Minority-Game-based Resource Allocation for Run-Time Reconfigurable Multi-Core Processors

Muhammad Shafique, Lars Bauer, Waheed Ahmed, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Karlsruhe, Germany
{muhammad.shafique, lars.bauer, ahmed.waheed, henkel}@kit.edu

Abstract:
A novel policy for allocating reconfigurable fabric resources in multi-core processors is presented. We deploy a Minority-Game to maximize the efficient use of the reconfigurable fabric while meeting performance constraints of individual tasks running on the cores. As we will show, the Minority Game ensures a fair allocation of resources, e.g., no single core will monopolize the reconfigurable fabric. Rather, all cores receive a "fair" share of the fabric, i.e., their tasks would miss their performance constraints by approximately the same margin, thus ensuring an overall graceful degradation. The policy is implemented on a Virtex-4 FPGA and evaluated for diverse applications ranging from security to multimedia domains. Our results show that the Minority-Game policy achieves on average 2x higher application performance and a 5x improved efficiency of resource utilization compared to state-of-the-art.

1. Introduction and Motivation

Advancements of FPGAs have led to the emerging trends of run-time reconfigurable multi-core processors [1][4][22] that integrate several cores (RISC, VLIW, etc.) with reconfigurable fabrics. These processors deliver high computing performance while considering the following run-time-varying *system dynamics*:

a) The mapping of tasks which depends upon the status of available processor cores, the state of the multi-core processor (e.g., due to thermal effects), the user behavior, etc.
b) Changing task requirements (for obtaining reconfigurable fabric) due to changing performance constraints, etc.
c) Varying workload characteristics of the applications.

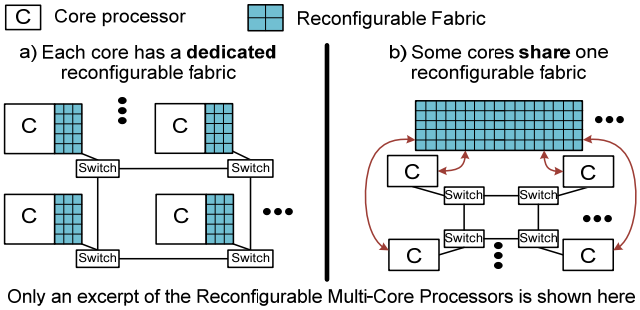

Fig 1. Two trends in reconfigurable multi-core processors (a) Dedicated fabric (like [1]), (b) shared fabric (like [4])

Depending upon the coupling of the fabric[1] to the core processor, there are two architectural categories of reconfigurable multi-core processors (as shown in Fig 1):

(a) Reconfigurable multi-core processors with a *dedicated fabric* (Fig 1a) comprise core processors and their individual fabrics. An example is RAMPSoC [1] where each core represents a reconfigurable instruction-set processor (e.g., [2] or [3]). However, such an architecture, scenarios may arise where one core fully utilizes its fabric (and probably could require even more than that), while another core may only partially or not at all use its fabric. This may result in overall inefficient fabric utilization (i.e., the speedup per available fabric).

(b) Reconfigurable multi-core processors with *shared fabric* overcome the above-mentioned deficiencies by coupling several core processors with one rather large fabric [4] (Fig 1b, such an architecture may be realized on the Xilinx's upcoming 7 series FPGAs where several ARM/Microblaze cores share the same fabric [21][22]). As target applications, Xilinx envisions mobile devices (with audio, video, 4G, games), 3D multimedia, medical image processing, software defined radio, Wireless LTE, etc. [21]. In processors with *shared fabric*, different cores may place their accelerators on the same shared fabric[2]. Therefore, these multi-core processors may provide a higher efficiency by allocating less or more fabric to individual cores depending upon their respective computational properties. Consequently, different cores compete for a share of the fabric to expedite their tasks.

The challenging problem that arises is: "To determine *which share* of the fabric should be allocated to *which core* at *which time* in order to i) optimize the fabric utilization and ii) meet the performance constraints of individual tasks under varying system dynamics". This instantiates the need for an *adaptive fabric resource allocation policy*.

1.1. Motivation for a Minority-Game-based Allocation Policy

Fig. 2 shows different allocation policies [14][15] where the fabric is shared among three tasks (A, B, C) executing in parallel on different cores. The system dynamics like priorities and performance constraints of these tasks change at run time (as denoted by *t0-t3* on the timeline, Fig. 2). The *demand* denotes the number of reconfigurable containers a task requires to meet its performance constraint. The following policies are analyzed:

- *Equal Distribution (ED)* provides an equal share to all tasks *irrespective* of their priorities and performance constraints. This results in an inefficient utilization of the fabric in certain cases (see Fig. 2).
- *First Come First Serve (FCFS)* favors the first tasks demanding the fabric. It ignores the priority and demand of the subsequently arriving tasks. Therefore, it may lead to a situation where a low priority task arriving first (see Task A in Fig. 2) may monopolize the entire fabric. Consequently, a high priority task that arrives afterwards (see Task B in Fig. 2) does not get a share from the fabric for acceleration. This may lead to task degradation or even to malfunction of system services, etc.
- *Highest Priority First (HPF)* fulfills the demands of tasks in the order of their priorities. It may lead to situations where the task with the highest priority may monopolize the entire fabric whereas other tasks are not considered (see Fig. 2). Similar to FCFS, this may also lead to task degradation or even to malfunction of system services, etc.

Summarizing: each policy is favorable in a certain scenario but no single policy performs efficient allocation (in terms of satisfying the demands of all the tasks) in multiple of the scenarios as occurring when the system dynamics change at run time.

A reconfigurable multi-core processor with unpredictable constraints is a complex dynamic system where typically no equilibrium (i.e., a fixed fabric allocation decision) is reached during the run time of an application. Moreover, run-time reconfiguration and varying dynamics of the system lead to a situation where reaching an equilibrium is no longer critical since from now on *adaptivity is the key to handle the dynamic nature of a multi-core system*. Therefore, a low-overhead run-time adaptive resource allocation policy for reconfigurable multi-core processors is desirable.

Minority Games [13] – inspired by the El Farol bar problem [20] – provide an ideal solution for complex dynamic systems where several selfish entities compete for resources and the system dynamics continuously change during the run time of a system [12][13]. Minority Games provide a means for multivariable optimization and give a near-to-optimal solution [18][19] (as shown by our results in Section 6.2). All entities tend to be selfish (i.e., tend to monopolize the resource, in our case fabric) and tend to place themselves into the so-

[1] For brevity, the term "fabric" is used when we mean to say "reconfigurable fabric".

[2] The fabric is partitioned into *rectangular containers* that are reconfigured at run time to contain different accelerators.

978-3-9810801-7-9/DATE11/©2011 EDAA

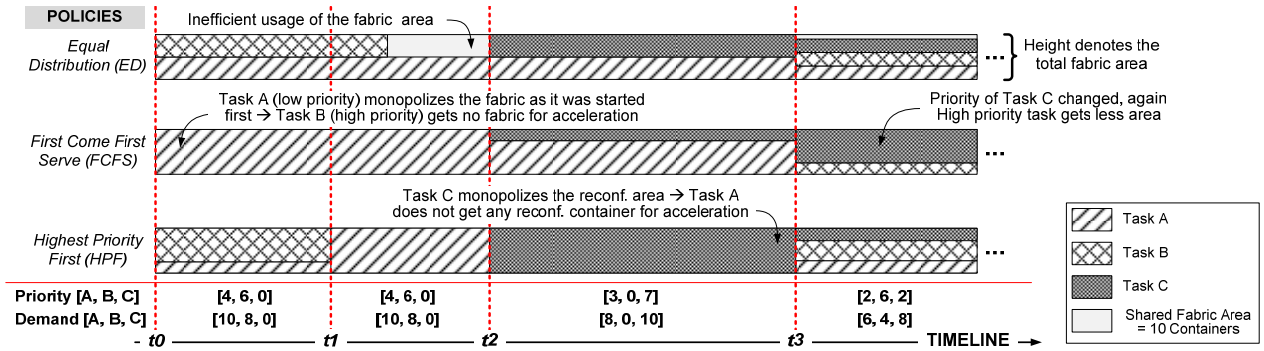

Fig. 2: Scenarios show various resource allocation policies at work when task priorities and performance constraints change

called minority group irrespective of other entities (see details in Section 3). In absence of a unique best way to proceed, the entities have no choice but to adapt their strategy and learn over time with respect to the behavior of other entities, after the result of a game is announced [18]. As a result, they behave less-selfish across the games, although they may be selfish within a single game.

The key difference to the above-discussed and other optimization techniques (like in [4]) is that Minority Game does not go for maximizing a certain benefit function (e.g., performance) of ONE certain entity. Rather it *collectively considers the demands of all entities* and *provides a fair distribution* [5][6]. In this way, the participating entities drive towards a state where their own performance improves, such that, the demands of all players are close to be fulfilled. Due to its properties and dynamics, Minority Games promise (nearly) satisfying the demands of all entities while maximizing the global efficiency (which is desirable in our fabric allocation problem). Therefore, our inspiration has its origin in the prominent Minority-Game-based approaches [5][6] for resource allocation and sharing in distributed computing systems that use Prospect Theory [7] to achieve emergence of cooperation among selfish entities while providing adaptivity. However, the solutions of [5][6][7] cannot be directly applied to the reconfigurable multi-core processors, as the system model of a dynamically reconfigurable processors is different the system model of [5][6]. *One of the key challenges is modeling the system and the corresponding adaptive resource allocation problem as a Minority Game, such that a game-theoretic paradigm can be employed to obtain a fair solution.*

Our novel contribution is as follows:

A Minority-Game-based adaptive policy for resource allocation in reconfigurable multi-core processors considering run-time varying system dynamics (task mapping, task priority, etc.); In particular we:

- formulate the system model of a reconfigurable multi-core processor with shared fabric and the corresponding adaptive resource allocation problem as a Minority-Game (Section 5.2)
- devise a decision function for determining the minority side and for fabric allocation in each round of the game (Section 5.2) to maximize the efficiency of fabric utilization in an adaptive way; the proposed decision function jointly considers priority, performance constraints, and the history of allocation decisions

The paper organization: Section 2 discusses related work. Section 3 presents an overview of the Minority Games. Section 4 provides our system model, while Section 5 presents our novel fabric allocation policy followed by the results in Section 6. Section 7 concludes.

2. Related Work

Resource allocation is a well-studied problem in distributed computing, especially within Grid collaborations [5][6][15] where adaptivity and less-selfishness are important design constraints.

Resource sharing and allocation policies in high-performance reconfigurable computing deploy a compile-time fixed partitioning of FPGAs, one for each microprocessor. The basic idea is to enlarge the FPGA virtually (from the applications' point of view) to enable sharing of an FPGA among concurrently executing task [8][9]. In case of contention, tasks wait for the availability of the virtual FPGA. Another drawback is that the same amount of fabric is allocated to each processor, i.e., analogous to the processors with dedicated fabric like RAMPSoC [1] (as shown in Fig 1 a). Consequently, the tasks with variable demands for fabric cannot be accommodated efficiently.

Recently, resource sharing has been emerged in reconfigurable multi-core processors with a shared fabric [4]. The policies in [4] use a fixed partitioning (in power of 2) of the fabric. Furthermore, they only target statically mapped tasks on certain cores with compile-time prepared groups of tasks with low and high demand of fabric. The tasks in the same group are mapped on the neighboring cores and they are allowed to share a partition. Such a static grouping performs inefficiently under run-time varying system dynamics. In contrast, our policy allocates the fabric in an adaptive and less-selfish way to avoid monopolization and to maximize the efficiency of fabric utilization, while considering the run-time varying system dynamics.

Before proceeding further, we will briefly present the basics of Minority-Game for better understanding of the paper.

3. Background: Overview of Minority Game

A complete Minority Game (MG) consists of several rounds. In each round, all players (i.e., the entities) play to obtain a certain part of the (limited) resource. Once the resource is completely allocated, the game is finished. There are two main categories of MG, *single-choice* games and *multi-choice* games [6]. In a single-choice game, each player has to choose one out of two options, i.e., the player decides *to use* or *not to use* the resource. In contrast, players in a multi-choice game have more than two options like, *which* and *how much* share of the resource to use. After all players have made their decisions, the minority side (i.e., the side with lesser number of players) is declared the winning side and the players of this side receive their demanded resource share. A history of the winning sides is kept for the previous rounds. In the subsequent rounds, the players make their decision by considering a set of entries in the history. During a game, players do not know the decisions of the other players. At the end of each game, the allocation result of the game is broadcasted which may be used by the players to adapt their strategy. This enables a less-selfish behavior across different games.

In the following, we explain the basic terminology that is used in game-based algorithms [5][6]:

- *Payoff:* the performance benefit that the demanding entity achieves by using the demanded resource (e.g., in our case, it is the fabric)
- *History:* a record of the winning sides for the previous rounds
- *Attitude:* it is defined as the tendency to obtain the resource; it is typically given as the weighting factors for the payoff and the history, as inspired by the Prospect Theory [7]
- *Attractiveness:* a joint function of payoff, attitude, and history as a behavioral predictor, for making a decision on the resource usage

Kindly note that we follow the above terminology for consistency reasons with conventional game-based algorithms [5][6]. The formulation of *Payoff*, *Attitude*, and *Attractiveness* is different in our case compared to that in [5][6] due to different system dynamics and a different system model.

4. System Modeling

4.1. Processor Model

In order to efficiently allocate and optimize the utilization of several fabrics within a reconfigurable multi-core processor, we devise the concept of Groups-of-Cores (GOCs, Fig 3). Each GOC is a coalition of N_C cores (RISC, VLIW, etc.) that share one fabric (Fig 3) for accelerating their respective application tasks. GOCs make their sharing decision independent of other GOCs, as the fabric is only shared among the cores of a GOC. A GOC can be formalized as:

$$GOC = \{ PRC_J \} + \{ C_K \},\ J \in [1, N_{PRCs}],\ K \in [1, N_C]$$

where, the fabric is partitioned into N_{PRCs} *Partially Reconfigurable Containers* (PRCs). They can be reconfigured at run time to contain accelerators. The cores are connected to each other via a Network-on-Chip whereas the PRCs are connected with segmented buses [16]. Each shared fabric is equipped with a *central playground* to host the game. It is a dedicated entity that is attached to the shared fabric resource where the players of the game dispatch their demands. In our case, it is a dedicated IP-core (a static non-reconfigurable soft core on the FPGA) attached to the PRCs (see Section 6.4 for area results).

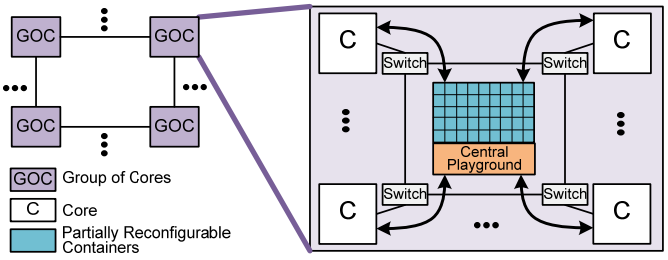

Fig 3. Reconfigurable multi-core processor with multiple GOCs; N_C cores of a GOC share one fabric (with several PRCs)

4.2. Application Model

There are N_T independent and autonomous application tasks[3] executing on the reconfigurable multi-core processor, where each task has a priority and performance constraint. We consider tasks with soft performance constraints (i.e., a deadline miss is acceptable and considered as quality of service degradation). Application tasks are denoted by:

$$A = \{ A_T \},\ T \in [1, N_T]$$

Only one task A_T executes on a particular core C_k of a *GOC* at an instant. The task mapping may change at run time. Hence, after a task mapping is fixed, the resource allocation policy always needs to reallocate the number of PRCs to a core and its task. Since each core has its own task (that efficiently utilizes its allocated PRC resource) and the duration of tasks is typically much longer (in multiples of seconds or minutes), the reconfiguration latency is typically negligible compared to the task duration. The demand D_{tk} of core C_k denotes the total number of PRCs that C_k requires to expedite task A_T for meeting its performance constraint. Note, depending upon the number of available PRCs, a task may execute slower or faster due to the realization of Custom Instructions (composed of hardware accelerators reconfigured in PRCs) in a more or less parallel mode. Such a case has already been exploited by the authors in [2]. Therefore, each task exhibits a certain latency improvement (L_{saving}) for a given number of allocated PRCs compared to its execution without using the PRCs. A task A_T requires a number of PRCs ($DStep_{TCk}$) for a performance improvement of $\Delta L_{savingTCk}$. Note, $DStep_{TCk}$ is not a constant value as it may change in each round of the game depending upon the number of the obtained PRCs in previous rounds. The efficiency of PRC utilization is defined as:

$$P_{Ck} = \Delta L_{savingTCk} / DStep_{TCk} \qquad (1)$$

The priority of task A_T is expressed by the weighting factor ap_{Ck}. Note that the task priorities and the task mapping are handled by the operating system and are considered as a given and available information (i.e., not part of the scope of this paper).

[3] In the scope of this paper, we do not consider dependent tasks. Authors in [17] used task criticality to adapt multi-core hardware for dependent tasks.

5. Minority-Game-based Resource Allocation Policy

5.1. The Policy in Overview

The players of the policy-implementing game are the cores of a GOC. P_{Ck} and $\Delta L_{savingCk}$ information (obtained by offline profiling and updated at run time), PRC demands, and core attitudes for application tasks are provided to the policy. Since these parameters may change at run time due to a change in the system dynamics, the policy adapts its PRC allocation decision accordingly (Section 5.2). Fig 4 shows the execution flow of our policy (see algorithm in Section 5.3). The following two rules need to be satisfied for starting a game:

Rule-1: If the total number of PRCs in the fabric (N_{PRCs}) is equal to or greater than the total demand of all participating cores then the game is skipped and the requirements are just fulfilled.

$$\sum_{\forall C_k \in C} D_{C_k} \leq N_{PRCs}$$

Rule-2: If only one core is playing the game, all PRCs may be allocated to this core and the game is skipped.

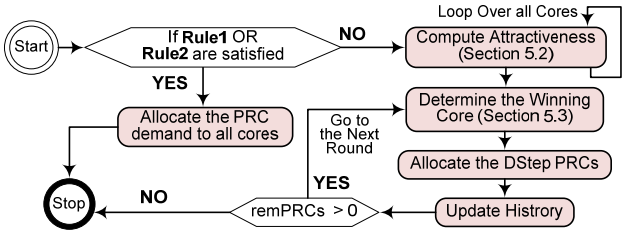

Fig 4. Execution Flow of the Policy

In each round of the game, all cores play for their task's demanded PRCs. First, the *attractiveness* of all cores as a decision function is computed which is a joint function of the efficiency of PRC utilization and the history of PRC allocations (see Section 5.2). The decision about the winning cores is made depending upon the attractiveness value (Section 5.2 & 5.3). The winning core receives its corresponding number of PRCs for which it was playing. Then, the PRC allocation history is updated which helps preventing monopolization. All cores should receive a *'fair share of fabric'*. We consider it a fair allocation, for instance, if tasks would miss their performance constraints by approximately the same margin, which would help to ensure an overall graceful degradation.

A game is completed once the demands of all participating cores are fulfilled or the available PRC resource is entirely allocated to the cores. The result of a game (i.e., the allocated PRCs and the unfulfilled demands) is transmitted to all cores within a GOC. The goal is to maximize the efficient use of the fabric while meeting the performance constraints of individual task running on the cores.

5.2. Modeling Resource Allocation as a Minority-Game

N_C cores $\{C_K\}$ with $K \in [1, N_C]$ (players of the game) in a *GOC* compete for the PRCs in the fabric $\{PRC_J\}$ with $J \in [1, N_{PRCs}]$. A game consists of $r \in [1, R]$ rounds. A round represents a situation where a core C_k wins $DStep_{TCk}$ number of PRCs (Fig 4). Since each player has the possibility to make different choices regarding the number of PRCs, it is a *multi-choice game* [6]. A core C_k receives a payoff P_{Ck} for its $DStep_{TCk}$ number of PRCs (see Eq. 1).

Following the Minority-Game, the history H of the winning side is consulted for making a choice. Inspired by the ideas of behavioral predictors [6], we use the following *attitudes*:

$$a_{HC_k}, a_{pC_k} \in [0, 1]\ ,\ \forall C_k \in \{ C_K \}$$

where, a_{HCk} and a_{pCk} represent the 'attitudes of a core C_k towards winning' for the history-based and payoff-based decisions, respectively. a_{pCk} is derived from the priority of the task A_T. If a_{pCk} is zero, the PRC allocation is purely based on the history. In this case, all cores have an equal probability to win the PRCs, which is somewhat similar to the Equal Distribution policy (Section 1.1). Alternatively, if a_{HCk} is zero, the PRC allocation is purely based on the priority and the payoff (i.e., similar to the Highest-Priority First policy but considering a performance constraint, thus performance-seeking). A core

may adapt its attitudes (thus adapting the strategy) for the next game depending upon the result of the preceding game to favor other cores. The payoff, the history of the winning side, and the attitudes are used to compute the *attractiveness* of a core's choice (i.e., its behavioral predictor for making a choice) for winning the PRC resource. The attractiveness is computed as:

$$Attr_{C_k} = [\, a_{HC_k} * (1 - P(H, C_k)) + a_{pC_k} * Payoff_{NC_k} \,] \quad (2)$$

where $P(H, C_k)$ and $Payoff_{NCk}$ are normalized to 1. $Payoff_{NCk}$ is the normalized payoff with respect to the highest payoff (Eq. 3). $P(H, C_k)$ is the history information that how many times the core C_k won the PRC resource in previous rounds: $P(H, C_k)=WinningCount C_k / H$.

$$Payoff_{NC_k} = P_{C_k} / max_{\forall C_k \in C}(P_{C_k}) \quad (3)$$

The core C_k with the highest attractiveness wins $DStep_{TCk}$ number of PRCs. All other cores on the losing side have a higher probability to win in the subsequent rounds due to their history. Our policy is a specialization of a Minority-Game where the winning side always contains exactly one player[4].

5.3. Adaptive Resource Allocation Policy

Fig 5 shows the pseudo-code of the policy and the inputs. Since the performance of tasks may vary (for a given number of PRCs), the entries in the compile-time prepared average case payoff lists are updated accordingly.

A performance constraint is conveyed through the demand of a core ($D[N_C]$). Two rules for starting the game are checked in lines 1-4. The total rounds of the game depend upon the total PRC demand of all cores and the total number of PRCs (N_{PRCs}, line 6). The maximum payoff is computed in Line 7. It is later used for normalizing the payoffs in line 11. For computing the attractiveness, a history of the winning side is obtained in line 9. In the first round of the game, all cores have an equal selfishness to compete for the fabric (lines 9 & 11). As a result, the decision is dominated by the priority and payoff. Then the core attractiveness is computed in line 11. It is checked for the highest attractiveness in lines 12-14 and a decision on the winning core is declared. After each round, the share of the winning core, the corresponding variables, and the history are updated (lines 16-21). The game continues until demands of all cores are fulfilled or the total number of PRCs is exhausted (i.e., distributed through allocation).

The worst case complexity of computing the attractiveness is O(#PRCs x #Cores) for all *DSteps=1*. A typical game includes cores executing tasks with both low and high PRC demands. Once the total demand of a task on a competing core is fulfilled (line 22), that core no longer participates in the game (line 23). Moreover, *DSteps* > 1 typically holds. Therefore, the expected run time of the policy is less than the worst-case run time.

6. Results and Evaluation

6.1. Experimental Setup

Application Tasks	Max. PRC Demand	# CIs used	Application Tasks	Max. PRC Demand	# CIs used
H.264 Encoder	20	12	*Susan*	20	*3*
H.264 Decoder	18	7	*ADPCM Encoder*	2	*1*
JPEG Encoder	18	*3*	*ADPCM Decoder*	2	*1*
JPEG Decoder	20	3	*SHA*	2	*1*
AES Encrypt	5	1	*CRC*	1	*1*
AES Decrypt	4	1			

Table 1: Max. PRC demands and number of CIs for all tasks

For evaluation, we have engaged various application tasks including an entire H.264 video en/decoder [10] and various application tasks from the MiBench suite [11] with different performance constraints and mapping combinations. Due to their diverse processing behavior, these tasks vary in their PRCs demand (see Table 1). The PRCs host hardware accelerators to implement Custom Instructions (CIs) that are deployed to expedite a task. In order to simulate run-time varying scenarios, we have validated our policy for more than 300 different task-mapping scenarios. These scenarios include combinations of simple (CRC, SHA, ADPCM) and complex (H.264, Susan, JPEG, AES) tasks. We have used four SPARC V-8 cores in a GOC. From our 572 different experiments, we have determined that the attitude for history $a_H = 0.6$ provides the highest performance saving for various sizes of the fabric. For all experiments, we use a 100 MHz frequency.

Adaptive Resource Allocation Through Minority Game

INPUT:
C, N_C : Set and Total Number of all participating cores in the game
N_{PRCs} : Total Number of PRCs shared by the cores of a GOC
$D[N_C]$, $DS[N_C][\,]$: List of Demands and all *DSteps* for tasks
$P[N_C][\,]$: List of avg. case Payoffs for each *DStep*, updated at run time
$a_p[N_C]$: List of Payoffs attitudes for all cores, may change w.r.t. priority.
H, a_H : Size of the History and History attitude for all cores
TRD: Total PRC demand of all participating cores

OUTPUT: $S[N_C]$: PRC share of each participating core

BEGIN:
1. **If** *Rule-1 or Rule-2 is satisfied* **Then** // Section 5.1
2. $S[C_k] = D[C_k],\ \forall C_k \in \{C_K\}$
3. return; // $D[N_C]$ is fulfilled for all cores and the game is skipped
4. **End If**
5. remPRCs = N_{PRCs}; Hist[H] = Ø; $S[N_C]$ = Ø;
   // Start Game
6. **While** *(remPRCs > 0)* **Do** // Loop over all rounds of the game
7. P_{max} = *findMax*(P[Nc][ ]); $Att_{High} = 0$;
8. **For** *all cores* $\forall C_k \in \{C_K\}$ **Do**
9. P(H, C_k) = (*firstRound*) ? 0, *FindOccurences*(C_k, Hist[H]);
10. IDS_{Ck} = *getCurrentDStepIndex*($DS[C_k][\,]$);
11. $Attr_{Ck} = a_H*(1-P(H, C_k)/H) + a_p[C_k]*(P[C_k][IDS_{Ck}]/P_{max})$;
12. **If** ($Attr_{Ck} > Att_{High}$)**Then**
13. $Att_{High} = Attr_{Ck}$; $C_{winning} = C_k$;
14. **End If**
15. **End For**
16. $DStep_{High}$ = *getDStep*($DS[C_{winning}]$);
17. remPRCs = remPRCs - $DStep_{High}$;
18. $D[C_{winning}] = D[C_{winning}] - DStep_{High}$;
19. $S[N_C] = S[N_C] + DStep_{High}$;
20. *updateHistory*($C_{winning}$, Hist[H]);
21. *moveToNextDStep*($C_{winning}$);
22. **If** ($D[C_{winning}] \le 0$) **Then**
23. $\{C_K\} \leftarrow \{C_K\} \setminus C_{winning}$; //Demand of a core is fulfilled
24. **End If**
25. **End While**
END

Fig 5. Pseudo-code for Resource Allocation

6.2. Comparison with State-of-the-Art

Fairness of Comparison: We compare our Minority-Game-based policy with state-of-the-art reconfigurable multi-core processors [1][4] and different allocation policies (see Section 1.1). The comparison with [1] is analogous to the comparison with the Equal Distribution policy as all cores have the same amount of (dedicated) fabric. The processor in [4] uses fixed-sized fabric partitions and *Statically Mapped Application* tasks (denoted as SMA for ease of further discussion). Considering all this, an approach like SMA is unable to react to run-time varying scenarios. However, for a fair comparison, we have considered the best case where the SMA approach knows the mapping and the difference between our policy and SMA is purely due to the sharing policy. For further fairness, we have provided the same set of hardware accelerators and CIs as well as the same amount of memory bandwidth and the same task mapping to all approaches.

Fig 6 shows the box plot summary of improvement in efficiency of PRC utilization (Eq. 4) and speedup of our policy compared to reconfigurable multi-core processors with a dedicated fabric (e.g., RAMPSoC [1]) such that each core has a same-sized fabric.

[4] Due to exactly one player in the minority side, there is no ambiguity in the decision. Therefore, our proposed MG model is applicable for both odd and even number of players (as it is also used in the extended version of the Minority Games in [5] and [6]) without any ambiguity.

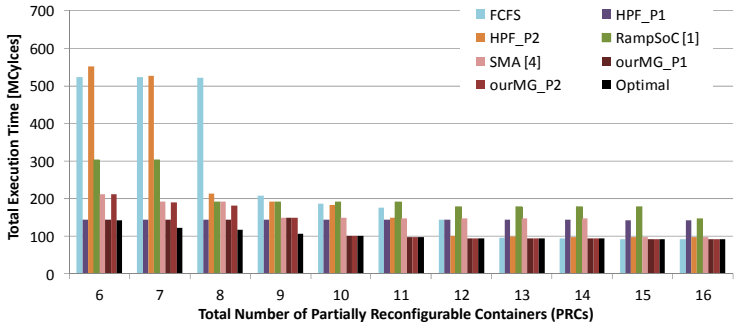

Fig 7. Performance comparison with state-of-the-art reconfigurable multi-core processors and various allocation policies

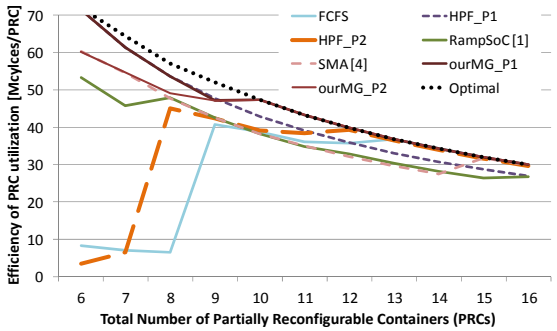

Fig 8 : Efficiency of fabric utilization: comparison with state-of-the-art processors & allocation policies

$$Efficiency = \frac{L_{Saving}}{\#PRCs}\left(\begin{array}{l} L_{Saving} \text{ is the performance improvement} \\ \text{in terms of cycles relative to the execution} \\ \text{without the reconfigurable fabric} \end{array}\right) \quad (4)$$

Fig 6 shows that on average our policy exhibits a 2x higher average utilization compared to RAMPSoC. There are some special cases where our policy achieves an improvement of up to 40x. It is the case of 7 PRCs for task mapping scenario of {CRC, SHA, Susan, ADPCM encoder}, where our policy allocates {1, 1, 3, 2} PRCs to each of the corresponding task. On the contrary, RAMPSoC [1] (due to same number of PRCs in each core) requires at least 12 PRCs (out of which 5 PRCs are unused) to handle the above-mentioned scenario. Therefore, the major improvement of our policy is achieved in cases of fewer PRCs where more fabric is allocated to complex tasks and less fabric to less-complex tasks (as demonstrated in detail by Fig 7). Overall, for the same given number of PRCs, our policy achieves a 5x improved efficiency of PRC utilization (Eq. 4) and a 2x higher performance compared to RAMPSoC.

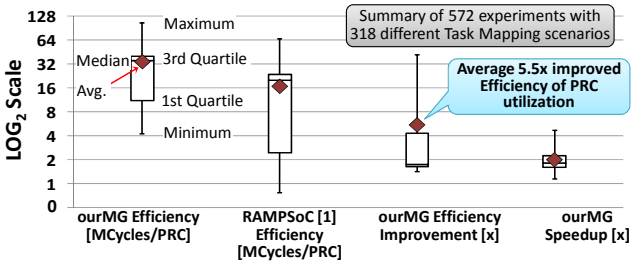

Fig 6. Summary of performance and efficiency of PRC utilization comparison with RAMPSoC [1]

Fig 7 and Fig 8 show a detailed comparison with state-of-the-art for total execution time and the efficiency of PRC utilization for shared fabrics of different sizes. The task-mapping scenario is {CRC, ADPCM encoder, AES encrypt, H.264 encoder}. For comparison with the *Highest Priority First* (HPF) policy, we provide two priority cases here (tasks named left-to-right have priority from high-to-low):

P1) {H.264 encoder, AES encrypt, ADPCM encoder, CRC}

P2) {AES encrypt, ADPCM encoder, H.264 encoder, CRC}

We discuss these comparison of Fig 7 and Fig 8 as follows:

- **Compared to RAMPSoC and SMA**, our policy achieves a performance improvement of up to 1.96x and 1.57x (avg. 1.65x and 1.21x) and up to 24.4% and 24.7% (avg. 16.3% and 10%) improved efficiency of PRC utilization, respectively. The major advantages of our policy can be achieved for an amount of 9-14 PRCs. Thereby, PRCs=8 is a very special case, as all four cores in the SMA approach have PRCs in power of 2 and each core in RAMPSoC also has 2 PRCs. However, our policy still performs better in this case due to its flexibility of the allocation decision.
- **Compared to the First Come First Serve (FCFS) policy**, our policy provides always a better efficiency of PRC utilization. For less than 9 PRCs, the efficiency of FCFS is significantly low as it gives PRCs to "CRC", "AES encrypt", and "ADPCM encoder" first and ignores the fact that "H.264 encoder" has a higher priority and efficiency.
- **Compared to the HPF policy with P1 sequence (HPF_P1)**, our policy (ourMG_P1) provides an equal efficiency of PRC utilization for up to 9 PRCs. Afterwards, HPF_P1 starts loosing as "H.264 encoder" has the highest priority and it monopolizes the entire fabric. Contrarily, ourMG_P1 gives PRCs first to "H.264 encoder" and later due to the history information, it avoids the monopolization.
- **Compared to the HPF policy with P2 sequence (HPF_P2)**, our policy (ourMG_P2) provides a better efficiency of PRC utilization up to an amount of 12 allocated PRCs. This is due to the fact that after 8 PRCs the remaining ones are allocated to "H.264 encoder". After "H.264 encoder" gets more than 4 PRCs, the relative speedup is less compared to the case of a lesser number of PRCs.
- **Compared to the Optimal[5] policy**, our policy with priority case P1 (ourMG_P1) suffers from an efficiency degradation of up to 9.2% (avg. 1.89%). The main difference between *optimal* and ourMG_P1 occurs for less than 10 PRCs. In these cases, the *optimal* policy gives all PRCs to "H.264 encoder" that has the highest efficiency of PRC utilization among all other tasks. However, ourMG_P1 allocates PRCs to other tasks, too. This is because of the consideration of history for *fair* allocation (see Section 5.2). Overall, it shows that our policy achieves a near-to-optimal efficiency of PRC utilization.

Summary: the above comparison illustrates that our policy is better than state-of-the-art in all cases and it is close to an optimal solution. However, compared to the *optimal* policy, our policy achieves on average 2% (worst case 9%) lower performance.

Now, we will present the results of different games under different system dynamics to show the adaptivity of our policy.

6.3. Analysis of Adaptivity under Varying System Dynamics

Fig 9 presents an in-depth analysis using an excerpt of five different games (out of 572) on the time line (t1-t5). In these games a fabric of 12 PRCs is shared between 4 cores (represented by different colors) of a GOC. Furthermore, the task mapping scenarios (matrices) along with their corresponding target speedup (obtained from the performance constraint) and actually achieved speedup (bar graphs) are shown.

(t1) **Competing Game:** four tasks are mapped on 4 cores. "Susan" misses its performance constraint by 8% due to lack of one PRC.

(t2) **Performance Constraint Changed:** the performance constraint of the "H.264 decoder" is increased to CIF@25fps (from CIF@10fps). Here, one PRC is taken from "CRC" and it is allocated to "H.264 decoder". However, since "Susan" and "H.264 decoder" have the same priority, "Susan" does not release its PRCs. Due to this, both "Susan" and "H.264 decoder" tasks miss their deadlines by 8% and 17%, respectively, due to the lack of 1 PRC each. Note that *our policy tends to keep the degradation by the same amount* (each task lacks 1 PRC).

(t3) **Priority Changed:** the priorities of "H.264 decoder" and "CRC" are increased and the priority of "Susan" is decreased. Due to its high priority, the "H.264 decoder" wins more PRCs to meet its performance constraints. Due to a low priority, "Susan" misses

[5] An optimal policy is realized by a Branch and Bound algorithm ignoring its runtime overhead to fix an upper limit of achievable efficiency of fabric utilization.

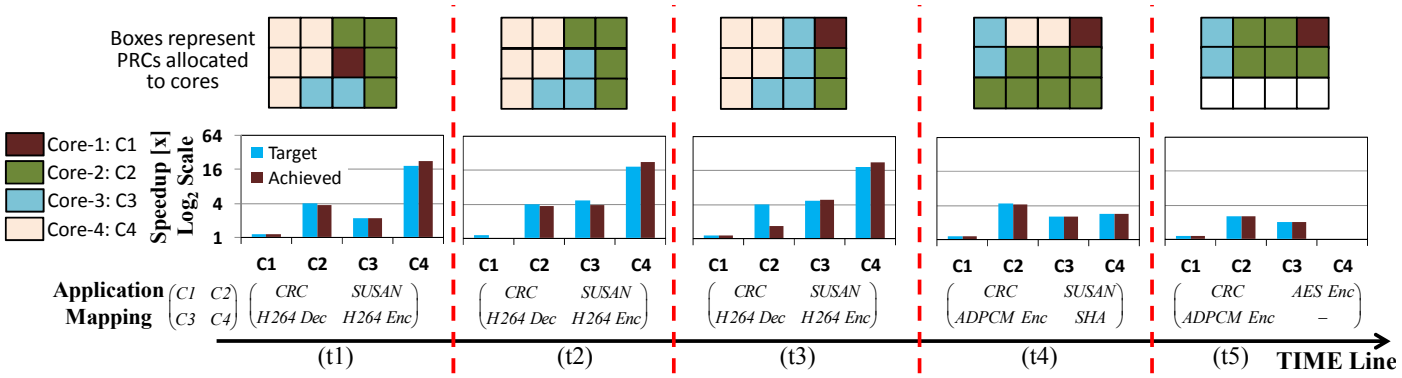

Fig 9. An excerpt of 5 games (out of 572 experiments) showing adaptivity across different games due to varying system dynamics. Shown are the allocated PRCs [colored boxes], target and achieved speedup [bars] of different tasks

its performance constraint by 59%. However, it is acceptable as the priorities of "CRC" and "H.264 decoder" are high.

(t4) **Task Mapping and Performance Constraint Changed:** the "H.264 en-/decoder" tasks are replaced by the "ADPCM encoder" and "SHA", which have a relatively low PRC demand. The performance constraint of "Susan" is increased requiring 9 PRCs. This game is played for 14 PRC but there are only 12 PRCs available. Here, "Susan" misses its deadline by 2.7%. Note that in all of the above cases, the PRC resource has been utilized 100%.

(t5) **Game Skipped:** The game is skipped as the total demand of all tasks is met with 12 PRCs.

6.4. Overhead and Implementation: Hardware Design

	Virtex-4-vlx160 FF1148 @ 50 MHz			
	Latency	Area		
	[Cycles]	Slices	LUT	Gate Equivalent
Memory	21	148	296	38,332
Minority Game	166	739	1,416	14,829
Total	187	887	1,714	53,161

Table 2: Performance & Area overhead of our policy

Table 2 shows the performance and area overhead on a Xilinx Virtex-4-vlx160 FPGA. The hardware area amounts to 887 slices which is 1.6% of the total area of a GOC (4 Leon-II core processors sharing 10 PRCs = 54,976 slices). The hardware implementation has 9 states. *DStep* and *Payoff* values are stored in dedicated memories of #cores * #PRCs * ($\log_2$ #PRCs +32) bits. The computation of the attractiveness is entirely implemented in integer arithmetic. Our implementation uses 3-stage pipeline for the attractiveness calculation. The worst-case execution time consists of the cycles for updating the memories (#PRCs+1) and the game ((#PRCs-2)*(#Cores+5)+4). For 20 PRCs and 4 cores, the performance overhead of one game is 187 cycles, while the execution time of a task is typically in multiples of million cycles (>15 MCycles for the smallest task in our benchmarks). Therefore, the impact of game latency is insignificant on the performance of a task.

7. Conclusion

Our *Minority-Game-based* policy enables efficient and adaptive allocation of shared reconfigurable fabrics in run-time reconfigurable multi-core processors. It provides a high degree of efficiency of fabric utilization. Compared to state-of-the-art reconfigurable multi-core processors (like [1], [4]) and resource allocation policies, our policy achieves on average 2x higher application performance and a 5x improved efficiency of resource utilization. This benefit mainly comes due to the joint consideration of priority, performance constraints, and the history of allocation for the main decision function (Section 5.2) in the Minority-Game. Achieving near-to-optimal efficiency makes our policy a practical solution for the sharing of reconfigurable fabric in a multi-core processor. It is implemented in a low overhead reconfigurable architecture.

8. Acknowledgment

We would like to thank Florian Kriebel (CES, Karlsruhe Institute of Technology) for his contribution to the hardware implementation.

9. References

[1] D. Göhringer et al., "Runtime adaptive multi-processor system-on-chip: RAMPSoC", International Symposium on Parallel and Distributed Processing, pp. 1–7, 2008.

[2] L. Bauer, M. Shafique, J. Henkel, "Run-time instruction set selection in a transmutable embedded processor", Design Automation Conference, pp. 56–61, 2008.

[3] T. Vogt, N. Wehn, "A Reconfigurable Application Specific Instruction Set Processor for Viterbi and Log-MAP Decoding", Workshop on Signal Processing, pp. 142-147, 2006.

[4] M. A. Watkins, M. J. Cianchetti, D. H. Albonesi, "Shared Reconfigurable Architectures for CMPs", International Conference on Field Programmable Logic and Applications, pp. 299-304, 2008.

[5] A. Galstyan, S. Kolar, K. Lerman, "Resource Allocation Games With Changing Resource Capacities", International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 145-152, 2003.

[6] K-M. Lam, H-F. Leung, "An Adaptive Strategy for Resource Allocation Modeled as Minority Game", International Conference on Self-Adaptive and Self-Organizing Systems, pp. 193-204, 2007.

[7] D. Kahneman, A. Tversky, "Prospect Theory: An analysis of decision under risk", Econometrica, 47(2), pp. 263-291, 1979.

[8] E. El-Araby, I. Gonzalez, T. El-Ghazawi, "Virtualizing and sharing reconfigurable resources in High-Performance Reconfigurable Computing systems", International Workshop on High-Performance Reconfigurable Computing Technology and Applications, pp. 1-8, 2008.

[9] A. Gavrilovska et al., "High-Performance Hypervisor Architectures: Virtualization in HPC Systems", Workshop on System-level Virtualization for High Performance Computing, 2007.

[10] H.264 Codec: http://iphome.hhi.de/suehring/tml/index.htm

[11] MiBench: http://www.eecs.umich.edu/mibench/

[12] E. Moro, "The Minority Game: an introductory guide", Advances in Condensed Matter and Statistical Physics, pp. 263–286, 2004.

[13] D. Challet, Y.-C. Zhang, Physica A 246, 407, 1997.

[14] V. Gupta, M. Harchol-Balter, "Self-adaptive admission control policies for resource-sharing systems", International Joint Conference on Measurement and Modeling of Computer Systems, pp. 311–322, 2009.

[15] C. Dumitrescu, I. Foster, "Usage policy-based CPU sharing in virtual organizations", International Workshop on Grid Computing, pp. 53–60, 2004.

[16] L. Bauer, M. Shafique, J. Henkel, "A Computation- and Communication-Infrastructure for Modular Special Instructions in a Dynamically Reconfigurable Processor", International Conference on Field Programmable Logic and Applications, pp. 203-208, 2008.

[17] Q.Cai et al., "Meeting points: using thread criticality to adapt multi-core hardware to parallel regions", International Conference on Parallel Architectures and Compilation Techniques, pp. 240-249, 2008.

[18] R. Metzler, C. Horn, "Evolutionary minority games: the benefits of imitation", Elsevier Physica A: Statistical Mechanics and its Applications, Vol. 329, no. 3, pp. 484-498, 2003.

[19] Y. She, H-F. Leung, "An adaptive strategy for allocation of resources with gradually or abruptly changing capacities", International Conference on Tools with Artificial Intelligence, pp. 415-422, 2008.

[20] W. B. Arthur, Am. Econ. Assoc. Papers and Proc. 84, 406, 1994.

[21] http://www.xilinx.com/technology/roadmap/7-series-fpgas.htm

[22] http://www.xilinx.com/support/documentation/white_papers/wp369_Extensible_Processing_Platform_Overview.pdf