A Partially Shared Thin Reconfigurable Array For Multicore Processor

Francisco Carlos Silva Junior
Universidade de Brasília (UnB)
Brasília, Brazil
juninho.ufpi@hotmail.com

Ivan Saraiva Silva
Universidade Federal do Piauí (UFPI)
Teresina, Brazil
ivan@ufpi.edu.br

Ricardo Pezzuol Jacobi
Universidade de Brasília (UnB)
Brasília, Brazil
jacobi@unb.br

***Abstract*—Reconfigurable architectures have been widely used as single-core processor accelerators. In the multi-core era, however, it is necessary to review the way that reconfigurable arrays are integrated into a multi-core processor. Generally, a set of reconfigurable functional units are employed similarly as they are used in single-core processors. Unfortunately, a considerable increase in the area ensues from this practice. Besides, in applications with unbalanced workload in their threads this approach can lead to inefficient usage of the reconfigurable architecture in cores with a low or even idle workload. To cope with this issue, this work proposes and evaluates a partially shared thin reconfigurable array, which allows sharing reconfigurable resources among the processor's cores. Sharing is performed dynamically by the configuration scheduler hardware. The results show that the sharing mechanism provided 76% of energy savings, improving the performance 41% on average when compared with a version without the proposed reconfigurable array. A comparison with a version of the reconfigurable array without the sharing mechanism was performed and shows that the sharing mechanism improved up to 11.16% in the system performance.**

***Index Terms*—Reconfigurable architecture, CGRA, Multi-core processor, Resource sharing, Binary translator.**

I. Introduction

Over the last years, increases in application complexity have been demanding more and more performance from processors. Two different approaches have been followed for such applications since the advent of integrated circuit development technologies. On the first approach, General Purpose Processors (GPPs) are used for the execution of any application. On the second, dedicated hardware or Application Specific Integrated Circuits (ASIC) are used for the execution of specific applications. For the latter, the performance gain is achieved through the specialization of the hardware structures. GPPs are flexible enough to run any application, however, they cannot supply enough performance to handle the increasing complexity of the latest applications. In contrast, ASIC offers high performance but poor flexibility, due to their hardware structure be specific to the application they were designed.

Reconfigurable Architectures (RA) emerged as an architectural solution to provide higher flexibility than ASIC and better performance than GPP. RAs are generally composed of a set of Reconfigurable Functional Units (RFU) or Processing Elements (PE), linked by an interconnection system.

Coarse-grained reconfigurable architectures (CGRA) have been successfully used as accelerators in single-core processors, providing energy savings and faster execution [1]. However, multi-core processors are dominant in the processor industry nowadays. This way, it is necessary to review how the reconfigurable architectures are integrated into the system.

A straightforward approach to integrate CGRA into multi-core processors consists of coupling each core with its own CGRA [2][3]. There are two main drawbacks in this approach: *i)* The CGRAs found in the literature usually have many processing elements and to replicate the CGRA to each core could have a significant cost in the area to the system; *ii)* In multi-core processors the application threads will not always present a balanced workload. In this case, it will lead to an inefficient usage of the reconfigurable resources in cores where the CGRA is underused or even completely idle.

To cope with these issues we propose and evaluate a partially shared thin reconfigurable architecture for multi-core processors. The thinness concept consists of coupling to each core a CGRA composed of three PEs and a single Load/Store unit. This set of resources is called *reconfigurable column*. There is one reconfigurable column for each processor core, but the column resources are not restricted to the core they are coupled and can be shared by other cores. Thus, by sharing the columns' resources among cores, the reconfigurable architecture is more efficiently integrated into the multi-core processor, saving area and power.

In order to evaluate the performance, we used a benchmark with four applications, comparing the results obtained with and without our reconfigurable architecture. They show a maximum speedup of 44.91% while saving up to 83% of energy.

The paper is organized in 5 sections. Section II presents related works and points out this study's contribution. Section III describes the system proposed in this study. Section IV presents the performance and energy results. The paper concludes with section V, presenting conclusions and future works.

II. Related Work

Many reconfigurable architectures have been proposed in the literature. Most of them are attached to a single-core processor, such as Chimaera [6], Morphosys [7], PipeRench

[8] and many other architectures that can be found in survey papers [9] [10]. In this work, we focus on the proposals that address the usage of reconfigurable architectures in multi-core processors.

Concerning how the resources of the reconfigurable architecture are distributed into a multi-core processor, there are two types of CGRA organization: homogeneous and heterogeneous. In the first organization, the reconfigurable resources are the same to all cores [11][2]. In the latter organization, the number of reconfigurable resources may be different for each core[3].

CReAMS [2] is a homogeneous CGRA for multi-core processor. It proposes a multi-core system where each core is a DAP (Dynamic Adaptative Processor). The DAP is a transparent reconfigurable architecture for single-thread applications proposed in [13]. To provide transparent reconfiguration, each DAP has a binary translator attached to the fetch stage of the pipeline processor and translates, at run-time, the instructions executed in the processor to run in the CGRA. The reconfigurable architecture is implemented using combinational logic and can execute up to three dependent operations in a single cycle.

HARTMP [3], as in [2], also uses DAPs to compose a multi-core system. However, unlike CREaMS, HARTMP has heterogeneous organization with distinct numbers of resource that compose the reconfigurable architecture in each core. This architecture aims to offer efficient execution of applications with distinct thread load balance.

In order to make the usage of reconfigurable architecture more power and area efficient in multi-core systems, Watkins [14] proposed a shared SPL (Specialized Programmable Logic). The reconfigurable architecture operates at a fine-grained level. The authors claim that the shared SPL is more efficient since the sharing mechanism proposed increases its utilization.

ReMAP (Reconfigurable Multicore Acceleration and Parallelization) [15] is an extension of the work proposed by Watkins [14]. In the ReMAP, fine-grained communication and barrier synchronization were added to accelerate applications within a heterogeneous multi-core. As in the previous work, ReMAP shares your SPL among the processor's cores to reduce power and area costs.

Other examples of architectures that allow resource sharing in a multi-core system are [16],[17],[18],[12]. In [16] and [17] a mechanism to share a custom instruction set among the cores is presented. Shafique et. al. [12] propose a novel policy, minority-game-based, for allocation reconfigurable fabric resource. Finally, in Garcia and Compton [18], a reconfigurable processor, working as a co-processor, is shared in a multi-core system.

This work proposes a partially shared thin reconfigurable array for a multi-core processor. The main contributions in the work proposed here over the reviewed works are:

- Unlike CReAMS and HARTMP, which provide private CGRA to each core, in this work we offer a reconfigurable column to each core but idle resources can be shared at run time providing a better utilization of the reconfigurable resources. The employed reconfigurable array is significantly smaller than CReAMS and HARTMP, amortizing the area cost. For instance, CReAMS uses a reconfigurable architecture which has 144 ALUs, 96 load/store units and 48 multipliers organized in 24 columns to each core. By the other hand, the reconfigurable array proposed in this work uses only 12 ALUs and 4 load/store units.
- Unlike ReMAP and its prior work [14] , the proposed architecture employs resource sharing in a coarse-grained reconfigurable architecture. Although fine-grained reconfigurable architectures are more flexible, CGRAs require fewer configuration bits and have, thus, faster reconfiguration time.
- Differently of [16], [17] and [12], in the work propose here, the whole reconfiguration process is transparent. Thus, neither compiler support nor code modification is necessary to use the reconfigurable architecture.

III. Proposed System

The proposed architecture has three main components: a set of processing cores, a reconfigurable array, and a configuration scheduler. The reconfigurable array is tightly coupled to the processing cores. A schematic view of the proposed architecture can be seen in Figure 1. The components will be explained in detail in the next sections.

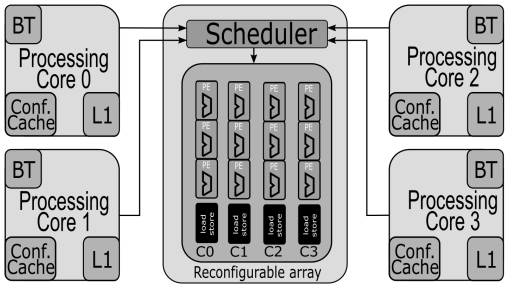

Figure 1. A high-level view of the proposed architecture.

A. Processing Cores

The processing core architecture can be seen in Figure 2. Four processing cores compose the architecture. Each core is a 5-stage pipelined MIPS and is attached to a binary translator (BT). The pipeline processor is shown in Figure 2 tagged with number one. The binary translator (BT) hardware was proposed in [13] and we adapt it to our architecture. A configuration cache and a configuration controller was added to each processing core. The configuration cache stores configurations generated at run-time by the binary translator. The configuration cache can hold up to 128 configurations and uses a LFRU (Least Frequent Recently Used) replacement policy. The configuration controller decides, based on Program Counter (PC) register value, what execute either in the reconfigurable array or in the processor.

Each processing core has private L1 caches. Both L1-Icache and L1-Dcache are 4-way set-associative and have 16KB each. A directory-based mechanism was implemented to provide cache coherency. The MSI (Modified, Shared, Invalid) protocol was used. Details of this protocol can be seen in [19]. Additionally, the sequential consistency model proposed by Lamport [20] was implemented to assure a consistent view of the memory, which is shared among the processing cores.

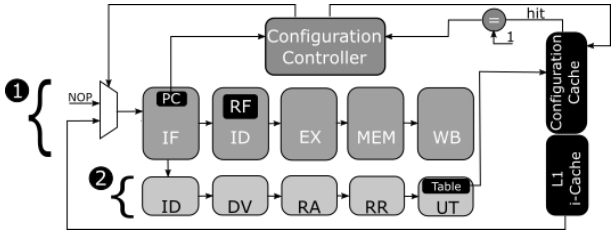

Figure 2. Processing core architecture.

B. The Reconfigurable Array

The reconfigurable array has four reconfigurable columns (block 2 in Figure 3), four register-file (32-entry each), a configurations scheduler (block 1 in Figure 3) and is tightly coupled to the processing cores. Each reconfigurable column has three processing elements (PE) and one load/store unit. The reconfigurable array is a coarse-grained reconfigurable architecture, hence, the PE does word-level operations.

The register-file (block 4 in Figure 3), one to each processing core, at the beginning of the configuration execution, stores the input context, which is a copy of the register-file from its processing core. The whole computation in the reconfigurable columns is register-file based. In other words, each operation reads its operands and writes the result into your register-file. At the end of the configuration execution, the register file is copied back to the processor's register-file and the control returns to the processor to continue its execution. Each register-file has 10 read-ports and 5 write-ports, therefore, it can perform up to 10 reads and 5 writes in parallel.

The configurations control the computation performed in the reconfigurable columns. Several configuration words composes a configuration and each configuration word defines the operations realized in all PEs and Load/Store Unit in a single cycle. This way, the resources of the reconfigurable columns are explored spatially and temporally. The spatial exploration occurs when instructions are executed in parallel in the same cycle and temporal exploration occurs when instructions are executed in different cycles, usually because of data dependency.

Each reconfigurable column is mainly used by its processing core, but it can lend its idle PEs to other processing cores. Therefore, processing cores with a higher workload can use resources from other reconfigurable columns that are being underused. The ability to lend resources imposes some difficulties. At first, if a PE from column C_i is lent to execute operations from a processing core PC_j, the result generated at the PE must be stored in the register-file from the column mainly used by the processing core PC_j. This is done at runtime by the configuration scheduler that adds configuration bits that inform where to save the results to each PE. This approach allows a partial resource sharing in the reconfigurable array. The block 3 in Figure 3 shows the scheduler routing the source registers by adding bits to control multiplexers. In Figure 3, only the routing to read the sources register is shown, but there are also multiplexers to redirect the PE's output and save the computation in the properly register-file.

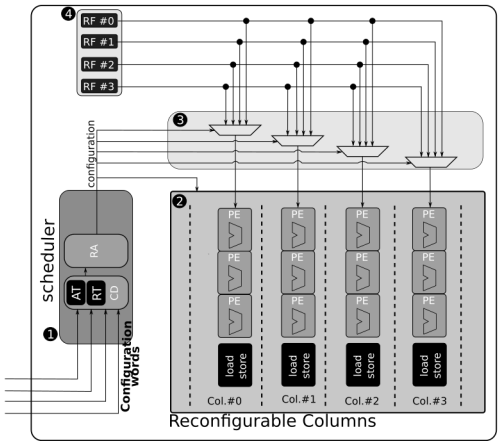

Figure 3. Reconfigurable array architecture.

C. Binary Translator

The binary translator (BT) is a hardware block that dynamically detects a sequence of instructions to be executed in the reconfigurable array. In the implementation proposed in [13], BT is implemented as a 4-stage pipelined circuit. In our implementation, the BT was extended to a 5-stage pipelined circuit to add a rename stage between the resource allocation and update table stages. This additional stage was designed to handle false dependence. The implementation proposed in [13] also handles false dependence but only in bus-based architectures. The BT pipeline and how it is attached to the processor pipeline is shown in block 2 in Figure 2.

The BT algorithm uses some tables to store information about the current configuration. The main tables are: write bitmap table, resource table and read table. In the write bitmap table, dependency information of each column is saved. The resource table is used to check if there is resource available in the columns. And the read table is used to store the source registers of each column. At the end of a configuration, all these tables are used to build the configuration, which is saved in configuration cache.

The 5-stage pipeline BT works as follow. In the first stage, the instruction is decoded, getting the source and target register and operation. In the second stage, the write bitmap table is used to verify data dependency. This stage calculates and returns the column number where this instruction can be added considering data dependency. In the third stage, using the information from the last stage, the resource table is used

to search for functional units available in the columns. In the fourth stage, the register renaming is done. Finally, the last stage saved the computed results for the last instruction updating the tables.

The configuration is saved, indexed by the PC of the first instruction belonging to that configuration, in the configuration cache when it reaches its end. There are three reasons for a configuration be ended by the BT: *i)* a not supported instruction was found in the BT; *ii)* there is no resource available in the reconfigurable array or *iii)* found a branch instruction (in the case where there is no speculative execution support). As mentioned before, the reconfigurable columns are explored temporally, therefore, the amount of resources in the reconfigurable array is limited to the number of configuration words that composes a configuration. Once the configuration is saved, the next time the processor reaches that PC value again, this snippet of code will be executed more efficiently in the reconfigurable array. The version implemented in this work supports one speculation level, so the BT allows up to 1 branch per configuration.

D. Configuration Scheduler

The configuration scheduler allocates the operations from configurations to be executed in the reconfigurable array, providing a dynamic partial sharing of the reconfigurable resources.

Each processing core controller sends its configuration words to the configuration scheduler which maps this set of configuration words in the reconfigurable array following a policy. The policy follows the basic priority rule, which states that each processing core uses its reconfigurable column. For instance, for the processing core i is guaranteed, at least, the resource of the reconfigurable column i. Besides the basic priority rule, there is a priority thread rule which assures a specific thread a higher priority in the allocation process. This allows a more efficient resource utilization because it gives more computational resources to more important processes in the operating system.

It is important to mention that the configurations are generated as if there were 5 PEs in each reconfigurable column. In the resource table (used in the BT's first stage) the column is seen as having 5 PEs. As the reconfigurable column actually does not have 5 PEs, if a configuration word has more than 3 PE operations, additional PEs must be lent from other columns. Regardless of the thread priority rule (or if two or more threads have the same priority), the scheduler always checks if the column of other cores has idle resources. If so and if there are cores with more than 3 operations in your configuration word, idle resources will be taken and lend for the threads that need them. In case there is no idle PE in other reconfigurable columns, the additional operations are executed in the next cycle in its reconfigurable column.

The configuration scheduler is organized in a 2-stage pipelined circuit. In the first stage, configuration decode (CD), the configuration words from each processing core are allocated following the basic priority rule and saved in a 4-entry table with 3 bits, named allocation table (AT). Each entry contains information about PEs from each reconfigurable column and each bit represents if the PE is allocated or not. The additional operations from each configuration word are mapped in a second 4-entry table with 2 bits, named request table (RT), and maps the operations requests for PEs from each processing core. In the second stage, resource allocation (RA), using the AT and RT, the configuration scheduler maps the configurations in the reconfigurable array. To each PE allocated in the AT, the configuration bits are forwarded to its reconfigurable column because this table maps the allocation done using basic priority rule. The RT is checked if there is some bit set to 1 (this means that the configuration word requires additional PEs), if so, a search for an idle resource in other reconfigurable columns is done. In case an idle PE is found, the configuration bits from the requested PE is forwarded to the idle PE found. Additionally, bits are added to the configuration to inform where to save the result from that computation. In this case, the register-file from the column which requested the PE.

IV. Results

A. Methodology

A benchmark with four applications was selected to evaluate the proposed system. The applications are: bitcount (from Mibench suite [21]), lu decomposition (from openMP suite [22]), matrix multiplication and a laplacian filter. These applications were selected due to their different characteristics. Bitcount is a control-flow application and has small basic blocks (5.87 instructions per basic block on average), which makes their acceleration harder due to the space to explore ILP (Instruction Level Parallelism) is too small. On the other hand, matrix multiplication is a compute-intensive application and has larger basic blocks (13.24 instructions per basic block on average). In the Lu and laplacian filter have both characteristics compute-intensive and control-flow. The bitcount apllication counts the number of 18,750 integers. Matrix multiplication takes two 20x20 matrix as input and generates a 20x20 matrix as result. Lu decomposion takes a 20x20 matrix as input and generate a 20x20 matrix as output. Finally, the laplacian takes an image in gray scale with SQCIF resolution (128x96). All the applications were written in C language and compiled to MIPS using the gnu gcc cross-compiler. To obtain the performance results, a simulator using the SystemC language was developed.

The performance and energy gains were evaluated comparing a version with and without the proposed reconfigurable architecture. Two additional versions of the proposed reconfigurable architecture were developed to measure the improvements the sharing mechanism can provide. In the first version, we disable the resource sharing, thereby, the reconfigurable columns are private to each core. In the second version, we also disable the resource sharing but add 2 PEs to each reconfigurable columns, thereby, each reconfigurable column has 5 PEs in this version. This version was named as 5 PEs version.

The energy analysis was done using McPAT framework 1.3 [5]. The column array was implemented with the Cadence GSCLIB045 (Cadence 45nm Generic Std Cell) library. With this implementation was possible to obtain a frequency of 700MHz. For energy results, we assume that this technology and frequency was used to implement the whole architecture. Counters were added to the systemC simulator to provide execution statistics required by McPAT.

The reconfigurable array energy estimation was done using also the McPAT output. We described, in the McPAT framework, a superscalar processor with three integers ALU's (Arithmetic Logic Unit) in its execution stage. We assume that the reconfigurable array power consumption is similar to the superscalar execution stage. It is important to mention that this is a pessimist approach due to the execution unit of a superscalar processor having much more hardware than our reconfigurable array, therefore, we compute more energy than really is used. The same energy estimation was performed to calculate the energy consumption in the 5 PE version.

B. Performance Results

The reconfigurable array provides performance improvement in the whole benchmark, as can be seen in Figure 4. The speedup provided is 41% on average. The resource sharing mechanism also improved the performance against the version without sharing. The improvement was 8.5% on average. Bitcount was the application with the smallest improvement due to the sharing mechanism. In this control-flow application, the basic blocks are small and with many dependent instructions, which limits the amount of parallelism. For this reason, bitcount was improved only by 4.65% in its performance using the proposed resource sharing mechanism. On the other hand, the laplacian filter, which has larger basic blocks than bitcount and higher ILP, performance improvement was 11.16%.

The proposed architecture also was compared to the 5 PEs version that represents the highest gain the reconfigurable architecture can provide. The speedup provided by our architecture was 4.73% smaller than the 5 PEs version on average. Bitcount was the closest to the 5 PEs version speedup. The reason is the same explained before: low ILP and small basic blocks. Therefore, increasing the resource amount doesn't imply in performance gains, once that 3 PEs is enough in most configurations.

Additional performance results can be seen in Table I. There is a trend that can be observed: applications with higher coverage by the reconfigurable architecture and bigger configuration sizes have better speedups.

The performance results show the proposed resource sharing mechanism could improve up to 11.16% in the laplacian filter performance when compared to a version without the sharing mechanism. However, the performance improvement in the bitcount application was not significantly, only 4.73%.

C. Energy Results

The energy improvements can be seen in Figure 5. The comparison showed that the proposed architecture uses only

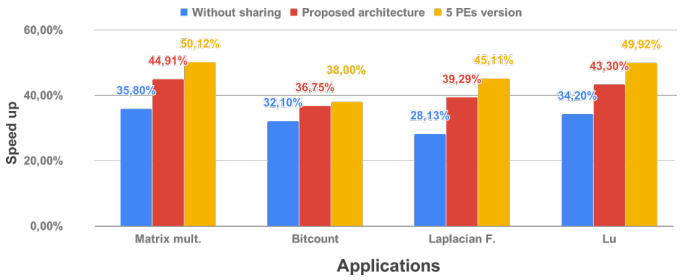

Figure 4. Analysys of the speed up provided by resource sharing.

Table I
PERFOMANCE RESULTS.

Application	Cycles	Average conf. size (in instr.)	Speed up	Coverage	Miss speculation rate
Matrix Mult.	67,334	27	44.91%	82.2%	4.7%
Laplacian F.	178,563	41	39.29%	62.5%	0.7%
Lu	17,676	22	43.30%	85.4%	5.02%
Bitcount	789,809	16	36.75%	72.09%	10.60%

24% of the power consumption of its counterpart on average. The best energy saving was in Lu and Matrix applications that also are those which present the highest coverage. On the other hand, the laplacian filter application showed the smallest energy saving. This occurs because it executes more code in the processor than the other applications. Therefore, it can be observed that a larger coverage implies better energy efficiency because the reconfigurable array consumes less power than the processor to run the same snippet of code.

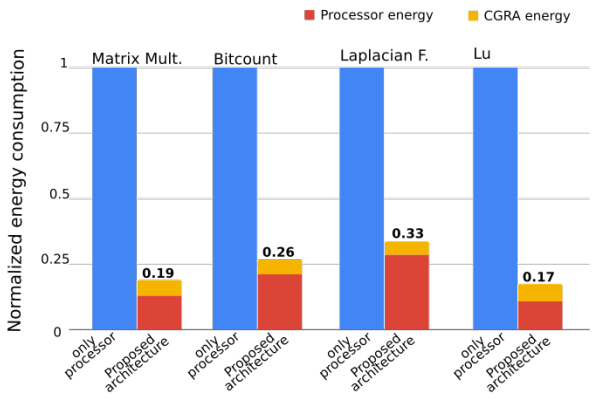

Figure 5. Energy results normalized by the processor power consumption.

An energy comparison between the 5 PEs version and the proposed architecture also was performed. This comparison aims to show a power-performance tradeoff of our architecture against a reconfigurable array that has more resources, but their resources are private to each core. The energy comparison can be seen in Figure 6. In the bitcount application, the 5 PEs version was 1.25% faster than the proposed architecture, however, it consumes 33% more power. As explained before,

this happens because the additional units are almost never used due to the low ILP and only add static power consumption in the system. On the other side, matrix multiplication is 5.21% faster and consumes only 13% more than the proposed architecture. The 5 PEs version reached the best speedup, but the increase in power was higher than the obtained speed up. This occurs due to the proposed sharing mechanism, which offers better resource utilization, saving energy and increasing performance. In Table II is shown a tradeoff performance-power of the 5 PEs version against the proposed architecture.

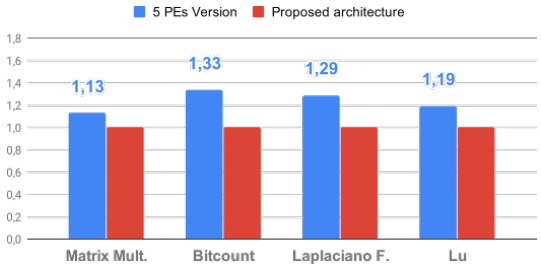

Figure 6. Energy results normalized by the proposed architecture power consumption.

Table II
COMPARISION BETWEEN THE 5 PES VERSION AND THE PROPOSED ARCHITECTURE

Application	speed up	Energy
Matrix Mult.	+5.21%	+13%
Bitcount	+1.25%	+33%
Laplacian F.	+5.82%	+29%
Lu	+6.62%	+19%

V. CONCLUSIONS AND FUTURE WORK

This work proposes a reconfigurable array that is partially shared among the processor cores. The sharing mechanism is performed dynamically and aims to provide better resource utilization of the reconfigurable array. The performance results showed that the sharing mechanism improves the performance with respect to a version without sharing. Additionally, energy results showed that the reconfigurable array provides significant energy savings when compared to a version without our reconfigurable array.

As future work, we intend to add more applications to our benchmark and to test more configurations of the system, varying the amount of PE that can be shared, the amount of PEs available to each column and so on. Thus, we can decide which configuration brings a better power-performance tradeoff. Additionally, a study in area reduction cost by resource sharing, because this is another advantage of the better utilization provided by the resource sharing mechanism and the thin reconfigurable array here proposed.

REFERENCES

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 37–47.

[2] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, "Towards a dynamic and reconfigurable multicore heterogeneous system," in *2014 Brazilian Symposium on Computing Systems Engineering*, Nov 2014, pp. 73–78.

[3] J. D. Souza, L. Carro, M. B. Rutzig, and A. C. S. Beck, "A reconfigurable heterogeneous multicore with a homogeneous isa," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1598–1603.

[4] Accellera, "Systemc language," https://www.accellera.org/downloads/standards/systemc, 2016, accessed: 29/07/2019.

[5] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[6] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The chimaera reconfigurable functional unit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, pp. 206–217, Feb 2004.

[7] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, May 2000.

[8] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, April 2000.

[9] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 235–244.

[10] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '01. Piscataway, NJ, USA: IEEE Press, 2001, pp. 642–649.

[11] D. B. Gottlieb, J. J. Cook, J. D. Walstrom, S. Ferrera, Chi-Wei Wang, and N. P. Carter, "Clustered programmable-reconfigurable processors," in *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, Dec 2002, pp. 134–141.

[12] M. Shafique, L. Bauer, W. Ahmed, and J. Henkel, "Minority-game-based resource allocation for run-time reconfigurable multi-core processors," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.

[13] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 1208–1213.

[14] M. A. Watkins, M. J. Cianchetti, and D. H. Albonesi, "Shared reconfigurable architectures for cmps," in *2008 International Conference on Field Programmable Logic and Applications*, Sep. 2008, pp. 299–304.

[15] M. A. Watkins and D. H. Albonesi, "Remap: A reconfigurable heterogeneous multicore architecture," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 497–508.

[16] L. Chen and T. Mitra, "Shared reconfigurable fabric for multi-core customization," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 830–835.

[17] L. Chen, T. Marconi, and T. Mitra, "Online scheduling for multi-core shared reconfigurable fabric," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 582–585.

[18] P. Garcia and K. Compton, "Kernel sharing on reconfigurable multiprocessor systems," in *2008 International Conference on Field-Programmable Technology*, Dec 2008, pp. 225–232.

[19] D. A. Patterson and J. L. Henessy, *Computer Architecture: A Quantitative Approach*, 5th ed. Oxford, USA: Morgan Kaufmann, 2016.

[20] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[22] A. J. Dorta, C. Rodriguez, and F. de Sande, "The openmp source code repository," in *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Feb 2005, pp. 244–250.