution model and microarchitecture is analyzed. The dynamic-scheduling model requires a dynamic configuration control scheme, which is not needed in a static-scheduling model. The static-dataflow model requires the microarchitecture to support checking data availability and executing operations dynamically. The dynamic dataflow further requires token passing and a matching scheme. Related problems are discussed in the following sections.

Note that the classifications are neither absolute nor complete. CGRAs in reality might combine several execution models in their architecture or adopt other models in rare cases.

2.2.6 Classifying CGRA Architectures. Table 2 lists the representative CGRAs reported over the past two decades and classifies them according to the above methods.

2.3 Evolution of CGRAs

This section reviews the CGRA evolution in the past two decades under the guidance of the proposed taxonomy and the architecture summary in Table 2.

The dominant programming model of CGRAs is imperative, as indicated in Table 2, for the purpose of being user-friendly. However, the fundamental contradiction between the spatial architecture and imperative programming cannot be reconciled yet [49]. CGRAs with high-level imperative programming languages require significant manual effort for optimization [16, 85]. This has become a compelling problem that hinders widespread adoption of CGRAs. Now, it is a good time to start considering the declarative and parallel/concurrent programming models, although they might be more challenging to use for the programmer. Some novel CGRAs have already altered their programming models [19, 61]. Even if CGRAs might still use imperative programming for productivity in future, they surely need much more powerful programming extensions, including general-purpose and domain-specific parallel patterns.

The mainstream computation model of CGRAs has been SCSD for a long time because SCSD has balanced energy efficiency and flexibility. SCMD could have better energy efficiency for some data-intensive applications, but it is less flexible for general applications [82]. In contrast, MCMD increases control overhead and thus might decrease the computational efficiency [75, 84]. However, as the scale of CGRAs increases, MCMD instances are observed more frequently. The technique of multi-threaded processing can improve the throughput of a large MCMD CGRA [84, 86] but the additional area and power overhead caused by complicated control scheme is a key problem that must be solved to maintain high energy efficiency. This could be a trade-off depending on the application demands. For example, Google still uses classic systolic arrays (SCSD) on TPU for accelerating deep neural networks (DNNs) because of the application type and power budget [87]. In fact, as indicated in Table 2, all three computation models are frequently adopted according to the target domain.

The most popular execution model of CGRAs is SSE because SSE provides an easy-to-use substrate for computation and management. Current compilers can perform static optimization for SSE CGRAs [85]. However, this is insufficient for irregular applications, of which the algorithmic parallelism cannot be exploited offline or working loads change greatly on-the-fly [83]. As a result, a considerable number of CGRAs adopt the other execution models that employ dynamic scheduling or dataflow mechanism to exploit dynamic parallelisms. These models enable high performance for more applications types and alleviate burdens on compilers [19, 20, 83, 84]. The execution models of CGRAs are evolving from static scheduling to dynamic scheduling and from sequential execution to dataflow execution, as indicated in Table 2. This trend is quite analogous to the evolution of CPU architectures from VLIW to out-of-order. Although the dynamic scheduling and dataflow mechanism consume additional power, they are worthwhile if greater performance improvement can be achieved.

2.4 Application Status

This part summarizes the domains to which CGRAs have already been applied. Since they sacrifice a degree of fine-grained flexibility and interconnect flexibility, CGRAs are favored as domain-specific accelerators rather than general-purpose prototyping and demonstration platforms. More

Table 2. Classification of Representative CGRAs

Architecture	Year	Programming model*	Computation model	Execution model**	Specifications
Xputer [8]	1991	D	SCSD	SSE	
PADDI [9]	1992	I	SCSD	SSE	
PADDI-2 [67]	1993	D	SCMD	DSD	
RAW [68]	1997	I	MCMD	DSD	*More like a multicore processor*
PipeRench [69]	1998	D	SCMD	SSE	
Morphosys [11]	2000	I	SCMD	SSE	
Wavescalar [62, 70]	2003	I	MCMD	DDD	*Dataflow-driven ISA*
PACT-XPP [13]	2003	I/C	SCSD	DSD	
DRP [26]	2004	I	SCSD	SSE	*programmable FSM controller*
ADRES [10]	2004	I	SCSD	SSE	*VLIW controller*
ASH [57]	2004	D	SCSD	SSD	
TRIPS [12]	2004	I	MCMD	DSD	*Dataflow-driven ISA*
CCA [52]	2004	*transparent*	SCSD	SSE	*Runtime-generated configurations*
Tartan [60]	2006	I	MCMD	DSD	*Asynchronous circuit*
TFlex [71]	2007	I	MCMD	DSD	*Dataflow-driven ISA*
RICA [72]	2008	I	SCSD	SSE	
PPA [54]	2009	I	SCSD	SSE	*Polymorphic configurations*
TCPA [50]	2009	D	SCSD	SSE	
C-Cores [73]	2010	I	SCSD	SSE	*Targeted reconfigurability, ASIC-like*
DySER [47]	2012	I	SCSD	SSD	
REMUS [30]	2013	I	SCSD	SSE	
Triggered Inst. [61]	2013	D	MCMD	DSD	
T3 [74]	2013	I/C	MCMD	DSD	*Dataflow-driven ISA*
SGMF [75]	2014	I	MCMD	DDD	
FPCA [76]	2014	I	SCSD	SSE	
DynaSPAM [53]	2015	*transparent*	SCMD	SSD	*Based on PipeRench*
NDA [77]	2015	-	SCSD	SSE	*Process-in-memory*
HARTMP [78]	2016	I	SCMD	DSD	
DORA [52]	2016	*transparent*	SCSD / SCMD	SSD	*Based on DySER*
HRL [79]	2016	D/I	SCSD	SSE	*Process-in-memory, mix-grained*
HReA [16]	2017	I	SCSD	SSD	*General-purpose*
Plasticine [19]	2017	D	SCMD / MCMD	SSD	*Parallel-pattern-based programming*
Stream-dataflow [20]	2017	I	SCSD	DSD	*Vector memory interface*
CGRA-ME [80]	2017	I	SCSD	SSE	*ADRES-like*
Wave DPU [18]	2017	I/C	SCSD	SSD	*Commercial product for DNN*
PX-CGRA [81]	2018	-	SCSD	SSE	*Approximate PEs*
i-DPs CGRA [82]	2018	-	SCMD	SSE	*Double-ALU/Reg. in each PE*
Parallel-XL [83]	2018	I/C	SCMD/MCMD	DDD	*Intel Cilk & work stealing*
dMT-CGRA [84]	2018	I/C	MCMD	DDD	*Based on SGMF*

*I–imperative programming model, D–declarative programming model, C–parallel/concurrent (imperative) programming model, "*transparent*" means that CGRA-related programming is not required, "-" means that programming is not mentioned in that work.

**SSE–static-scheduling sequential-execution, SSD–static-scheduling static-dataflow-execution, DSD–dynamic-scheduling static-dataflow-execution, DDD–dynamic-scheduling dynamic-dataflow-execution.

specifically, since they provide sufficient computation resources but have less support for control flow, CGRAs are mostly used in computation- and data-intensive domains, which is consistent with the trend of emerging applications. The application status of CGRAs is introduced in terms of the following aspects.

Security. Security applications, e.g., encryption and decryption, impose requirements on the flexibility and physical-attacks-resistance of underlying hardware because security applications typically contain hundreds of ciphers that evolve continuously and suffer from physical attacks that are more threatening than conventional attacks. CGRAs meet these requirements. In addition to flexibility, CGRAs possess a capability to resist physical attacks because software-defined hardware is safer than individual hardware implementation or software implementation. CGRA-based cryptographic processors have attracted increasing interest in recent years. COBRA was proposed in Reference [88] as a reconfigurable processor, which is customized for over 40 symmetric ciphers. Celator was proposed in Reference [89] as a reconfigurable coprocessor for block ciphers (AES and DES) and hash functions (SHA). Cryptoraptor is a reconfigurable cryptographic processor proposed in Reference [90] for symmetric-key cryptography algorithms and standards. This processor is reported to support the widest range of cryptographic algorithms, with a peak throughput of 128Gbps for AES-128.

Signal and image processing. Signal and image processing applications are typical stream processing. Since CGRAs perform well at streaming processing, a large quantity of CGRA architectures targets these applications. The classic architecture, ADRES, has been applied to video processing (H.264/AVC decoding [91]), image processing [92], a software defined radio (SDR) signal processing (SDM-OFDM) receiver [93], and MIMO SDM-OFDM baseband processing [94]). The commercial XPP-III has also been applied to video processing (MPEG4 and H.264/AVC decoding [30]). A FLEXDET was proposed in Reference [95] for a multimode MIMO detector. The CGRA proposed in Reference [96] implemented the multioperable GNSS positioning function. The CGRA proposed in Reference [30] implemented MPEG4/H.264/AVS standards. In industry, Samsung has applied a CGRA video processing platform for 8K UHD TV [22].

Deep learning. DNNs have become quite popular since the 2010s in modern artificial intelligence (AI) applications, including computer vision, speech recognition, medical, game play and robotics. DNNs perform complex computation on a large amount of data with frequent interlayer communications. CGRAs are capable of high-throughput computation and on-chip communication, making them a superior implementation for DNNs. Eyeriss, proposed in Reference [97], minimizes data movement energy consumption through maximizing input data reuse. A reconfigurable architecture that can reconfigure its data paths to support a hybrid data reuse pattern and scalable mapping method was proposed in References [98, 99]. A runtime reconfigurable 2D dataflow computing engine that can implement a variety of CNN operations in a systolic manner was proposed in Reference [100].

Others. All the domains above contain intensive computation and data movement, which benefit most from CGRA acceleration. Other similar domains, such as network processing [101], spacecraft [24], biomedicine, and scientific data analysis are also suitable for CGRAs.

3 CHALLENGES REGARDING CGRAS

Currently, CGRAs are still too immature for wide commercial use in terms of *programmability, productivity* and *adaptability*. This section analyzes the root causes for these shortages and identifies the most critical challenges under the guidance of the proposed taxonomy. Figure 6 illustrates the analytical method of this work. The challenges are discussed, and then the corresponding technical trends are discussed in the next section.

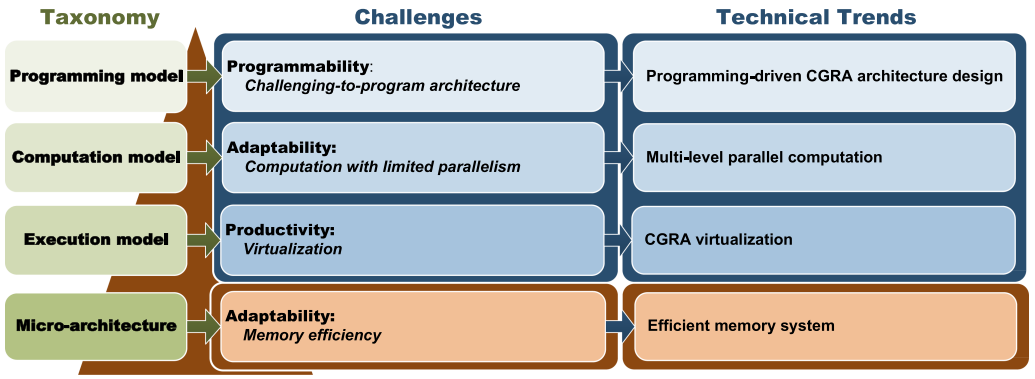

Fig. 6. Relations between the taxonomy, challenges, and technical trends in this survey.

3.1 Challenge 1: Challenging-to-Program Architecture

3.1.1 Problem Formulation. As the top level of architecture abstraction that describes applications, the programming model has a major impact on the performance and productivity of the underlying computation architecture [49]. An extremely low-level programming model is unfriendly to programmers and requires a long compilation time, while an extremely high-level programming model makes it challenging to create a high-performance implementation. However, this problem has been only rarely considered in previous works, which makes most CGRA implementations challenging to use or inefficient [19]. Therefore, it is a crucial problem to identify the most suitable programming model at the very beginning of CGRA design.

3.1.2 Major Challenges. It is a major challenge that an ideal programming model for CGRAs must balance the competing goals of productivity and implementation efficiency. **(1) From a hardware perspective**, the fundamental reason is that CGRAs are powerful computing substrates completely different from conventional sequential processors. Their architectures are far more complicated than CPUs because a two-dimensional array of hardware resources must be scheduled and cooperate. They are even more difficult to program than VLIWs (general-purpose VLIWs, e.g., Itanium, turned out to be a dramatic failure, since the wished-for compilers were basically impossible to write), because explicit dataflow via interconnections is more difficult to manage than implicit dataflow via shared memory. CGRAs task partitioning is also more difficult than that for FPGAs due the characteristic of temporal computation, and programming FPGAs with high-level languages is still formidable. Therefore, the extreme architectural complexity makes programming CGRAs particular challenging. **(2) From a software perspective**, there are fundamental conflicts between the complicated CGRA architectures that demand massive fine-grained parallelism and the popular software programming models that exhibit sequential styles. To achieve productivity, the programming model should use abstraction to avoid the need for programmers to learn the architecture's intricate details. High-level abstraction can only provide coarse-grained parallelism, which is insufficient to fulfill the hardware potential. To achieve high efficiency, the model should explicitly expose the key elements of the architecture that have an impact on implementation efficiency such that programmers can explore the architectural design space to determine the performance bottleneck. For example, FPGAs can be programmed with high-level models (such as OpenCL) and high-level synthesis tools, generating less-efficient designs [102–104]. Alternatively, FPGAs can be programmed with low-level models (such as VHDL) and compiled by synthesis, placement and routing tools, thereby generating an implementation of higher performance but consuming longer design and compiling time [105]. The struggle is delivering performance while

increasing the level of abstraction [49]. In other words, the challenge of designing a CGRA programming model lies in how to expose only the architecture features that have a severe impact on performance while isolating the other hardware details. There are successful examples of similar architectures, such as CUDA for GPGPU.

Since CGRAs are mostly domain-specific and their performance also depends on applications, another challenge is how to take application characteristics into consideration during programming model design (such as in the MATLAB language). It is expected that the application-oriented extensions to the programming model could ease the programming effort and improve performance.

Currently, there are no perfect programming paradigm for CGRAs. Most CGRAs and their compilers, if any, require a significant amount of manual work. Specifically, these CGRAs use high-level languages for programming and rely on the compiler front-end to transform the high-level input into a lower-level intermediate representation, which will subsequently be optimized and converted into machine code. In these cases, a remarkable gap exists between the compiler results and manual optimization results [38], which implies that exposing more architecture details to programmers (in other words, adopting a lower-level programming model) would be more efficient than merely relying on compiler-oriented optimization.

3.2 Challenge 2: Computation with Limited Parallelism

3.2.1 Problem Formulation. As the abstraction level that exploits algorithmic parallelism, the computation model is often required to support efficient implementation of various parallelism semantics in the programming model. The key motivation is implementation efficiency. Real applications always contain various proportions of algorithmic parallelism. An architecture that can exploit more algorithmic parallelism achieves better performance. This can be realized with architectural support of multilevel parallelism, including instruction-level parallelism (ILP), data-level parallelism (DLP), memory-level parallelism (MLP), task-level parallelism (TLP), and speculative parallelism. For example, Superscalar processors are more efficient than scalar ones due to the utilization of ILP, multi-core processors are even more powerful than the superscalar due to the utilization of TLP, and out-of-order processors have better performance than in-order ones due to the utilization of speculative parallelism.

However, supporting multilevel parallelism does not come without a price. The area and power overheads are two major limitations on the forms of parallelism feasible to CGRAs. When the power and area overheads offset the performance gain, the energy efficiency and area efficiency are degraded. The degradation depends on both architectures and applications. Therefore, how to implement multilevel parallelisms in the computation model is the major problem discussed in this part.3.2.2 Major Challenges.

CGRAs already support a variety of parallelism in their computation models. For example, TRIPS supported three modes of execution, each of which is well suited for a different type of parallelism, i.e., ILP, DLP and TLP [12]. Polymorphic Pipeline Array supported fine-grained parallelism with software pipeline and coarse-grained pipeline parallelism, which come from ILP and TLP [54]. MLP and DLP support have become a topic in recent CGRAs for data-intensive domains [19, 20]. However, there are few CGRAs that support speculative parallelism well, which is an important resource for parallelism exploration. The major impediment to implement the speculative parallelism lies in area and power limitations. Although speculative execution has been exploited in GPPs, problems that are challenging to surmount arise when this technique is applied to CGRAs.

Speculation is a fundamental method in computer architecture that exploits parallelism by first eliminating some dependences and subsequently checking and restoring/validation. Speculation

is beneficial for performance when the gain in parallelism outweighs the checking and recovering cost. It plays a quite important role in improving the implementation performance of many applications, because predictable dependences, e.g., control dependence and ambiguous memory dependence, provide enormous potential benefit to the speculation technique. Control dependence poses a relatively weaker order on two instructions than data dependence because the consumer instruction only needs a predicate bit from the producer instruction. Ambiguous memory dependence is a partial data dependence, which might not require any order when aliasing memory accesses turn out to be conflict free. According to analyses on popular benchmarks, such as SPEC2006 and EEMBC, branch instructions account for approximately 20% of all instructions [106, 107], which means that control dependence accounts for more than 20% of all dependences. As discussed in [10], since the computation-intensive kernels could be accelerated to a large degree (~30×), the rest workload, which is often limited by the control dependence or ambiguous dependence, would dominate the total execution cycles according to Amdahl's Law. The authors of Reference [108] conducted an analysis regarding the speculative parallelism potential of SPEC2006; the results showed that speculatively parallelizing loops (including speculation on loop control dependence and loop-carried ambiguous memory data dependence) can improve the overall performance by more than 60%. This result motivates applying speculation techniques to CGRAs.

Speculative parallelism is exploited in GPPs with the main techniques of branch prediction and out-of-order execution. These techniques have been widely adopted by commercial GPPs to ensure satisfactory performance. (1) Branch prediction is typically combined with speculative execution, eliminating the control dependence at the cost of a misprediction penalty. The predicted path can be prefetched or even promoted before the branch instruction such that the dependence chain length can be shortened. The misprediction penalty includes flushing and refilling the processor pipelines, employing additional buffers to store the modifications of speculative instructions to the system states before they can be correctly committed, or implementing a roll back mechanism for these modifications if necessary. (2) Out-of-order execution is a speculative technique aimed at memory dependence. Ambiguous memory accesses execute concurrently as if there were no dependence. Thus, the instructions dependent on them are not delayed. Load-store-queue is usually adopted to buffer these memory accesses before they are committed to the memory system. Although these techniques achieve conspicuous success in improving performance, their area and power overhead is also significant. According to the power analysis on RISC processors (90nm), the energy dissipation of arithmetic operations that perform the useful work in a computation only accounts for less than 1% of total power dissipation in media-processing applications [109]. Speculation techniques play an important role in decoding and scheduling instructions for deep pipelines and multiple processing units, which are extremely power-hungry. This is evidenced by the fact that out-of-order cores are less energy-efficient than in-order cores.

CGRAs are spatial architectures that enable temporal computation, which is different from GPPs and makes CGRAs challenging to exploit speculative parallelism. First, the branch prediction has a lower accuracy when performed at the configuration level because the branch history is much shorter than the instruction-level prediction [74]. Although the accuracy could be improved by capturing all branch-related information in configurations, the hardware cost becomes unaffordable because of the considerable array scale. Second, the branch misprediction penalty booms because of configuration-level checking and validation, which implies flushing and refilling the corresponding configurations and buffering and reordering all memory accesses in these configurations. Consequently, in contrast to GPPs, the performance penalty, power overhead and area overhead could be much larger. Considering that the prediction accuracy decreases and the misprediction penalty increases, the branch prediction technique becomes far less effective. Third, the out-of-order execution technique also suffers from the significant overhead of buffering

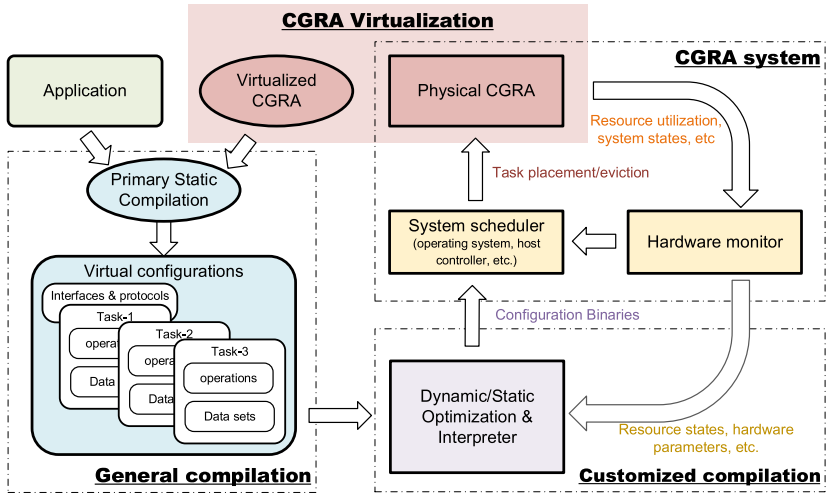

Fig. 7. CGRA virtualization and the supporting system.

configuration-level memory accesses (memory accesses within a configuration are atomic). As a summary, the major challenges to exploiting the speculative parallelisms on CGRAs are how to minimize mis-speculation ratio and penalty.

3.3 Challenge 3: Virtualization

3.3.1 Problem Formulation. The algorithmic parallelisms exploited by the computation model can be represented in the forms of parallel operations, data sets or tasks. The execution model is an abstraction that describes how to schedule these intermediate representations onto suitable hardware substrates. The underlying hardware of CGRAs has extremely variable designs and development environments, posing a challenge for the productivity of various CGRAs. Virtualization is an effective solution to this problem. As illustrated in Figure 7, the CGRA virtualization method provides a unified CGRA model, i.e., virtualized CGRA, which comprise standardized interfaces, communication protocols and an abstraction of execution. On basis of this model and input applications, the static compilers generate virtual configurations that fit this series of CGRAs. Then, the virtual configurations are optimized and interpreted online/offline onto a specialized physical CGRA. In addition, the generated configuration binaries are sent to system schedulers, which determine runtime task placement and eviction according to the resource utilization and states. Virtualization facilitates the usage of CGRAs with unified models such that CGRAs can be easily incorporated into operating systems. Virtualization also facilitate the CGRA design because the designer can produce any application-dependent CGRAs that fit the common development environment. Therefore, the major problem to be discussed in this part is how to implement virtualization of CGRAs.

3.3.2 Major Challenges. Virtualization of reconfigurable components can be traced back to the idea of integrating FPGAs into operating systems in 1990s. CGRA virtualization is quite similar to the relatively developed FPGA virtualization but is much less studied. The reason is that the architecture and design of CGRAs are far from mature. Many problems are still unsettled: there are no widespread commercial products, there are no acknowledged fundamental research platforms and compiling systems, there are no common benchmarks for evaluation, and there are no acknowledged architecture templates. Remarkably, many researchers are continually proposing various

Table 3. Design Space of CGRAs: Microarchitectural View

Dimensions/Property	Design space
PE function	LUT-ALU-Approximate Unit-Custom block
Interconnect topology	Crossbar-Mesh-Bus-NoC
Reconfiguration	Static-Dynamic; Partial-Full
Memory system	Scratchpad-Cache-PIM-computation memory
Communication among PEs	RF-FIFO-Wire switch; Synchronous-Asynchronous-Elastic
Interface with the main controller	Loose-Tight; Datapath-Accelerator-Coprocessor
Control scheme of PEs	Distributed-Centralized
Main controller	FSM-GPP-DSP-VLIW

novel architectures, but it is difficult to make any objective and detailed comparisons. This situation results from the dramatic diversity of and controversy regarding architecture designs, which has become a major challenge to the CGRA virtualization. Currently, it seems almost impossible to extract a unified hardware abstraction on CGRAs with dramatically varied execution models and microarchitectures, such as external interfaces, memory systems, system control schemes, PE functions, and interconnection. Another challenge lies in compilation. Since CGRAs mostly rely on static compiling to exploit their computational resources and interconnections, it is difficult to dynamically schedule those configurations whose execution sequencing and synchronous communication have been settled.

3.4 Challenge 4: Memory Efficiency

3.4.1 Problem Formulation. The microarchitecture directly determines the performance, area and power consumption of hardware implementation of algorithmic parallelisms. In this part, we concentrate on the challenges at the microarchitecture level, and the major target is efficiency. As indicated in Table 3, the microarchitectures of CGRAs exhibit noticeable design differences in terms of many dimensions. Designers typically perform design space exploration of these dimensions to customize microarchitectures for specific applications [41]. Take PE function as an example. Approximate computing units have been introduced to improve energy efficiency at the expense of accuracy losses [81, 110]. This method is effective only for applications that tolerate inaccuracy, such as multimedia processing, signal processing and DNNs. Custom design can be utilized to enhance the data processing capability through sharing single control logic among multiple ALUs and registers [82]. This method is effective for parallelizable and reduction kernels, such as the bio-DSP domain.

However, based on this design space, CGRAs cannot avoid the well-known problem of the memory wall, especially when performing data-intensive applications. We argue that the design of memory systems should be further explored. Specifically, there are two motivations. First, with the development of information technology, the size of the data on which computations are performed is ever increasing, and the energy overhead of data movement has already exceeded that associated with computation. For example, data movement between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating point operation [111]. Data movement with DRAM consumes approximately 95% of the total energy of an architecture design for data-intensive applications [112]. The era of big data has arrived, which is altering the form and location of memory on all computing platforms. The impact will be especially significant for CGRAs. Second, current CGRAs adopt conventional cache-based or scratch-based memory schemes that are developed for general-purpose computers, e.g., CPUs and GPUs. Since these designs are not carefully explored or optimized for different memory access patterns, they have become

limitations on memory bandwidth. Developing a flexible memory system to accommodate constantly evolving applications and algorithms has consequently been attracting attention.

3.4.2 Major Challenges. Since the problem of the memory wall has a similar impact on all computing platforms, mature solutions could be introduced into CGRAs. However, CGRAs have some distinct characteristics that produce additional challenges.

First, as a spatial computing fabric, many CGRAs have arrays of spatially distributed PEs, which could launch many scattered and redundant memory address generation and memory access events. However, regular memory access with continuous addresses or in vector style is more efficient. Thus, CGRAs suffer more from the memory wall than vector architectures that have a parallel memory interface.

Second, most CGRAs can specialize the computing facilities for various applications, but few of them can customize the memory access pattern for different applications. These CGRAs typically adopt a conventional memory system, which cannot explore memory-level parallelism efficiently at runtime or upon compilation.

Third, as a typical spatial computing fabric, CGRAs are suitable for integration with the memory array in a PIM manner. In theory, a DRAM can incorporate a PE into each block as described previously [113]. However, because the logic of a PE is much more complex than Boolean logic operations or comparators, it would be too challenging to integrate CGRAs into the DRAM process technology. Moreover, the memory technology is optimized for storage density and has a lower speed [114], which would limit the performance of the integrated logics.

4 STATE-OF-THE-ART CGRAS AND TECHNICAL TRENDS

In this section, we survey the state-of-the-art techniques developed in response to the four challenges described in Section 3. Note that some challenges have not attracted enough attention from CGRA researchers, so the corresponding survey covers the techniques adopted by similar architectures, such as FPGAs. Based on existing methods, prospective solutions are further discussed.

4.1 Trend 1: Programming-Driven Architecture Design

4.1.1 State-of-the-art Techniques. As CGRAs can serve as domain-specific accelerators, the mainstream of CGRA design is still driven by applications. Figure 8(a) illustrates a common design flow for CGRAs [41, 115, 116]. The steps on the right column show a common compilation flow. The target applications, which are typically described in high-level languages, are first partitioned and optimized separately for the host processor and reconfigurable fabric. The front end of the compilation tools generates a low-level intermediate representation, which exposes some concurrency of input programs. The middle end of the compilation tools makes further code optimizations, and the back end performs the low-level steps of mapping, placement and routing [38]. The steps on the left column show the design and verification flow. Applications are profiled first to identify the hotspot region, which is used to guide partitions. The application features can be extracted according to the analysis of hotspots, which are used to guide architecture design space exploration. The performance of the generated architecture specification in architecture description language (ADL) is evaluated via simulation tools together with the compilation output [117]. Based on the evaluation results, the design space exploration is iterated with a loop closure to identify the suitable architecture for target applications.

However, the application-driven design flow borrowed from ASIC designs focuses too closely on the performance of a specific application while overlooking the programmability and overall performance of the targeted domain of CGRAs. There are two problems in this flow. First, neither programmers nor users play any role in the whole design flow. Productivity is generally not

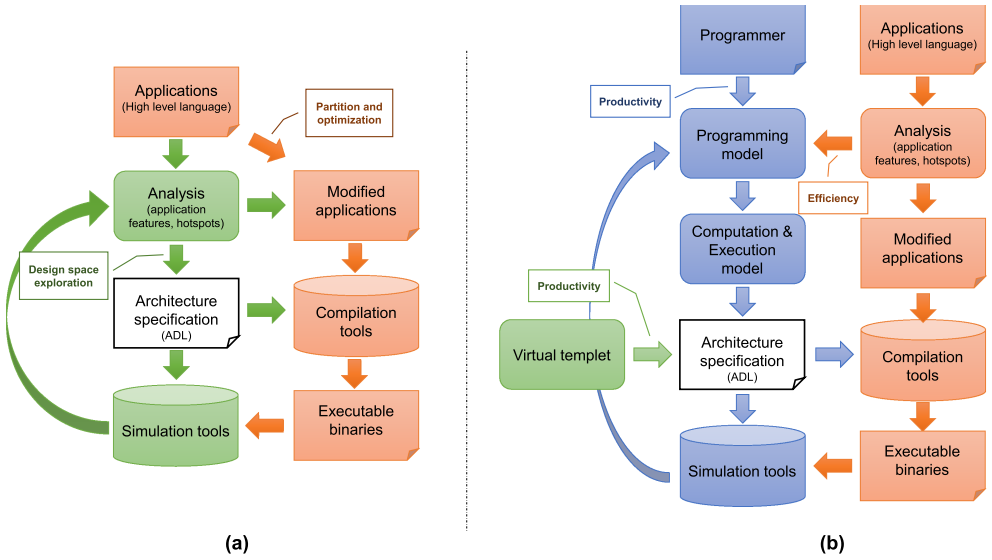

Fig. 8. Comparison of (a) application-driven CGRA design flow and (b) programming-driven CGRA design flow.

considered at all, and thus the created CGRA architecture could be challenging to use. Second, the feasible design space exploration is typically limited to an architecture baseline or specification format. However, it was not stated clearly in previous works which baseline was used or why the baseline was adopted [66, 85, 118, 119]. If the baseline is not suitable to the target domains, then the process of the design space exploration might converge to a suboptimal solution. In all, application-driven design flow concentrates on improving domain-oriented performance but neglects the importance of programmer-oriented productivity and the architecture models' efficiency, which is insufficient to create an ideal architecture.

CGRAs that can be transparently programmed seem to be a fundamental solution to the challenge of an ideal programming model, since they do not require any CGRA-oriented programming. However, they have major drawbacks, including insufficient efficiency and considerable energy overhead, because all the work of compilers is offloaded to hardware translators and optimizers, which overburdens hardware.

4.1.2 Prospective Directions. It has already been realized that programming models should be introduced into the design flow of CGRAs as a primary design issue.

This trend emerged earlier in FPGA design, which can be considered as a commercial fine-grained analog of CGRAs. George et al. [120] introduced a high-level domain-specific language (DSL) into FPGA development. This language helps programmers expose the computing features of applications and then map them into a set of structured computation patterns, such as map, reduce, foreach, and zipwith. Because these computation patterns are few and well understood, it is natural to use premeditated strategies to optimize these patterns and generate high-quality hardware modules to implement them. Therefore, this design flow uses the application features to design a programming model (in declarative functional language) with domain-specific semantics, achieving a suitable balance among productivity, generality and efficiency. Prabhakar et al. [121] further performed optimization on the compilation process from parallel patterns to FPGA design with tiling and metapipelining. Koeplinger et al. [122] presented an overall framework of an FPGA design generation from a high-level parallel patterns programming model, where a parameterized

IR and DHDL were adopted to refine the design space exploration. The results demonstrate a 16.7× speedup relative to a multicore commodity server processor. Li et al. [123] introduced a programming model with thread-level speculation into FPGA hardware generation to accommodate irregular applications whose data or control dependences are determined at runtime. Accordingly, an inherently parallel programming abstraction that packages parallelism in the form of concurrent tasks for the runtime schedule was proposed. The generated hardware could manage the software tasks dynamically according to task rules. A performance improvement relative to server-grade multicore processors was observed.

Clearly, the trend of programming-driven design flow of FPGAs can be applied to CGRAs. Prabhakar et al. [19] proposed a CGRA architecture specialized for a parallel patterns programming model in which pattern compute units were specialized for computing nested patterns as a multistage pipeline of SIMD PEs and pattern memory units were specialized for data locality as banked scratch-pad memory and configurable address decoders. Nowatzki et al. [20] proposed a so-called stream dataflow architecture that was composed of a CGRA, a control core and a stream memory interface. In this architecture, the design flow begins with a target domain analysis. The characteristics of applications are summarized as follows: high computational intensity with long phases and small instruction footprints with simple control flow, straightforward memory access and reuse patterns. Subsequently, a stream-dataflow programming model, its execution model, and ISA were proposed. Finally, the microarchitecture was implemented with up to 50% power efficiency of an ASIC (matching performance).

In Figure 8(b), a programming-driven design flow is summarized according to the above works for both FPGAs and CGRAs (the virtualization is discussed in the following part). In contrast to the application-driven design flow in Figure 8(a), programmers play a key role in designing a productive and efficient programming model, which determines the underlying computation model, execution model and microarchitecture to a large degree. Two critical factors should be considered by programmers. First, the abstraction level of the programming model should be high enough to abstract away hardware details such that programmers without hardware knowledge can use it. Second, the abstraction of the programming model should be able to explicitly express the features that boost performance of the target applications. At least the following features should be covered during application analysis:

(1) Independent task parallelism is straightforward to use and orthogonal to other parallelisms. Such parallelism should be supported in programming model designs.
(2) Word-level parallelism is common in computation-intensive applications and has already been supported by most CGRA hardware.
(3) Bit-level parallelism can be found in some cryptography applications such as SHA but is incompatible with normal CGRA granularity. Therefore, such parallelism results in extensive modifications to the whole design if it is required by applications. Of course, the benefit is also significant if a mix-grained CGRA targeted at cryptography could support bit-level parallelism.
(4) Data locality in various applications is diverse. If this aspect can be explicitly expressed in programming and then implemented with spatially distributed efficient CGRA interconnections, then the burden on the external memory bandwidth will be lowered.
(5) The memory access patterns of target domains should be analyzed to help alleviate the possible memory bound in low-level architecture design.

In summary, the design of the programming model is the first step of CGRA design flow. Determining how to design a programmer-oriented and application-oriented efficient programming model is the primary problem that should be considered by CGRA designers.

4.2 Trend 2: Multilevel Parallel Computation

4.2.1 State-of-the-art Techniques. **Instruction-level parallelism** means to perform multiple independent instructions or operations from one task in parallel. Such parallelism can be implemented with many architectural techniques. For instance, in a general-purpose processor, the instruction pipelining technique overlaps multiple instructions' execution. The instructions could be dependent, but this condition will bring about a pipeline race and hazard. In superscalar and VLIW processors, multiple execution units are used to execute multiple independent instructions in parallel. These instructions are typically exploited by compilers. In out-of-order processors, instructions are executed in an order that is determined dynamically by data flow. Thus, out-of-order execution exploits ILP in a dynamic hardware manner. All of the above techniques have already been introduced into CGRAs. First, the instruction or operation pipelining does not match CGRAs in nature, because CGRAs rely mainly on energy-efficient spatial computation rather than time-multiplexing instruction pipelines. Nevertheless, many CGRAs adopt the software pipelining technique to explore a coarser granularity of kernel- or loop-level parallelism [124–127]. Second, in contrast to one-dimensional spatial computation in the VLIW or superscalar processors, CGRAs advance to two-dimensional spatial computation, which is one of their essential characteristics and is adopted by almost all CGRAs. Third, the dataflow technique is widely used in CGRAs [12, 20, 62, 75]. This technique can be used in hardware to manage the inherent execution order and data preparation order of a static-scheduling DFG, or it can be used to accommodate simultaneous multiple DFGs on a single array. Therefore, in general, CGRAs use the spatial computation technique to implement the ILP that is statically explored by compilers, while they use the dataflow technique to dynamically explore the ILP that cannot be traced by compilers. Memory-level parallelism can be viewed as a special type of ILP, and it is typically implemented with a dataflow technique.

The major differences between the ILP in CGRAs and other architectures are area and power overhead. First, CGRAs provide distributed interconnect, which is much more energy-efficient than the multiport register files in CPUs, GPUs, DSPs, and so on, resulting in a much smaller power overhead. Second, CGRAs rely on both compilers and the dataflow dispatch method, resulting in a smaller area and power overhead than out-of-order processors.

Data-level parallelism means to perform the same operation on different subsets of a large data structure. This term typically refers to SIMD computation such as that performed by GPGPUs. Although CGRAs are generally considered more compatible with MIMD computation, a few CGRAs adopt an SIMD-like structure, i.e., SCMD to reduce the power and area overhead of instruction dispatch [19, 11]. CGRAs can implement DLP with higher efficiency than SIMD/SIMT, because they can provide programmable interfaces to consolidate memory accesses and avoid redundant address generation. CGRAs also provide efficient memory for data reuse. However, these CGRAs could be quite domain-specific.

Task-level parallelism or thread-level parallelism means to perform uncorrelated instructions or operations from different tasks in parallel. The multiple tasks typically execute asynchronously, while many CGRAs adopt a centralized control scheme in which all PEs' execution and reconfiguration occur in lockstep, i.e., using a large configuration for energy and resources consideration [10, 47, 69]. Because of synchronization conflicts, TLP cannot be implemented on such CGRAs. Nevertheless, multiple centralized CGRAs could be integrated to form a larger CGRA that enables coarse-grained TLP. More recent CGRAs have adopted a distributed control scheme such that operations from multiple tasks can execute simultaneously [86]. The dataflow technique is suitable to implement TLP. The static dataflow enables multiple DFGs to execute in spatial parallel (such as TRIPS [12]), whereas the dynamic dataflow further enables multiple DFGs to execute in

an overlapping manner (such as Wavescalar and SGMF [75]). Various implementation forms for intertask communicating in CGRA, such as shared memories, distributed FIFOs and circuit-switch interconnections, exist.

Compared to multiprocessing, CGRAs rely on the dataflow technique to enable fine-grained TLP, avoiding complicated multi-issue and multithreading logic overhead. They also provide much more energy-efficient intertask communication than the usual shared memory.

Speculative parallelism means to perform multiple dependent operations in parallel as if they were independent or uncorrelated. Conservative parallelism must be strictly consistent with the data and control dependences, which means that any instruction pair that possibly has an internal control or data dependence must execute in sequence. In contrast, speculative parallelism needs to break these dependences to shorten the length of the dependence chain. To ensure the accuracy of program execution, speculative operations must be squashed and the modification to the system state should be rolled back if these operations turn out to be not performed. Such parallelism can provide a large quantity of supplementary instruction-level, data-level or task-level parallelisms at the cost of dynamic buffering, detecting and squashing mechanisms. For example, a general-purpose processor uses the branch prediction technique to exploit speculative parallelism in the pipeline. Two control-dependent instructions can be predicted as independent and then issued to the pipeline successively without any bubbles, thus avoiding pipeline stalling. If the prediction is incorrect, then the pipeline is thoroughly flushed, causing even longer pipeline stalling. In a multiprocessing system, speculative parallelism is typically exploited at the task level. The task-level speculation (TLS) technique can perform multiple threads that may have internal data dependence in parallel. Threads should be squashed at the detection of any dependence violations. Reorder buffers or similar structures are typically used to buffer all the modifications to the system state by speculative operations or threads to avoid rolling back the whole system state.

To the best of our knowledge, only a few CGRAs support speculative parallelism at either the instruction level or the block level. Partial predicated execution [128, 129] is a method to explore ILP through speculatively executing the branching paths in parallel with conditional computation. The side effect of the paths not taken must be limited inside the array, so the method has limited applications. TFlex [71] proposed a block-level fused predicate/branch predictor for CGRAs that uses all the predicates in a block to predict the next block. The speculative block is flushed from its pipeline if misprediction is detected.

4.2.2 Prospective Directions. The control flow in programs can be implemented in three forms on CGRAs. First, the external configuration controller (e.g., the GPP and FSM module) could express all control dependences as configuration sequencing. Second, the PE array could express the forward control dependence inside a single configuration (HyperBlock [130]). Third, one PE could express a local control flow if this PE is autonomous, such as in TIA. For nonautonomous CGRAs, such as DySER, the third form is disabled.

In theory, a speculation technique could be applied to all these architecture levels. Figure 9(a) shows a simple control flow to compare their implementation results. Figure 9(b) illustrates the result of applying the speculation technique to the external controller. Assuming that the alternative path B is predicted, CGRAs may only execute the first configuration with A, B and P. If misprediction is detected according to the result of A, then the first configuration is flushed, and the second one with C and P is fetched and executed. Simultaneously, the side effects of the speculative block B in the first configuration should be eliminated. Figure 9(c) illustrates the result of applying the speculation technique to the PE array. The original predicated blocks B and C become speculative and execute in parallel with A. Mis-speculation will always occur either in B or C. The side effects of both B and C should be constrained inside the PE array. The major advantage is the reduction in

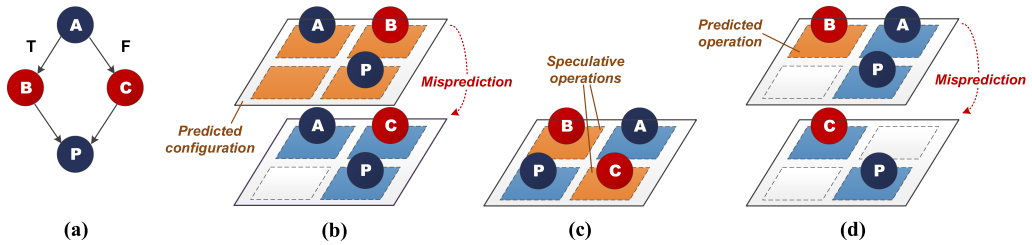

Fig. 9. (a) Forward control flow, (b) speculation at the external controller of the PE array, (c) speculation at PE arrays, and (d) speculation at individual PEs. The operations on orange PE are executed speculatively.

the reconfiguration time. Figure 9(d) illustrates the results of applying the speculation technique to individual PEs in a CGRA with a distributed control scheme. Block B is executed speculatively in parallel with block A. Upon detection of misprediction, only the PE that executes block B is reconfigured. Simultaneously, the side effects of B should be eliminated. Several conclusions can be drawn:

(1) Exploring speculative parallelism in a PE array (Figure 9(c)) is the most profitable approach, because it does not incur any reconfiguration penalty [129, 131]. However, this strategy executes all the alternative paths, which consumes more power and degrades the performance if the hardware resources cannot provide matching parallelism. Additionally, the PE array alone cannot implement backward control dependence (e.g., irregular loops), so the speculative parallelism that can be explored is also limited.
(2) Exploring speculative parallelism in the external controller and in autonomous PEs can minimize the required computation operations if the prediction is always correct.
(3) Considering that the total execution time of the control flow as shown in Figure 9(a) is equal to the sum of the predicted configuration's execution time, the reconfiguration penalty multiplied by the misprediction rate, and the alternative configuration's execution time multiplied by the misprediction rate, exploring speculative parallelism in autonomous PEs can always achieve higher performance than in an external controller, because less reconfiguration penalty is incurred. Of course, if a partial reconfiguration technique is supported, then the reconfiguration penalty would be the same, but the computation of the remaining PEs would be strictly constrained by the control flow, decreasing the PE utilization rate.
(4) Exploring speculative parallelism in a PE array and in autonomous PEs (or in an external controller for CGRAs with a centralized control scheme) is necessary, because neither of these individual approaches can achieve better performance in all circumstances. As discussed previously [132, 133], highly biased and highly predictable branches are preferred in the superblock method, minimally biased and highly predictable branches are preferred in the dynamic prediction method, and minimally biased and minimally predictable branches are preferred in the predication method.
(5) Exploring speculative parallelism at any architecture level requires isolating the speculative operations from the other operations and the external system until the predicate comes out. The possible side effects include memory operations, exceptions and other operations that modify system registers.

In summary, this part has discussed how to design a computation model to maximize the parallelism that can be explored from the upper programming model. Based on the state-of-the-art architecture, the implementation of speculative parallelism is the most important problem to be

addressed. A distributed control scheme that supports multithreads and a comprehensive solution that operates at multiple architecture levels are preferred. It is true that speculative parallelism can improve the performance to a large degree. However, such parallelism should be adapted to different circumstances.

4.3 Trend 3: Virtualization

4.3.1 State-of-the-art Techniques. Virtualization is an extensively discussed topic regarding FPGAs, whereas virtualization for CGRAs is much less studied. A mature CGRA of high productivity should support virtualization, and it should be easily integrated into common heterogeneous computing platforms and operating systems. This is particularly important for the users and developers that are unfamiliar with underlying hardware.

State-of-the-art FPGA virtualization techniques could provide an inspiration for CGRA virtualization. Currently, FPGA virtualization research is addressing two problems that are the keys to integrating FPGAs into operating systems: abstraction and standardization [42]. To resolve these problems, there are three major directions. **(1) Overlays**, especially CGRA overlays**,** could be an effective solution for FPGA abstraction [134, 135]. Overlays or intermediate fabrics improve productivity through providing a layer of programmability that abstract the low-level details of an FPGA. CGRA overlays free users from specialist hardware CAD tool flow and HDLs, which enables agile development several orders faster than the analogous high-level synthesis. CGRA overlays also provide a capability of dynamic partial reconfiguration that is easier and faster than using the CAD tools from vendors. **(2) Virtual hardware processes** that provide an abstract concept that manages the execution and scheduling of hardware resources are another direction. A virtual hardware process is allocated a number of execution units with respect to the currently available resources. The major requirements to which virtual hardware processes are subject includes standard software API, standard hardware interfaces and protocols, and unified execution models. FPGAs could be abstracted as two different execution abstractions for high-level scheduling. First, if the FPGA is used as an accelerator, then it could be controlled as a slave device with a driver. Several papers [136–138] have discussed the standard interface and library design for such FPGA accelerators. The usage as accelerators is common today and is widely supported by the toolchains of commercial FPGA vendors. Second, if the FPGA is used as a parallel computing processor that is equal to the CPU, then this architecture could be abstracted as one or several hardware applications (hw-application), which are independent from software applications and can interact with the software applications by means of communication and synchronization. If the communication is implemented with a message passing mechanism, then the hardware application is often called an hw-process [139]. If the communication is implemented with a coupled memory sharing mechanism, then it is often called an hw-thread. In addition to the execution styles, resource management is another problem with the virtual hardware process. Most FPGAs adopt an island-style architecture, in which one task/process/thread cannot be placed across multiple reconfigurable areas or so-called islands, because it is the easiest for FPGA design flow and facilitates task/process/thread pending and resuming. For a utilization purpose, finer-grid-style architectures have been proposed, in which one thread can occupy several grids from the unused ones [140, 141]. **(3) Standardization** is an open problem that requires a generally accepted unified solution. If standardization could be accomplished in the future, then reconfigurable computing will spread with faster development, better portability and reusability. Fortunately, we have observed that many large industrial companies, including ARM, AMD, Huawei, Qualcomm and Xilinx, are engaging in defining these standards, such as CCIX [142] and the HSA Foundation [143]. Therefore, rapid development of FPGA standardization is likely.

4.3.2 Prospective Directions. CGRA virtualization is still at a primary stage, and it should learn from the state-of-the-art techniques above. (1) The **standardized** software API, hardware interface and protocols are approximately the same as those for FPGAs. Standardization is not difficult but requires broad agreement from researchers and users. (2) Since CGRAs are overlays of FPGAs that are effective in terms of reconfiguration performance, what an effective **overlay for CGRAs** look has not yet been discussed. It is expected that overlays of CGRAs could possess easy but powerful programmability. The computation models discussed in Section 4.2 also provide inspiration. (3) **Virtual hardware processes** are relatively easier to implement on CGRAs, because coarse-grained resources facilitate dynamic scheduling. It is obvious that CGRA virtualization is a problem closely associated with the technical maturity of CGRAs. It might remain controversial until sophisticated CGRAs that are widely accepted by the research and industrial communities are developed. Here, we review current design methods that are valuable for CGRA virtualization.

The execution of CGRAs is somewhat similar to that of FPGAs, because both of them are used for the same purpose of accelerating computation-intensive and data-intensive tasks offloaded from CPUs. Some CGRAs could be used as accelerators with standard software APIs, e.g., special drivers and libraries in forms of configuration contexts. Most CGRAs fall into this category, such as Morphosys, ADRES, and XPP. Some CGRAs, such as TIA and TRIPS, could be used as a parallel computing processor that is equal to a CPU in an operating system. They could be abstracted as hardware applications, which can interact with the CPU by means of standard hardware/software communication and synchronization. Some CGRAs are used as an alternative data path of a CPU, such as DySER and DynaSpAM [53]. These CGRAs could be abstracted at a lower level as an instruction set architecture extension.

The resource scheduling of CGRAs is a little different from that of FPGAs. Since most FPGA designers have agreed on an island-style architecture, no similar architecture templates exist for CGRAs. Some CGRAs support only PE-array-based reconfiguration, as in FPGA islands, such as ADRES. Some CGRAs support PE-line-based reconfiguration, such as PipeRench. Every PE line is a pipeline stage. Some CGRAs support PE-based reconfiguration, such as TIA. Many communication channels exist among PEs, such as shared register files, wire switches and FIFOs. Some CGRAs adopt the static dataflow execution model, in which several PEs can be aggregated into an area to execute a thread/task, such as TRIPs. Other CGRAs adopt the dynamic dataflow execution model, in which multiple threads/tasks can be executed on the same PE area in a time-multiplexing dataflow manner, such as SGMF. Therefore, it is challenging for an operating system to manage the diverse CGRA resources. Of course, there are common characteristics that might facilitate virtualization. The internal communication is conducted mostly in the message-passing mechanism instead of the shared memory model. The mechanism of internal synchronization is typically based on the handshaking and dataflow manner.

Although CGRA virtualization is just emerging, some interesting embryonic ideas can be observed. PipeRench virtualizes pipelined computation via pipelined reconfiguration [69], which makes performing deep-pipelined computation possible even if the entire configuration is never present in the fabric at one time. Tartan can execute an entire general-purpose application with a realistic amount of hardware by exploring a virtualization model [60]. Therefore, PipeRench and Tartan provide virtualizing execution models for their compilers. TFlex is a composable CGRA whose PEs (simple, low-power cores) can be arbitrarily aggregated together to form a larger single-threaded processor. Thus, a thread can be placed on TFlex with its optimal number of PEs, running at its most energy-efficient point. The composability comes mainly from its instruction set architecture (explicit data graph execution). Therefore, TFlex provides a dynamic and efficient PE-level resource management scheme for operating systems. Pager et al. [86] proposed a software scheme for multithreading CGRAs, which transformed a traditional single-threaded CGRA into a

multi-threaded coprocessor for a multi-threaded CPU. This approach can transform the configuration binary at runtime such that this thread occupies fewer pages (a page resembles an island of an FPGA). Therefore, this work provides a dynamic page-level resource management scheme and addresses how to efficiently integrate a CGRA into a multi-threaded embedded system as a multi-threaded accelerator. Overall, current CGRA virtualization techniques mainly target facilitating static compiling, and CGRA virtualization for an operating system and dynamic resource management are forthcoming.

The virtualization is the first and most critical step toward widespread adoption of CGRAs. Further development urgently requires persistent exploitation and cooperation from the CGRA research community.

4.4 Trend 4: Efficient Memory System

4.4.1 State-of-the-art Techniques. In the era of big data, applications are demanding massive computation on very large datasets. Because the computation is performed by a processor while its data are stored in the main memory, massive data have to be transferred between these two locations, applying pressure on the main memory bandwidth. As main memory speed is improved at a much slower pace than processor speed, the bandwidth and access latency are unable to match the processor and thus become the performance bottleneck for many applications, ranging from cloud servers to end-user devices (referred to as the memory wall) [144]. The problem of access latency has been alleviated to a certain extent by multilevel caches with different sizes and speeds. However, the problem of bandwidth is much more challenging.

Compression techniques are a classic method that can alleviate memory bandwidth bottlenecks [145–147]. These works mainly compress the configuration binaries sent to CGRAs offline with common compression algorithms, e.g., Run Length coding, Huffman coding, arithmetic coding and LZSS coding. Thus, the memory bandwidth requirement for configuration decreases. However, this method incurs significant area/power/performance overheads caused by the online decompressor. Moreover, the variable length coding causes inefficiency in configuration storage and loading in parallel. Thus, the compression technique is not common in CGRAs now.

The second solution is to improve the bandwidth of the main memory through integrated circuit technology, such as 3D chip stacking. For instance, AMD launched the first GPU that integrated a 3D-stacked DRAM (HBM), called Fiji, in 2015. Subsequent generations of 3D-stacked DRAM, namely, HBM2 and HBM3, were adopted by many more large companies, such as Samsung, Nvidia and Hynix. The bandwidth of HBM3 could be even greater than 512GB/s. The problem of data starving exists in computation platforms other than the GPUs. Google delivered its first generation of TPU as a domain-specific accelerator (ASIC) for deep learning [92], which used a DDR3 memory with a bandwidth of 30GB/s. It is anticipated that its next-generation TPU2 will use a 64GB HBM2 with a bandwidth of up to 600GB/s.

The third solution to the bandwidth limitation is to eliminate the data transfer between the off-chip memory and processor. In this case, the storage and computation could be integrated together in a single chip and should be coupled in a much tighter manner such that the data movements can be performed on the extended internal channels with small access latency and high energy efficiency. Two approaches can be taken to achieve this condition. One approach is to integrate more memory, such as an embedded DRAM, onto the processor chip as caches, replacing the off-chip main memory to some extent. This approach can alleviate the bandwidth and access latency requirement for off-chip memory. This computation-centric integration suffers from the lower density of eDRAM, however. It is challenging to add enough memory to the processor chip. The other approach is to integrate computation components into memory. This idea was first proposed as intelligent RAM in 1997 [148], which integrated vector processing units for floating point

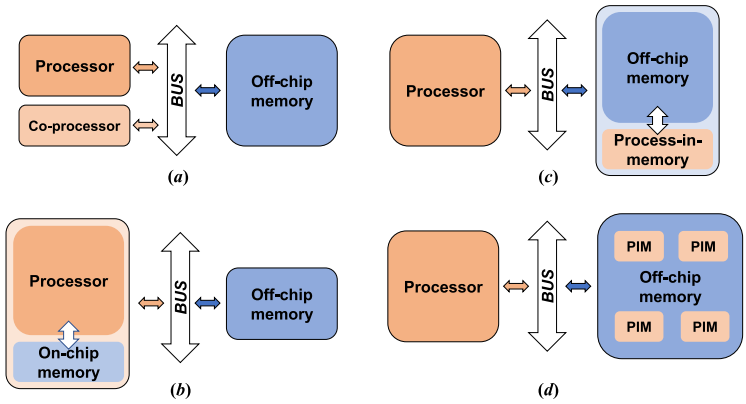

Fig. 10. (a) Conventional processor system, (b) expanding the on-chip memory size, (c) PIM at memory interface, and (d) distributed PIM at memory array.

add, multiply, divide, integer operations, load/store, and multimedia operations on the memory interface. Oskin [149] proposed to distribute FPGA blocks to a DRAM subarray interface. Such process-in-memory (PIM) architecture or compute-capable memory has gained increasing interest. Additionally, in recent years, this approach has become quite popular for data-intensive demands, such as deep learning and bioinformatics. A DRAM-based PIM architecture that integrated an ALU array into the memory interface was proposed in Reference [150]. A DRAM-based PIM architecture called DRISA, proposed in Reference [119], integrated fine-grained logic units into the memory array. A STT-MRAM-based PIM architecture, proposed in Reference [151], modified the memory cell design to support the function of ternary content addressable memories. A ReRAM-based PIM architecture called PRIME, proposed in Reference [152], used the analogous computing feature of memory array. A PRAM-based PIM architecture proposed by IBM [153] also uses the analogous computing feature of the memory array. The 3D chip stacking technique has also been used to mount processing layers on the memory layers to form PIM or near-data processing (NDR) [154, 155].

Figure 10 summarizes the above methods that are intended to bridge the gap of processor and main memory. To alleviate the bandwidth and access latency problem, as shown in Figure 10(a), the first method is to expand the on-chip memory (cache or scratch-pad) size, as shown in Figure 10(b), such that many data movements can be conducted inside the processor chip. The second method is to integrate PIM into the main memory interface, as shown in Figure 10(c), such that the many data movements can be conducted internally on the memory chip. The third method is to integrate PIM into the main memory array as shown in Figure 10(d), making use of the internal bandwidth of the memory array. These approaches are common in offloading some data movements of interchip buses to the more efficient inner-chip channels or 3D-stacking via.

4.4.2 Prospective Directions. The special challenges described above have been partly observed by CGRA researchers. To address them, some effective methods taking different directions have been proposed.

For the problem of fragmented and redundant memory accesses and address generation, one possible solution is to design vectorized or streaming parallel memory interfaces for CGRAs, especially for the domains of stream processing, computer vision and machine learning. There is extensive literature about this solution. DySER, comprising a CGRA tightly coupled with a processor, supports a wide memory interface similar to that used in streaming SIMD extensions (SSEs). The

triggered instruction architecture manages the data input of PEs in a streaming pattern. Plasticine designs programmable address generating units and coalescing units for the memory interface. The stream-dataflow approach combines a static dataflow CGRA with a stream engine, which manages the concurrent accesses to the memory interface and on-chip memory system in a streaming manner. The stream engine is controlled by the stream commands generated by a single-issue in-order processor. As a domain-specific CGRA accelerator for convolutional neural networks, Eyeriss [97] uses a streaming access to fetch data for a dataflow CGRA. However, if there are few opportunities for coalescing fragmented memory accesses into streams or vectors, then the second challenge is encountered.

For the problem of an inflexible memory access pattern, one possible solution is a dynamic CGRA framework that can adapt its memory access patterns at runtime or upon compilation. First, a dynamic dataflow mechanism can explore MLP within the issue window at runtime. The memory operations that incur cache misses could be suspended, and then its following memory operations would be executed. Therefore, more parallelism of memory accesses is exploited at runtime. WaveScalar and SGMF are typical examples. Second, exposing communication or memory accesses explicitly in the program or ISA provides more opportunities for compilers to explore MLP. RSVP [156] exposes streams in the core's ISA to communicate with reconfigurable fabric. Stream-dataflow acceleration also provides an ISA that presents memory streams explicitly. A specialized hardware architecture called memory access dataflow (MAD) with a special ISA, proposed in Reference [157], can be integrated with a different accelerator and boosts the memory accesses that are offloaded from processors. Additionally, some studies have analyzed the data movements and accordingly designed the cache structure and associating mechanism to improve memory efficiency [158].

For the problem of bridging the gap between CGRAs and main memory, a possible solution is constructing a near-memory CGRA fabric with the 3D chip stacking technique. A CGRA can be designed on the logic layer of a 3D stack memory, enabling near-memory reconfigurable computing. The 3D TSVs could provide a wider bus and thus provide larger bandwidth. This technique is also compatible with the HBM and HBC memory technology. There are several works on this front. Gao et al. [79] proposed a heterogeneous reconfigurable logic (HRL) that consists of three main blocks: fine-grained configurable logic blocks for control functions, coarse-grained functional units for basic arithmetic and logic operations, and output multiplexer blocks for branch support. Each memory module employs HMC-like technology. Farmahini et al. [77] proposed coarse-grained reconfigurable arrays on commodity DRAM modules with minimal changes to the DRAM architecture. These works do not yet support direct communication or synchronization between near memory computing stacks.

In summary, because the target domains are becoming increasingly more data intensive, as in the case of machine learning and big data, the memory systems of CGRAs require more improvement and optimization, avoiding the bottleneck in data movements. The directions include the following: (a) improving the regularity and parallelism of the distributed memory accesses through an efficient memory interface, (b) improving the flexibility of the memory interface through a programmable memory-managing unit, and (c) improving the bandwidth and latency of the bus between CGRAs and memory through 3D-stacking chip technology.

5 FUTURE DEVELOPMENT OF ARCHITECTURES AND APPLICATIONS

Since general-purpose processors have stopped their rapid increase in performance but the demand for massive data processing capability continues to increase, application-specific accelerators are becoming popular. CGRAs have unique advantages of flexibility, performance and energy efficiency relative to other accelerators, and thus it is expected to play a key role in future

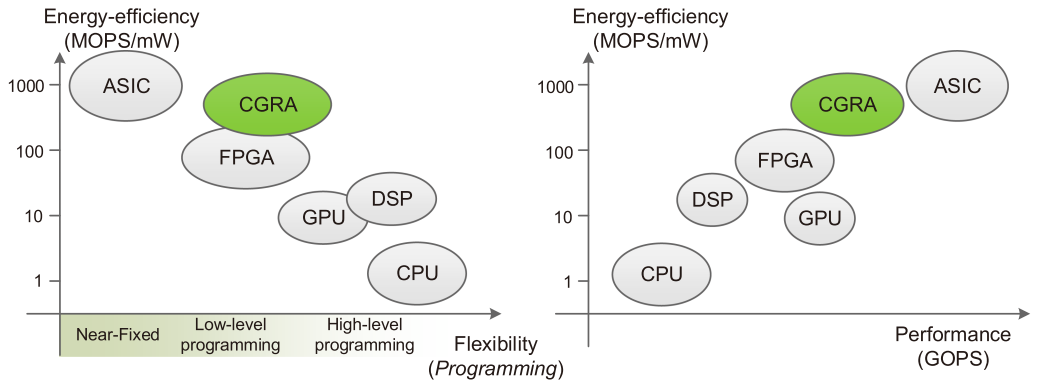

Fig. 11. (a) Architecture comparison in terms of flexibility and energy efficiency and (b) application demand analysis in terms of repurposing and power budget (note that this figure is qualitative and considers general cases).

computing systems. This section provides analyses and predictions regarding the future trends of CGRA architectures and applications.

5.1 Prospective Evolutions of Architectures

The prospective evolutions of CGRA architectures have been partially discussed in Section 4 in terms of the possible solutions to different challenges. This part presents a summary and some extensive discussion.

(1) We argue that CGRA architectures should comply with a popular programming language, which might be concurrent or declarative. The architecture details beyond the programming paradigm should be mostly transparent to users for the purpose of alleviating the burdens on compilers and programmers.
(2) We argue that speculative execution technique should be introduced into CGRAs for performance improvement across more applications. The dataflow mechanism could be coupled with speculative methods, improving the effectiveness of both.
(3) We argue that virtualization and composability shall become essential ingredients for CGRAs. A uniform software/hardware interface and protocols are desired.
(4) We argue that process-in-memory and computation memory should be introduced into CGRA architectures to address the bottleneck of data movement.
(5) We argue that programmable memory management is becoming increasingly important as data-intensive applications, e.g., deep learning and big data, gain momentum. A CGRA architecture with flexible memory system could optimize for various access patterns.
(6) We argue that dynamic compilation hardware that implements runtime optimization for data flow is an important direction for emerging applications.

5.2 Prospective Evolutions of Applications

This part analyzes the opportunities in future to determine possible killer applications for CGRAs. As shown in Figure 11(a), CGRAs are promising computing fabric with enormous potential. They can achieve an energy efficiency within 10× that of ASICs and can be programmed with high-level languages. As a reconfigurable computing fabric, CGRAs avoid two major weaknesses of FPGAs: low performance because of long interconnection and low efficiency because of low hardware

utilization [6]. As a result, CGRAs are a superior alternative to FPGAs in accelerating applications. However, there is a lack of killer applications for CGRAs.

The killer applications of CGRAs probably lie in the domains that have higher requirements in terms of both power and flexibility. Figure 11(b) lists popular application scenarios and their various requirements in terms of the power budget and repurpose frequency during their lifecycle. A comparison with Figure 11(a) indicates that CGRAs well match the scenarios on the top right, such as datacenters, the mobile end, cloud servers and robotics. For instance, Microsoft's Catapult project uses FPGAs to speed up its Bing search engine in their datacenters, leading to significant improvements while retaining the ability to change and improve the algorithm over time within the system [159]. IBM, Intel, and Baidu have also made similar decisions [160, 161], integrating FPGAs into their datacenters and cloud servers, and this scheme is becoming more popular. We argue that CGRAs, as a more efficient alternative to FPGAs, will find their place in the domain of datacenters and servers for big data applications and network processing. Mobile end and IoT scenarios are also promising domains for CGRAs, as these have strict requirements on power consumption and function updating for a wide range of applications, including AI, signal and image processing, and security.

In addition to energy efficiency and flexibility, other characteristics of CGRAs could be utilized to enable killer applications. CGRAs typically provide massive runtime reconfigurable resources, which typically cannot be fully used. Therefore, redundant resources at runtime could be employed for reliability and security use. First, redundant resources could replace the faulty ones by transforming the mapping result equivalently. Thus, CGRAs can provide fault-tolerant designs against a low-yield process and physical failures [162–165], making it suitable for applications that have a high requirement for fault tolerance, such as aerospace and high-end servers. Second, the redundant resources can implement security check modules to avoid physical attacks and hardware trojans [166]. Countermeasures against physical attacks for reconfigurable cryptographic processors were proposed in References [167, 168]; these used the characteristics of partial reconfiguration and time-multiplexed SRAM to improve the security. The method proposed in Reference [169] utilizes spatial and temporal randomizing reconfiguration as countermeasures for double fault attacks. A method based on the Benes network was proposed in Reference [170] as a random infection countermeasure for block ciphers against fault attacks.

6 CONCLUSION

CGRAs have become a viable alternative to existing computing architectures. As the circuit technology continues to scale down, the advantages of CGRAs in terms of power and flexibility have become even more significant. Therefore, CGRAs are expected to play an important role in mainstream computing infrastructures.

This survey presents a comprehensive survey regarding CGRAs. First, a novel multidimensional taxonomy is proposed by abstracting the programming, computation and execution paradigm of CGRAs. Second, the major challenges regarding the architecture and design of CGRAs are summarized and discussed. Third, the application opportunities for CGRAs in the future are analyzed. We believe that these challenges can be addressed rapidly with the cooperation of academia and industry. Then, CGRAs could be a widespread solution to the problem of performance, energy efficiency and flexibility beyond Moore's Law.

REFERENCES

[1] G. E. Moore. 2002. Cramming more components onto integrated circuits. *Proc. IEEE* 86, 1 (2002), 82–85.
[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circ.* 9, 5 (1974), 256–268.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro* 32, 3 122–134.
[4] R. Courtland. 2013. The end of the shrink. *IEEE Spectrum* 50, 11 (2013), 26–29.
[5] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright. 2016. Pushing the limits of accelerator efficiency while retaining programmability. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 27–39.
[6] M. Duranton, K. D. Bosschere, C. Gamrat, J. Maebe, H. Munk, and O. Zendra. 2017. The HiPEAC Vision 2017. In *Proceedings of the European Network of Excellence on High Performance and Embedded Architecture and Compilation*. 12.
[7] G. Estrin. 1960. Organization of computer systems-the fixed plus variable structure computer. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*. 33–40.
[8] R. W. Hartenstein, A. G. Hirschbiel, M. Riedmuller, and K. Schmidt. 1991. A novel ASIC design approach based on a new machine paradigm. *IEEE J. Solid-State Circ.* 26, 7 (1991), 975–989.
[9] D. C. Chen and J. M. Rabaey. 1992. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. *IEEE J. Solid-State Circ.* 27, 12 (1992), 1895–1904.
[10] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of the International Conference on Field Programmable Logic and Application (FPL'03)*. 61–70.
[11] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* 49, 5 (2000), 465–481.
[12] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. Mcdonald, and C. R. Moore. 2004. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Trans. Architect. Code Optimiz.* 1, 1 (2004), 62–93.
[13] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. 2003. PACT XPP—A self-reconfigurable data processing architecture. *J. Supercomput.* 26, 2 (2003), 167–184.
[14] M. Horowitz. 2014. 1.1 computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-state Circuits Conference (ISSCC'14)*. IEEE, 10–14.
[15] G. Theodoridis, D. Soudris, and S. Vassiliadis. 2007. A survey of coarse-grain reconfigurable architectures and cad tools. In *Fine-and Coarse-Grain Reconfigurable Computing*. Springer, Dordrecht, 89–149.
[16] L. Liu, Z. Li, Y. Chen, C. Deng, S. Yin, and S. Wei. 2017. HReA: An energy-efficient embedded dynamically reconfigurable fabric for 13-dwarfs processing. *IEEE Trans. Circ. Syst. II Express Briefs* 65, 3 (2017), 381–385.
[17] L. Liu, D. Wang, M. Zhu, Y. Wang, S. Yin, P. Cao, J. Yang, and S. Wei. 2015. An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. *IEEE Trans. Multimedia* 17, 10 (2015), 1706–1720.
[18] C. Nicol. 2017. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. *Wave Computing White Paper*.
[19] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. 2017. Plasticine: A Reconfigurable Architecture for Parallel Paterns. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 389–402.
[20] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 416–429.
[21] DARPA. 2017. Retrieved from https://www.darpa.mil/.
[22] S. Kim, Y. H. Park, J. Kim, M. Kim, W. Lee, and S. Lee. 2016. Flexible video processing platform for 8K UHD TV. In *Proceedings of the Hot Chips 27 Symposium*. 1–1.
[23] Samsung. 2014. Retrieved from http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5430/.
[24] PACT. Retrieved from http://www.pactxpp.com/.
[25] Intel. 2016. Retrieved from https://newsroom.intel.com/news-releases/intel-tsinghua-university-and-montage-technology-collaborate-to-bring-indigenous-data-center-solutions-to-china/.
[26] M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and Others. 2004. Stream applications on the dynamically reconfigurable processor. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*. IEEE, 137–144.
[27] T. Sato, H. Watanabe, and K. Shiba. 2005. Implementation of dynamically reconfigurable processor DAPDNA-2. In *Proceedings of the IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test (VLSI-TSA-DAT)*. 323–324.
[28] B. Mei, B. Sutter, T. Aa, M. Wouters, A. Kanstein, and S. Dupont. 2008. Implementation of a coarse-grained reconfigurable media processor for avc decoder. *J. Signal Process. Syst.* 51, 3 (2008), 225–243.

[29] M. K. A. Ganesan, S. Singh, F. May, and J. Becker. 2007. H. 264 Decoder at HD resolution on a coarse grain dynamically reconfigurable architecture. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 467–471.
[30] L. Liu, C. Deng, D. Wang, and M. Zhu. 2013. An energy-efficient coarse-grained dynamically reconfigurable fabric for multiple-standard video decoding applications. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 1–4.
[31] A. H. Veen. 1986. Dataflow machine architecture. *ACM Comput. Surveys* 18, 4 (1986) 365–396.
[32] G. Gupta and G. S. Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. ACM, 59–70.
[33] H. Tseng and D. M. Tullsen. 2011. Data-triggered threads: Eliminating redundant computation. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 181–192.
[34] R. Hartenstein. 2001. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 642–649.
[35] R. Tessier, K. L. Pocek, and A. Dehon. 2015. Reconfigurable computing architectures. *Proc. IEEE*. 103, 3 (2015), 332–354.
[36] M. Wijtvliet, L. Waeijen, and H. Corporaal. 2016. Coarse-grained reconfigurable architectures in the past 25 years: Overview and classification. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS'16)*. 235–244.
[37] H. Amano. 2006. A survey on dynamically reconfigurable processors. *IEICE Trans Commun*. 89, 12 (2006), 3179–3187.
[38] J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt. 2010. Compiling for reconfigurable computing: A survey. *ACM Comput. Surveys* 42, 4 (2010), 1–65.
[39] Zain-ul-Abdin and B. Svensson. 2009. Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing. *Microprocess. Microsyst.* 22, 3 (2009), 161–178.
[40] A. Dehon. 2015. Fundamental underpinnings of reconfigurable computing architectures. *Proc. IEEE*. 103, 3 (2015), 355–378.
[41] A. Chattopadhyay. 2013. Ingredients of adaptability: A survey of reconfigurable processors. *VLSI Design* 2013, 10.
[42] M. Eckert, D. Meyer, J. Haase, and B. Klauer. 2016. Operating system concepts for reconfigurable computing: Review and survey. *Int. J. Reconfig. Comput.* 2016, 1–11.
[43] M. Gokhale and P. S. Graham. 2007. Reconfigurable computing systems. *Proc. IEEE* 90, 7 (2007), 1201–1217.
[44] K. Compton and S. Hauck. 2002. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surveys* 34, 2 (2002), 171–210.
[45] T. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. 2005. Reconfigurable computing: architectures and design methods. *IEE Proc. Comput. Dig. Techn.* 152, 2 (2005), 193–207.
[46] A. Dehon, J. Adams, M. Delorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. C. Vanier, and M. G. Wrighton. 2004. Design patterns for reconfigurable computing. In *Proceedings of the IEEE Symposium on Field-programmable Custom Computing Machines*. 13–23.
[47] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.
[48] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. 1995. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, vol. 2. 61–70.
[49] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams. 2006. The landscape of parallel computing research: A view from berkeley. EECS Department, University of California, Berkeley.
[50] H. Dutta, D. Kissler, F. Hannig, A. Kupriyanov, J. Teich, and B. Pottier. 2009. A holistic approach for tightly coupled reconfigurable parallel processors. *Microprocess. Microsyst.* 33, 1 (2009), 53–62.
[51] M. A. Watkins, T. Nowatzki, and A. Carno. 2016. Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'16)*. 138–150.
[52] N. Clark, M. Kudlur, H. Park, and S. Mahlke. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the International Symposium on Microarchitecture*. 30–40.
[53] F. Liu, H. Ahn, S. R. Beard, and T. Oh. 2015. DynaSpAM: Dynamic spatial architecture mapping using Out of Order instruction schedules. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 541–553.
[54] H. Park, Y. Park, and S. Mahlke. 2009. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 370–380.
[55] M. J. Flynn. 2009. Some computer organizations and their effectiveness. *IEEE Trans. Comput.* C-21(9), 948–960.

[56] M. Budiu and S. C. Goldstein. 2002. Pegasus: An efficient intermediate representation. Computer Science Department, Carnegie Mellon University.
[57] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. 2004. Spatial computation. *ACM SIGPLAN Notices* 39(11), 14–26.
[58] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM.* 21, 1 (1978), 666–677.
[59] G. Kahn. 1974. The semantics of simple language for parallel programming. *Info. Process.* 74, 471–475.
[60] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. 2006. Tartan: Evaluating spatial computation for whole program execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 163–174.
[61] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. C. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, and A. Jaleel. 2013. Triggered instructions: A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, Vol. 41. 142–153.
[62] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. 2007. The WaveScalar architecture. *ACM Trans. Comput. Syst.* 25, 2 (2007), 4.
[63] J. B. Dennis and D. P. Misunas. 1974. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Comput. Architect. News* 3, 4 (1974), 126–132.
[64] I. Watson and J. Gurd. 1979. A prototype data flow computer with token labelling. In *Proceedings of the National Computer Conference*. 623–628.
[65] A. Null and V. C. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318.
[66] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. 2007. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *Proceedings of the International Workshop on Applied Reconfigurable Computing*. 1–13.
[67] A. K. W. Yeung and J. M. Rabaey. 1993. A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms. In *Proceeding of the 26th Hawaii International Conference on System Sciences*, Vol. 1. 169–178.
[68] E. Waingold, M. Taylor, D. Srikrishna, and V. Sarkar. 1997. Baring it all to software: Raw machines. *Computer* 30, 9 (1997), 86–93.
[69] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. 2000. PipeRench: A reconfigurable architecture and compiler. *IEEE Comput.* 33, 4 (2000), 70–77.
[70] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. 2003. WaveScalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. 291.
[71] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. 2007. Composable Lightweight Processors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 381–394.
[72] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan. 2008. The reconfigurable instruction cell array. *IEEE Trans. Very Large Scale Integr. Syst.* 16, 1 (2008), 75–85.
[73] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugomartinez, S. Swanson, and M. B. Taylor. 2010. Conservation cores: Reducing the energy of mature computations. *ACM SIGARCH Comput. Architect. News* 38, 1 205–218.
[74] B. Robatmili, D. Li, H. Esmaeilzadeh, S. Govindan, A. Smith, A. Putnam, D. Burger, and S. W. Keckler. 2013. How to implement effective prediction and forwarding for fusable dynamic multicore architectures. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 460–471.
[75] D. Voitsechov and Y. Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. In *Proceeding of the International Symposium on Computer Architecture*. 205–216.
[76] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*. 9–16.
[77] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'15)*. 283–295.
[78] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck. 2016. A reconfigurable heterogeneous multicore with a homogeneous ISA. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. 1598–1603.
[79] M. Gao and C. Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 126–137.
[80] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *Proceedings of the IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP'17)*. 184–189.

[81] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique. 2018. PX-CGRA: Polymorphic approximate coarse-grained reconfigurable architecture. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'18)*. 413–418.
[82] L. Duch, S. Basu, M. Peón-Quirós, G. Ansaloni, L. Pozzi, and D. Atienza. 2019. i-DPs CGRA: An interleaved-datapaths reconfigurable accelerator for embedded bio-signal processing. *IEEE Embedded Systems Letters* 11, 2 (2019), 50–53.
[83] T. Chen, S. Srinath, C. Batten, and G. E. Suh. 2018. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. 55–67.
[84] D. Voitsechov, O. Port, and Y. Etsion. 2018. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. 42–54.
[85] S. A. Chin and J. H. Anderson. 2018. An architecture-agnostic integer linear programming approach to CGRA mapping. In *Proceedings of the ACM/ESDA/IEEE Design Automation Conference (DAC'18)*. 1–6.
[86] J. Pager, R. Jeyapaul, and A. Shrivastava. 2015. A software scheme for multithreading on CGRAs. *ACM Trans. Embed. Comput. Syst.* 14, 1 (2015), 1–26.
[87] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and Others. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 1–12.
[88] A. J. Elbirt and C. Paar. 2005. An instruction-level distributed processor for symmetric-key cryptography. *IEEE Trans. Parallel Distrib. Syst.* 16, 5 (2005), 468–480.
[89] D. Fronte, A. Perez, and E. Payrat. 2008. Celator: A multi-algorithm cryptographic co-processor. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*. 438–443.
[90] G. Sayilar and D. Chiou. 2014. Cryptoraptor: High throughput reconfigurable cryptographic processor. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'14)*. 155–161.
[91] B. Mei, F. J. Veredas, and B. Masschelein. 2005. Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 622–625.
[92] M. Hartmann, V. Pantazis, T. V. Aa, M. Berekovic, and C. Hochberger. 2010. Still image processing on coarse-grained reconfigurable array architectures. *J. Signal Process. Syst.* 60, 2 (2010), 225–237.
[93] D. Novo, W. Moffat, V. Derudder, and B. Bougard. 2005. Mapping a multiple antenna SDM-OFDM receiver on the ADRES coarse-grained reconfigurable processor. In *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation*. 473–478.
[94] M. Palkovic, H. Cappelle, M. Glassee, B. Bougard, and D. P. L. Van. 2008. Mapping of 40 MHz MIMO SDM-OFDM baseband processing on multi-processor SDR platform. In *Proceedings of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*. 1–6.
[95] X. Chen, A. Minwegen, Y. Hassan, D. Kammler, S. Li, T. Kempf, A. Chattopadhyay, and G. Ascheid. 2012. FLEXDET: Flexible, efficient multi-Mode MIMO detection using reconfigurable ASIP. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*. 69–76.
[96] G. Kappen and T. G. Noll. 2006. Application specific instruction processor-based implementation of a GNSS receiver on an FPGA. In *Proceedings of the Design, Automation and Test in Europe (DATE'06)*. 6.
[97] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circ.* 52, 1 (2017), 127–138.
[98] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. 2017. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Trans. Very Large Scale Integr. Syst.* 25, 8 (2017), 2220–2233.
[99] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, and S. Wei. 2017. A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications. In *Proceedings of the Symposium on VLSI Circuits. C26-C27*.
[100] C. Farabet, B. Martini, B. Corda, and P. Akselrod. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the Computer Vision and Pattern Recognition Workshops*. 109–116.
[101] B. Li, K. Tan, L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, E. Chen, and E. Chen. 2016. ClickNP: Highly Flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the Conference on ACM SIGCOMM 2016 Conference*. 1–14.
[102] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. A. Vissers, and Z. Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 30, 4 (2011), 473–491.
[103] Xilinx. Retrieved from http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[104] Intel. Retrieved from https://software.intel.com/en-us/intel-opencl.
[105] Z. Wang, B. He, W. Zhang, and S. Jiang. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *Proceedings of the IEEE International Symposium on High PERFORMANCE Computer Architecture*. 114–125.

[106] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru. 2007. Performance characterization of SPEC CPU benchmarks on Intel's core microarchitecture-based processor. In *Proceedings of the Spec Benchmark Workshop*. 1–7.

[107] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, and X. Tian. 2008. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the Intel Core 2 Duo processor. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 132–141.

[108] V. Packirisamy, A. Zhai, W. C. Hsu, and P. C. Yew. 2010. Exploring speculative parallelism in SPEC2006. In *Proceedings of the IEEE International Symposium on PERFORMANCE Analysis of Systems and Software*. 77–88.

[109] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz. 2015. Convolution engine: Balancing efficiency and flexibility in specialized computing. *Commun. ACM* 58, 4 (2015), 85–93.

[110] M. Brandalero, A. C. S. Beck, L. Carro, and M. Shafique. 2018. Approximate on-the-fly coarse-grained reconfigurable acceleration for general-purpose applications. In *Proceedings of the ACM/ESDA/IEEE Design Automation Conference*. 1–6.

[111] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (2011), 7–17.

[112] Y. Chen, Y. Chen, Y. Chen, Y. Chen, Y. Chen, Y. Chen, and O. Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 269–284.

[113] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. 2017. DRISA: A DRAM-based reconfigurable *in situ* accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 288–301.

[114] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.

[115] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid. 2008. A design flow for architecture exploration and implementation of partially reconfigurable processors. *IEEE Trans. Very Large Scale Integr. Syst.* 16, 10 (2008), 1281–1294.

[116] T. Stripf, R. Koenig, and J. Becker. 2011. A novel ADL-based compiler-centric software framework for reconfigurable mixed-ISA processors. In *Proceedings of the International Conference on Embedded Computer Systems*. 157–164.

[117] R. Leupers. 2008. LISA: A uniform ADL for embedded processor modeling, implementation, and software toolsuite generation. In *Processor Description Languages*. Elsevier, Vol. 1. 95–130.

[118] D. Suh, K. Kwon, S. Kim, and S. Ryu. 2012. Design space exploration and implementation of a high performance and low area coarse-grained reconfigurable processor. In *Proceedings of the International Conference on Field-Programmable Technology*. 67–70.

[119] Y. Kim, R. N. Mahapatra, and K. Choi. 2010. Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 10 (2010), 1471–1482.

[120] N. George, H. J. Lee, D. Novo, and T. Rompf. 2014. Hardware system synthesis from domain-specific languages. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–8.

[121] R. Prabhakar, D. Koeplinger, K. J. Brown, H. J. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun. 2015. Generating configurable hardware from parallel patterns. *ACM SIGPLAN Notices* 50, 2 (2015), 651–665.

[122] D. Koeplinger, C. Delimitrou, R. Prabhakar, C. Kozyrakis, Y. Zhang, and K. Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 115–127.

[123] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei. 2017. Aggressive pipelining of irregular applications on reconfigurable hardware. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 575–586.

[124] B. R. Rau and C. D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming (MICRO'81)*. 183–198.

[125] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Design, Automation and Test in Europe Conference and Exhibition*. 296–301.

[126] M. Hamzeh, A. Shrivastava, and S. Vrudhula. 2012. EPIMap: Using epimorphism to map applications on CGRAs. In *Proceedings of the Design Automation Conference*. 1284–1291.

[127] M. Hamzeh, A. Shrivastava, and S. Vrudhula. 2013. REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *Proceedings of the Design Automation Conference*. 1–10.

[128] S. A. Mahlke, R. E. Hank, J. E. Mccormick, D. I. August, and W. M. W. Hwu. 1995. A comparison of full and partial predicated execution support for ILP processors. *ACM/SIGARCH Comput. Architect. News* 23, 2 (1995), 138–150.

[129] K. Chang and K. Choi. 2009. Mapping control intensive kernels onto coarse-grained reconfigurable array architecture. In *Proceedings of the International SoC Design Conference*. I-362-I-365.

[130] S. A. Mahlke, D. C. Lin, W. Y. Chen, and R. E. Hank. 1995. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the International Symposium on Microarchitecture*. 45–54.

[131] G. Lee, K. Chang, and K. Choi. 2010. Automatic mapping of control-intensive kernels onto coarse-grained reconfigurable array architecture with speculative execution. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. 1–4.

[132] D. S. Mcfarlin and C. Zilles. 2015. Branch vanguard: Decomposing branch functionality into prediction and resolution instructions. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 323–335.

[133] J. Wang, L. Liu, J. Zhu, S. Yin, and S. Wei. 2015. Acceleration of control flows on reconfigurable architecture with a composite method. In *Proceedings of the Design Automation Conference*. 45.

[134] A. K. Jain, D. L. Maskell, and S. A. Fahmy. 2016. Are coarse-grained overlays ready for general purpose application acceleration on fpgas? In *Proceedings of the IEEE 14th International Conference on Dependable, Autonomic, and Secure Computing, 14th International Conference on Pervasive Intelligence and Computing, and 2nd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech'16)*. 586–593.

[135] C. Liu, H. Ng, and H. K. So. 2015. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *Proceedings of the International Conference on Field Programmable Technology (FPT'15)*. 56–63.

[136] J. H. Kelm and S. S. Lumetta. 2008. HybridOS: Runtime support for reconfigurable accelerators. In *Proceedings of the International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. 212–221.

[137] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. 2011. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 25–28.

[138] K. Fleming and M. Adler. 2016. The LEAP FPGA operating system. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'16)*. 1–8.

[139] K. H. So and R. W. Brodersen. 2007. BORPH: An operating system for FPGA-based reconfigurable computers. University of California, Berkeley.

[140] F. Redaelli, M. D. Santambrogio, and S. Ogrenci Memik. 2008. An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*. 97–102.

[141] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada. 2011. Rainbow: An OS extension for hardware multitasking on dynamically partially reconfigurable FPGAs. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*. 416–421.

[142] CCIX Consortium. 2019. Retrieved from http://www.ccixconsortium.com/.

[143] HSA Foundation. 2019. Retrieved from http://www.hsafoundation.com/.

[144] D. Burger, J. R. Goodman, and A. Kägi. 2005. Memory bandwidth limitations of future microprocessors. In *Proceedings of the International Symposium on Computer Architecture*. 78–89.

[145] S. M. Jafri, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen. 2011. Compression-based efficient and agile configuration mechanism for coarse-grained reconfigurable architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. 290–293.

[146] Y. Kim and R. N. Mahapatra. 2010. Dynamic context compression for low-power coarse-grained reconfigurable architecture. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 1 (2010), 15–28.

[147] M. Suzuki, Y. Hasegawa, V. M. Tuan, S. Abe, and H. Amano. 2006. A cost-effective context memory structure for dynamically reconfigurable processors. In *Proceedings of the IEEE International Parallel \& Distributed Processing Symposium*.

[148] D. A. Patterson, T. E. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. E. Kozyrakis, R. Thomas, and K. A. Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (1997), 34–44.

[149] M. Oskin, F. T Chong, and T. Sherwood. 1998. Active Pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 192–203.

[150] J. Lee, J. H Ahn, and K. Choi. 2016. Buffered compares: Excavating the hidden parallelism inside DRAM architectures with lightweight logic. In *Proceedings of the Design, Automation & Test in Europe Conference and Exhibition (DATE'16)*. 1243–1248.

[151] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman. 2013. AC-DIMM: Associative computing with STT-MRAM. In *Proceedings of the International Symposium on Computer Architecture*. 189–200.

[152] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the International Symposium on Computer Architecture*. 27–39.

[153] A. Sebastian, T. Tuma, N. Papandreou, M. L. Gallo, L. Kull, T. Parnell, and E. Eleftheriou. 2017. Temporal correlation detection using computational phase-change memory. *Nature Commun.* 8, 1 (2017), 1115.

[154] B. Akin, F. Franchetti, and J. C. Hoe. 2015. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 131–143.

[155] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Comput. Architect. News* 43, 3 (2015), 105–117.

[156] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. 2003. The reconfigurable streaming vector processor (RSVP™). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. 141–150.

[157] C. Ho, S. J. Kim, and K. Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 118–130.

[158] P. A. Tsai, N. Beckmann, and D. Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 652–665.

[159] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, and J. Gray. 2015. A Reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro* 35, 3 (2015), 10–22.

[160] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang. 2016. SDA: Software-defined accelerator for large-scale DNN systems. In *Proceedings of the Hot Chips 26th Symposium*. 1–23.

[161] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. 2014. Enabling FPGAs in the cloud. In *Proceedings of the Conference on Computing Frontiers*. 3.

[162] L. Liu, Y. Ren, C. Deng, S. Yin, S. Wei, and J. Han. 2015. A novel approach using a minimum cost maximum flow algorithm for fault-tolerant topology reconfiguration in NoC architectures. In *Proceedings of the Design Automation Conference*. 48–53.

[163] D. Alnajiar, Y. Ko, T. Imagawa, and H. Konoura. 2010. Coarse-grained dynamically reconfigurable architecture with flexible reliability. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 186–192.

[164] T. Imagawa, H. Tsutsui, H. Ochi, and T. Sato. 2013. A cost-effective selective TMR for heterogeneous coarse-grained reconfigurable architectures based on DFG-level vulnerability analysis. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*. 701–706.

[165] D. Alnajjar, H. Konoura, Y. Ko, Y. Mitsuyama, M. Hashimoto, and T. Onoye. 2013. Implementing flexible reliability in a coarse-grained reconfigurable architecture. *IEEE Trans. Very Large Scale Integr. Syst.* 21, 12 (2013), 2165–2178.

[166] L. Liu, Z. Zhou, S. Wei, M. Zhu, S. Yin, and S. Mao. 2017. DRMaSV: Enhanced capability against hardware trojans in coarse-grained reconfigurable architectures. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 37, 4 (2017), 782–795.

[167] N. Mentens, B. Gierlichs, and I. Verbauwhede. 2008. Power and fault analysis resistance in hardware through dynamic reconfiguration. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. 346–362.

[168] R. Beat, P. Grabher, D. Page, S. Tillich, and M. Wojcik. 2012. On reconfigurable fabrics and generic side-channel countermeasures. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 663–666.

[169] B. Wang, L. Liu, C. Deng, M. Zhu, S. Yin, and S. Wei. 2016. Against double fault attacks: Injection effort model, space and time randomization-based countermeasures for reconfigurable array architecture. *IEEE Trans. Info. Forens. Secur.* 11, 6 (2016), 1151–1164.

[170] B. Wang, L. Liu, C. Deng, M. Zhu, S. Yin, Z. Zhou, and S. Wei. 2017. Exploration of Benes network in cryptographic processors: A random infection countermeasure for block ciphers against fault attacks. *IEEE Trans. Info. Forens. Secur.* 12, 2 (2017), 309–322.

Received April 2018; revised May 2019; accepted August 2019