Maximizing the Serviceability of Partially Reconfigurable FPGA Systems in Multi-tenant Environment

Tuan D. A. Nguyen*
duyatuan@acm.org
Technische Universität Dresden
Chair for Processor Design, cfaed
Dresden, Saxony, Germany

Akash Kumar
akash.kumar@tu-dresden.de
Technische Universität Dresden
Chair for Processor Design, cfaed
Dresden, Saxony, Germany

ABSTRACT

In cloud computing, software is transitioning from monolithic to microservices architecture to improve the maintainability, upgradability and the flexibility of the applications. They are able to request a service with different implementations of the same functionality, including hardware accelerator, depending on cost and performance. This model opens up a new opportunity to integrate reconfigurable hardware, specifically, FPGA, in the cloud to offer such services. There are many research works discussing solutions for this problem but they focus primarily on the high-level aspects of resource manager, hypervisor or hardware architecture. The low-level physical design choices of FPGA to maximize the accelerator allocation success rate (called *serviceability*) is largely untouched. In this paper, we propose a design space exploration algorithm to determine the best configuration of partially reconfigurable regions (PRRs) to host the accelerators. Besides, the algorithm is capable of estimating the actual resources occupied by the PRRs on the FPGA even before floorplanning. We systematically study the effects of having more PRRs on the system in various aspects, i.e., serviceability, waiting time and resource wastage. The experiments show that at a certain number of PRRs, upto 91% serviceability can be achieved for 12 concurrent users. It is a significant improvement from 52% without our approach. The average amount of time that each request has to wait to be served is also reduced by 6.3X. Furthermore, the cumulative unused FPGA resources is reduced almost by half.

KEYWORDS

FPGA; cloud; microservice; design automation; design space exploration; partial reconfiguration; floorplan

ACM Reference Format:
Tuan D. A. Nguyen and Akash Kumar. 2020. Maximizing the Serviceability of Partially Reconfigurable FPGA Systems in Multi-tenant Environment. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375305

*Tuan has joined Xilinx Research Labs Asia Pacific after submitting this article.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375305

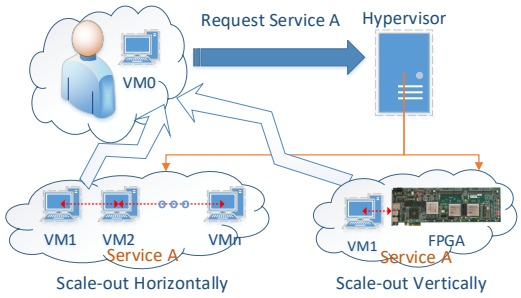


Figure 1: The microservice hypervisor decides whether to allocate conventional Virtual Machines (scaling-out horizontally) or hardware accelerator (FPGA) (scaling-out vertically).

1 INTRODUCTION

The software architecture has been evolving from monolithic to modular architecture. In the former case, the software stack is written as a long stretch of code which is tightly coupled. The functions are highly dependent on each other. The functions' library compatibility also causes difficulties in deploying complicated and large-code-base software. In the modular approach, the software is decomposed into smaller, loosely-coupled (if not independent) pieces called microservices [8]. It addresses the aforementioned problems in monolithic architecture. Another advantage of microservices is scalability. Since the services are deployed independently, they can be replicated quickly to handle the growing processing demands. The services can be mixed and matched with different performance and memory characteristics or even platform from several vendors. The purpose is to utilize the resources more efficiently [26] as long as the communication protocols are compatible [25]. i

The microservices architecture fits really well to the context of utilizing FPGA as an accelerator for an application. Conventionally, a software-based implementation of a service is loaded into memory and CPU when requested. Now, a hardware accelerator, in the form of bitstream, can be configured to the FPGA instead. It runs just as any other microservice as shown in Figure1. The presence of FPGA is transparent to the users [30]. There are many efforts from both industry and research community attempting to incorporate FPGA into the cloud [1, 3, 4, 10, 13, 14, 18, 22, 24, 27, 29, 30, 33, 34]. The most common feature that these works suggest is the use of the partial reconfiguration (PR) [32] to provide the virtualized FPGA-based computing resources.

The aforementioned works cover various aspects of FPGA in the cloud such as **(1)** the high-level hypervisor to provide virtualization service; **(2)** the middle-ware to manage, select and reconfigure the

PR regions (PRRs); **and finally** the system architecture on the FPGA with predefined PRRs. The size of the PRRs, their utilized FPGA resources and their quantity are neither discussed nor analyzed. Those PRRs are mostly fully-mapped, i.e, each of them can host all of the existing accelerators (hereby called PR modules or PRMs). In this case, the valuable FPGA resources will be wasted if the smaller PRMs are used most of the time. Additionally, the larger these PRRs are, the fewer of them that can be implemented on a fixed-size FPGA. The number of concurrent users (or tenants) who can have access to the FPGAs becomes lower. In other words, if the number of tenants is kept the same, the access requests from them will be rejected more regularly. It results in low **serviceability**, the ratio of the serviceable requests. Thus, the number, sizes of PRRs and the mapping of PRMs to PRRs should be optimized to maximize the serviceability of the system. As a result, the mapping of PRMs to PRRs should be heterogeneous instead of homogeneous.

It is a common knowledge that having more PRRs will deliver higher parallelism. Nevertheless, as the number of PRRs and PRMs increase, it becomes harder and almost impossible to analyze and implement the system manually. The vendor system architect (hereby called architect) has to simultaneously determine the best number of PRRs, the mapping of PRMs to PRRs while making sure that they can physically fit in the designated FPGA. Regarding the last task, the architect has to do actual floorplanning for the PRRs on the FPGA based on the PRM-PRR mapping. However, the current PR development flow requires him/her to do that manually [32]. It is especially tedious at the design space exploration (DSE) stage where the system architecture is not yet finalized. Therefore, in this work, we propose a novel automation tool to consider all of these aspects. Our contributions are as followed.

- **An optimization algorithm** to perform the DSE to find the best possible number of PRRs and the mapping of the PRMs to those PRRs (one PRM can be mapped to multiple PRRs) based on the existing PRM pool.
- **An estimation method** to quickly estimate the actual FPGA resources occupied by the PRRs based on the requested resources during the mapping process. We also integrate an automatic floorplanner [19] after mapping to verify that the resulting system can physically fit in the designated FPGA.
- **A lightweight simulator** to assess the serviceability of the system under test during the DSE.

The contributions presented in this work are not limited to microservice-based applications. They are applicable to all types of PR-based dynamic systems such as [6, 16]. In these systems, the list of hardware accelerators is known at design time. However, their actual usage is only known and determined at runtime. The behavior of those applications at runtime resembles the concurrent users in the cloud environment. Our approach can be used as another layer of optimization during the architectural design phase.

We focus primarily on studying the advantages of having more PRRs and how to map the PRMs to them. Therefore, we need a flexible PR-based system template that can be generated with varying numbers of PRRs. Besides, the relative resource requirements of the PRMs to the FPGA should not be too high. It is to make sure that a large number of PRRs that can be implemented. These requirements are essential to assess the scalability of our mapper.

The experiments are carried out with the architecture template taken from [20]. Since their target FPGA is Xilinx Virtex-6, we assume the same chip throughout the article. We collect a set of 50 real-world hardware accelerators synthesized by Xilinx Vivado HLS version 2016.3 and ISE version 14.7 from the different publicly available sources [9, 12, 21, 31]. The results show that the best system we can find is the one with 11 PRRs. It can maintain a serviceability of more than 91% for up to 12 concurrent users. On the other hand, only 52% serviceability can be achieved for the homogeneous system with 4 PRRs. The average time the tenants have to wait to be served is reduced by 6.3X. Besides, the cumulative wasted FPGA resources are reduced by 50%. It could potentially improve the energy efficiency of the system. More importantly, during the DSE process, we find that the serviceability only increases with the number of PRRs up to a certain point; after that, it degrades. Therefore, it is very important to perform our proposed DSE to find that sweet spot for a specific system and usage scenarios.

The remaining paper is organized as follows. The recent works on PR for shared FPGA in the cloud are discussed in Section 2. The proposed approach is presented in Section 3, followed by experimental results in Section 4. Finally, the conclusions and future works are presented in Section 5.

2 RELATED WORKS

The integration of FPGA in cloud computing to accelerate computation is becoming a major trend in both industry and academia. In this section, the related works are discussed in more details.

Chen et al. [4] discuss four major issues in enabling FPGAs into the cloud. They are abstraction layers, sharing of resources, compatibility between FPGA tool chains and finally security. The main contribution of [4], similar to [1, 18, 27, 34], is to propose a general framework and guidelines to tackle those issues, focusing on the high-level layer where the resource manager, scheduler and hypervisor run. However, the authors overlook the partitioning details of the accelerator slots, i.e, the PRRs, which also affect the serviceability of the system when users request the resources.

Published at the same time as [4], the work by Byma et al. [3] also proposes a general framework to integrate FPGA with PR hardware accelerators into existing cloud computing models. The PRRs are considered as generic cloud resources in OpenStack to provide seamless FPGA virtualization similar to regular virtual machines. Even though the authors acknowledge the need to have mixed size PRRs to improve the flexibility of the system and propose it as future work, no further analysis has been done.

The cloud management and hypervisor named RC3E developed by Knodel et al. [14] offers multiple models for utilizing FPGA, either full access to the reconfigurable resource (entire FPGA), *Reconfigurable Silicon as a Service*; or only part of it with the introduction of PR virtual-FPGAs, *Reconfigurable Accelerators as a Service* and *Background Acceleration as a Service*. In the last two cases, an FPGA can host up to 4 virtual-FPGAs. The work by Weerasinghe et al. [29] takes an alternative approach by not only utilizing virtual FPGA concept but also proposing an infrastructure to allow large-scale deployment of FPGAs across the cloud. Nevertheless, as in the case of [14], further information about the virtual FPGA physical implementation is not provided.

Fahmy et al. [10] present a framework for cloud computing with virtualized FPGA accelerators in a similar method as [14]. The authors do suggest partitioning the FPGA into various-sized PRRs and use a greedy approach to allocate the hardware accelerators to PRRs. For each accelerator usage request, the smallest PRR that can host it is reserved. The purpose is to maximize the possibility of configuring the larger accelerators for the later requests. If there is no such free PRR, the request is rejected and processed in software. Unfortunately, the impact of the size and the number of PRRs on the success rate of serving the requests is not analyzed. The authors also do not discuss how the PRMs are mapped to the PRRs.

The hypervisor proposed in [30] provides insight into how the PRMs/PRRs can be managed with a similar approach to the software-based system. The performance and operation of the hypervisor are assessed on the PR system with 3 PRRs. These PRRs have different sizes. Each of them can only host a subset of the PRMs used in the experiments. Similar to [10], the authors do not study the impact of the PRRs to the overall performance of the system.

Zhao et al. [33] introduce the hardware project management and building tool called hCode2.0. It provides an easy-to-use framework to map and generate partial bitstreams for the accelerators within a shell. A shell is a system architecture with placeholders for PRRs. When a new accelerator is imported to a particular shell, it will be mapped to all of the available PRRs which have sufficient resources. Unfortunately, the tool needs a pre-defined system architecture as input with already-placed fixed number of PRRs.

The work [13] approaches the problem from a different perspective. Instead of having multiple PRRs on one FPGA for the tenants to share, they target the case where the tenants need one common accelerator. They propose a load balancing and monitoring framework to manage the bandwidth and the request rates from the tenants. However, this method is only suitable for the servers with one dedicated acceleration service. It may not be applicable for a general microservice environment.

In the embedded systems domain, there are several similar attempts in trying to find a suitable system architecture for the applications [5, 7, 23]. These works start by analyzing the task graph of the application. After that, they optimize the mapping and scheduling of those tasks (with the corresponding PRMs) on either software/FPGA or FPGA-only systems with different numbers of PRRs. However, their methods stop mapping the PRMs to PRRs once a feasible schedule is found. Our method, on the other hand, does not work at the task graph because it is not known at design time. We instead optimize the distribution of PRMs to PRRs to maximize the chance of finding a compatible PRR for a PRM at runtime.

All these works discuss many interesting high-level and system-architecture aspects of integrating FPGAs as virtualized resources into the existing cloud infrastructure. Still, the idea of improving the success rate in allocating accelerators upon requests from users by optimizing the number of PRRs as well as their sizes is untouched.

3 PROPOSED APPROACH

3.1 Design Space Exploration

The proposed DSE flow chart is illustrated in Figure2. There are six major steps. It starts by obtaining the PRMs' resources requirement (Step 1). The requests from each tenant are generated randomly in

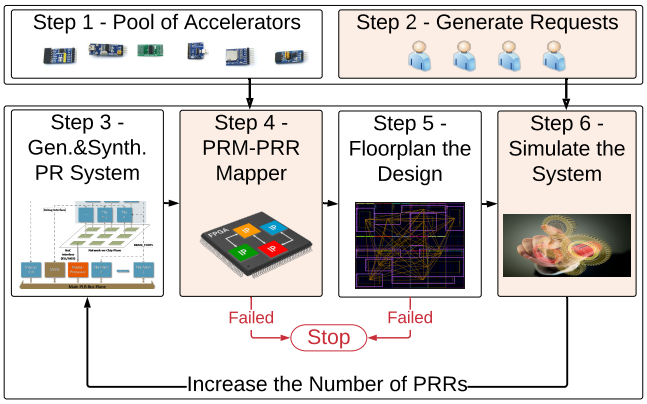


Figure 2: Our proposed DSE flow used to find the optimal configurations for PRRs from the accelerator pool and system template. Our contributions are in light orange.

Step 2 (described in Section 3.4). The tenants in the cloud environment are independent of each other [30]. Thus, we can mimic the behaviors of multiple tenants by simply combining their requests.

Thereafter, in Step 3, the PR-HMPSoC template provided by Nguyen et al. [20] is used to create the PR systems with different number of PRRs. The choice of PR-HMPSoC is for the experimental purposes; it does not represent the actual cloud-based system. None of the works in [3, 4, 10, 14, 29] is designed as a flexible template to have systems with a varying number of PRRs. Therefore, we are unable to use in the experiments. Nevertheless, our DSE approach is agnostic to the underlying architecture.

Afterwards, each PR system is synthesized on Virtex 6 by Xilinx ISE 14.7 to get the real resources requirement of each component from Step 3. This information and the one obtained from Step 1 is fed to our PRM-PRR mapper presented in Section 3.2 and 3.3 to calculate the best mapping of PRMs to the PRRs (Step 4). The tenants' requests are not known a priori by the PRM-PRR mapper.

Next, we use PRFloor tool [19] to floorplan the system (Step 5). This step is required to make sure that the resulting system can physically fit in the FPGA. It may happen that the PRM-PRR mapper is not able to determine a proper mapping. The PRFloor may also fail to find a feasible floorplan. In both cases, it is due to the limited FPGA resource capacity. The PRM-PRR mapper always tries to map at least 2 PRMs to each PRR. If the number of PRRs is too high, the total resources required by the system may be larger than the FPGA. In floorplanning, the more PRRs are requested, the larger the system is with many more supporting static components. It is more difficult for PRFloor to successfully find a floorplan.

Step 6 is executed when both PRM-PRR mapper and PRFloor have run successfully. Our simulator takes the requests and system specification (list of PRMs, PRRs and PRM-PRR mapping) generated from Step 2 and 5 respectively to simulate the system. The quality metrics (serviceability, waiting time and resource wastage) of the current system are then recorded as one *design point* to compare with the others. The simulator is presented in Section 3.4. Finally, the requested number of PRRs is increased to go through another iteration with a new system architecture.

When the system architecture and the PRM-PRR mapping are finalized, a TCL script can be used to instruct Xilinx tool to generate the partial bitstreams. This process may take a significantly long time to complete. This problem can be alleviated by using bitstream-relocation-aware PRM-PRR mapper, floorplanner and related low-level techniques. However, it is not considered in this work.

There might be a concern about having too many processing elements. It will affect the overall performance of the system. Specifically, the memory/peripheral access contention may increase. Our approach is made such that it does not concern about the internal architecture. Since a system generator knows best about its architecture, it should have its own performance analysis. When those metrics are worsen, an invalid system should be returned. Our DSE flow will stop processing further. Those metrics can be combined with ours to explore the Pareto multi-objective optimization.

3.2 PRM-PRR Mapper

The problem we are trying to solve in this work is to serve as many hardware accelerator requests as possible. However, since the FPGA resource is limited, it is impossible to implement all accelerators on the FPGA. Conventionally, the architect tried to analyze the application use-cases to generate a set of predefined FPGA configurations; each contains a set of accelerators. When a request for the accelerators is received, one bitstream is chosen to reconfigure the whole device [15]. This method is impractical in the cloud environment where the behaviors of the tenants are nondeterministic [30]; even if they are, the number of use-cases will explode exponentially. Furthermore, triggering the full reconfiguration will interrupt other tenants who are sharing the same FPGA.

As a result, PR systems are the most suitable platforms in this case. The FPGA is partitioned into multiple slots (called PRRs), the accelerators (or PRMs) can be dynamically loaded into those slots when needed. Unfortunately, there are several technical challenges. The PRRs must be specified at design time [32]. The architect has to decide which PRRs one PRM can run onto to define the sizes and required resources of the PRRs appropriately. The mapping decision is affected by many factors such as how often the PRMs are used, how many PRRs that each PRM should be mapped to, what types of resources that each PRM requires, how many resources are actually occupied by the PRRs after floorplanning, etc. For this reason, we propose the automatic PRM-PRR mapper. The optimization goal of the mapper is to map one PRM to as many PRRs as possible to maximize the chance of finding a suitable PRR when that PRM is requested. Since the PRRs are heterogeneously mapped, there are more PRRs compared to the homogeneous case. Hence, more tenants can be served at the same time.

Based on the characteristics of the goal, we model it as an Integer Linear Programming (ILP) problem [28]. We define the function $F(PRM_i) = F_i$ to represent the number of PRRs that PRM_i is mapped to. The naive objective is to maximize the sum of these F_i as shown in Equation 1.

$$\sum_{i=1}^{n} F(PRM_i) \qquad (n \text{ is the number of PRMs}) \tag{1}$$

There are three issues with this objective.

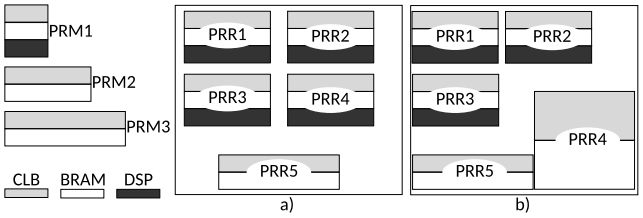


Figure 3: The issue of one PRM is being mapped to too many PRRs compared to the others. In a), PRM1 is mapped to PRR1-4 while PRM3 is only mapped to PRR3. In b), a better mapping, PRM1 is mapped to PRR1-3, PRM3 is mapped to PRR4-5.

- **Issue 1**: the architect, after analyzing the tenant usage behaviors, may observe that several PRMs are more common than the others. It would be better to increase their F_i.
- **Issue 2**: some PRMs can be mapped to too many PRRs compared to others, affecting the fair share between PRMs to FPGA resource as illustrated in Figure3. In Figure3**a**, PRR1–4 can host PRM1 and PRM2; PRR5 can host PRM2 and PRM3. If PRM1 is mapped to PRR5, there will not be enough DSP because PRR5 must cover a larger region. Here, $F(PRM1) = 4$, $F(PRM2) = 5$, $F(PRM3) = 1$. In Figure3**b**, PRR1–3 can host PRM1 and PRM2, PRR4–5 can host PRM2 and PRM3; therefore $F(PRM1) = 3$, $F(PRM2) = 5$, $F(PRM3) = 2$. The total number of $F(PRM_i)$ in both cases is 10. However, in Figure3**b**, PRM3 has more chance of finding a suitable PRR to load to.
- **Issue 3**: if the numbers of PRMs mapped to each PRR are not balanced, some PRRs will become hot-spots where they are used more frequently than others. It will potentially cause more resource contentions.

Consequently, the final ILP program is built as follows. The objective function is presented in Equation 2. The most important part of the ILP constraint section is described by Equations 3 to 5.

Minimize:

$$-\alpha * \sum_{i=1}^{n} \frac{F(PRM_i)}{priority_i} + \beta * \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} DEV_{PRM_{ij}} + \gamma * \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} DEV_{PRR_{ij}} \tag{2}$$

Subject to:

$$F(PRM_i) = \sum_{j=1}^{m} MAP\_PRM_i PRR_j >= 1 \tag{3}$$

$$G(PRR_j) = \sum_{i=1}^{n} MAP\_PRM_i PRR_j >= 2 \tag{4}$$

$$ALL\_PRR_{CLB} = \sum_{i=1}^{m} PRR_{i_{CLB}} <= MAX_{CLB} \tag{5}$$

The first term of Eqn. 2 introduces a new measure of priority for each PRM, $priority_i$, to solve the aforementioned **Issue 1**. The highest possible $priority_i$ is 1. Therefore, the lower the priority (higher number) is, the smaller the impact of the corresponding PRM on the summation of $F(PRM_i)$.

The second and third terms of Eqn. 2 address the last two issues respectively. The $DEV_{PRM_{ij}}$ measures the difference, or deviation, between the number of PRRs that PRM_i and PRM_j are mapped to. The intrinsic idea is that, minimizing $DEV_{PRM_{ij}}$ ($\forall i \neq j, i, j = 1 \rightarrow n$) will balance the values of $F()$ between all PRMs. By this way, the second issue can be avoided. For instance, in Figure3, the total deviation of PRMs in example **a** is **8** while being just **6** in **b**. The same method is applied to the third issue related to PRRs by using $DEV_{PRR_{ij}}$ ($\forall i \neq j, i, j = 1 \rightarrow m$, m is the number of PRRs). Equations 6 to 9 illustrate how to compute these deviations as constraints in the ILP program.

$$DEV_{PRM_{ij}} - F(PRM_i) + F(PRM_j)) >= 0 \quad (6)$$

$$DEV_{PRM_{ij}} - F(PRM_j) + F(PRM_i)) >= 0 \quad (7)$$

$$DEV_{PRR_{ij}} - G(PRR_i) + G(PRR_j) >= 0 \quad (8)$$

$$DEV_{PRR_{ij}} - G(PRR_j) + G(PRR_i) >= 0 \quad (9)$$

Eqn. 3 shows the calculation of the number of PRRs that PRM_i is mapped to. The number of PRMs that PRR_j can host is described in Eqn. 4. The set of binary variables $MAP\_PRM_iPRR_j$ ($\forall i = 1 \rightarrow n, j = 1 \rightarrow m$) indicates the mapping of PRM to PRR. If $MAP\_PRM_iPRR_j = 1$, then PRM_i is mapped to PRR_j. We rely on these variables to determine the final PRM-PRR mapping.

Eqn. 5 is used to restrict the total number of CLBs occupied by all PRRs from exceeding the available CLBs (after deducting the static modules). The requested number of CLBs of PRR_i, $PRR_{i_{CLB}}$, is the largest requested number of CLBs among all PRMs that are mapped to PRR_i. Eqn. 10 – 11 present the computation of $PRR_{i_{CLB}}$ from two representative PRMs, PRM_j and PRM_k. The same computation is applied for CLBM, BRAM and DSP.

$$PRR_{i_{CLB}} - PRM_{j_{CLB}} * MAP\_PRM_jPRR_i >= 0 \quad (10)$$

$$PRR_{i_{CLB}} - PRM_{k_{CLB}} * MAP\_PRM_kPRR_i >= 0 \quad (11)$$

In Eqn. 2, there are three weight parameters, α, β and γ. They are used to balance the preference of the architect over three objectives. To make it easier to adjust these parameters, each objective should be normalized to its corresponding maximum possible value. The first objective, maximizing all $F(PRM_i)$, reaches its maximum when each PRMs can be mapped to all PRRs, hence the value is $m \sum_{i=1}^{n} priority_i^{-1}$.

Calculating the maximum values for the deviation metrics is not as straight forward. Eqn. 12 – 13 are the simplified functions used to calculate the total deviations of $DEV_{PRM_{ij}}$ and $DEV_{PRR_{ij}}$. It is assumed that $F(PRM_i)$ is sorted in the decreasing order of value when $i = 1 \rightarrow n$. The same assumption is applied for $G(PRR_i)$.

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} DEV_{PRM_{ij}} = (n+1) * \sum_{i=1}^{n} F(PRM_i) - 2 * \sum_{i=1}^{n} i * F(PRM_i) \quad (12)$$

$$\sum_{i=1}^{m-1} \sum_{j=i+1}^{m} DEV_{PRR_{ij}} = (m+1) * \sum_{i=1}^{m} G(PRR_i) - 2 * \sum_{i=1}^{m} i * G(PRR_i) \quad (13)$$

In our emperical analysis, the total deviations of all PRMs get its maximum value when half of the PRMs is mapped to all PRRs; and the other half is mapped to only 1 PRR. A similar observation can be drawn for $DEV_{PRR_{ij}}$. Therefore, Eqn. 12 – 13 are simplified to Eqn. 14 and 15.

$$MAX(\text{Eq. 12}) = \begin{cases} 0.25n^2(m-1), \text{ n is even} \\ 0.25n'(n'+2)(m-1), \text{ n is odd, } n' = n-1 \end{cases} \quad (14)$$

$$MAX(\text{Eq. 13}) = \begin{cases} 0.25m^2(n-1), \text{ m is even} \\ 0.25m'(m'+2)(n-1), \text{ m is odd, } m' = m-1 \end{cases} \quad (15)$$

After calculating the maximum values for three objectives, the parameters α, β and γ are then adjusted as in Eqn. 16. The architect balances the weights via $\alpha\prime, \beta\prime$ and $\gamma\prime$.

$$\alpha = \frac{\alpha\prime}{m \sum_{i=1}^{n} priority_i^{-1}}; \beta = \frac{\beta\prime}{\text{Eq. 14}}; \gamma = \frac{\gamma\prime}{\text{Eq. 15}} \quad (16)$$

3.3 Estimate Occupied Resources

In PRFloor work [19], the authors explain that the actual FPGA resources occupation of the PRRs can be very different from the initial requirements. It is due to the heterogeneity and non-uniform distribution of the FPGA resources. If the PRM-PRR mapper does not have the notion of this issue, it may unknowingly over-assign (assign more than it should be) the PRMs to the PRRs. Consequently, the floorplanner will fail to find a feasible floorplan for the system because the PRRs become too big. The DSE process described in Section 3.1 may terminate prematurely. As a result, the potential optimal design points later are lost.

We propose a method to estimate the actual occupied resources for PRRs based on the initial requirements. The pseudo-code used to estimate the resources occupation of a PRR is shown in Algorithm 1. This algorithm is only run once for each type of FPGA. The basic idea is to apply the curve fitting algorithms to determine the best-fit functions whose the inputs are the number of requested resources. In line 4, a placement of a PRR is the smallest rectangle region (starts from any location of FPGA) that covers at least the same amount of resources requested by that PRR. The algorithm finds all placements for the PRR from every possible location of the FPGA. In this work, the fitting functions found from line 9—12 are obtained by using the *Matlab Curve Fitting Toolbox* [17]. These estimations are used in constraint section of the PRM-PRR mapper (Eqn. 10 – 11) for the PRMs instead of the resources from the synthesis reports. The reason we can use the estimation directly to the PRMs instead of PRRs is explained as follows. The required CLB resource for the PRR in which PRM_i, ($i = 1 \rightarrow k$) are mapped to, is: $PRR_{req_{CLB}} = \max PRM_{i_{req_{CLB}}}$. The estimated resource occupation of PRR is: $est_{PRR_{i_{CLB}}} = est(\max PRM_{i_{req_{CLB}}}) = \max est(PRM_{i_{req_{CLB}}})$. The accuracy of the algorithm is discussed in Section 4.3.

Algorithm 1 Estimate the actual resources occupation

Require: $FPGA_{clb}, FPGA_{clbm}, FPGA_{bram}, FPGA_{dsp}$
1: **for all** rsrc ∈ {clb, clbm, bram, dsp} **do**
2: generate PRRs which require $num_{rsrc} = 1 \rightarrow FPGA_{rsrc}$, $num_{remaining\_rsrcs} = 0$
3: **for all** PRR **do**
4: find all possible placements on the FPGA
5: got mean: $mean\_clb(num_{rsrc})$, $mean\_clbm(num_{rsrc})$, $mean\_bram(num_{rsrc})$, $mean\_dsp(num_{rsrc})$
6: **end for**
7: **end for**
8: determine 4 fitting functions to estimate the occupied CLB
9: got: $est\_clb\_from\_clb(num_{clb}) \approx mean\_clb_{num_{clb}}$
10: got: $est\_clb\_from\_clbm(num_{clbm}) \approx mean\_clb_{num_{clbm}}$
11: got: $est\_clb\_from\_bram(num_{bram}) \approx mean\_clb_{num_{bram}}$
12: got: $est\_clb\_from\_dsp(num_{dsp}) \approx mean\_clb_{num_{dsp}}$
13: // *estimate the CLB from* $req_{clb}, req_{clbm}, req_{bram}, req_{dsp}$
14: $est\_clb(req_{clb}, req_{bram}, req_{dsp})$
$= \max(est\_clb\_from\_clb(req_{clb}),$
$est\_clb\_from\_clbm(req_{clbm}),$
$est\_clb\_from\_bram(req_{bram}),$
$est\_clb\_from\_dsp(req_{dsp}), req_{clb})$
15: similar calculation is used to compute the other resources est_clbm, est_bram and est_dsp

3.4 Request Generator and Simulator

In the microservice-based environment, when the tenants request for services with a specified workload, performance requirement and cost preference, the hypervisor takes responsibility to process the requests. It decides whether to scale out horizontally, allocating conventional virtual machines (VMs) to serve the requests, or to scale out vertically, choosing a VM with a suitable processing elements such as FPGA (Figure1). This process is transparent to the tenants. Each tenant works independently of each other. The hypervisor can assign multiple FPGA slots (PRRs) to the tenants. It may happen that there is no available PRR left. The hypervisor has to either wait for the other tenants to release the PRRs within a predefined timeout period, or allocate the conventional VMs [10]. These behaviors of the tenants and hypervisor are considered here.

3.4.1 Request Generator. We propose a request generator shown in Algorithm 2to mimic the behaviors of the tenants. The requests from each tenant are generated independently of the others. For each request, the required service (or PRM), the *start time* when the request must be served and the *duration* that the tenant will use the service upon successful allocation are randomly generated. Additionally, the generator can specify the timeout period during which the hypervisor is allowed to postpone the request to wait for a suitable PRR. The maximum number of PRMs that one tenant is allowed to ask for at any particular time is configurable (the *concur_prm* parameter in line 4).

The requests parameters are uniformly generated. However, in practice, it is uncommon for the tenants to use all kinds of services from different application domains. Each tenant is only primarily interested in a particular domain. Our request generator takes that into account by offering an option to specify the primary domain for each tenant. In that case, most of the services requested by the tenant will be from that domain (line 10 of Algorithm 2).

Algorithm 2 The Request Generator

Require: num_req, $priority$, $primary\_domain$, $duration_{min}$, $duration_{max}$, $timeout_{max}$, $concur\_prm$, $window_{length}$
1: $i = 0$; $window_{start} = 1$; $start\_time_{prev} = 1$
2: **while** $i < num\_req$ **do**
3: $latest\_end\_time = 0$
4: **for** $j = 1$ **to** $concur\_prm$ **do**
5: // *gen_req() randomly decides whether to generate a new request*
6: **if** $gen\_req() == 0$ **then** continue
7: $gen\_start\_time(req_i, window_{start}, start\_time_{prev})$
8: // $start\_time >= \max(window_{start}, start\_time_{prev})$
9: generate $duration$, $timeout$
10: generate app_domain
11: generate $req\_prm \in app\_domain$
12: $expected\_end = start\_time + duration + timeout$
13: **if** $latest\_end\_time < expected\_end$ **then** update it
14: $i = i + 1$
15: $start\_time_{prev} = start\_time$
16: **end for**
17: $window_{start} = window_{start} + window_{length}$
18: **if** $window_{start} < latest\_end\_time + 1$ **then** update it
19: **end while**

Algorithm 3 The Simulator

Require: $list\_of\_tenants, list\_of\_prm, list\_of\_fpga$
1: $global\_tick = 0$; $queue_{cur\_req} \leftarrow \emptyset$
2: **while** all requests are not served **do**
3: advance $global\_tick$
4: $remove\_all\_timed\_out\_requests(queue\_cur\_req)$
5: **for all** $cur\_tenant \in list\_of\_tenants$ **do**
6: $free\_completed\_requests(cur\_tenant)$
7: **while** $(req = pop\_next\_req(cur\_tenant, global\_tick))$ **do**
8: $push\_to\_queue(queue\_cur\_req, req)$
9: **end while**
10: **end for**
11: $serve\_requests(queue\_cur\_req)$
12: **end while**
13: report quality metrics

3.4.2 Simulator. Our simulator is developed to act as a simplified hypervisor serving the requests from the tenants. Algorithm 3 describes how it works. One simulation time is called a *tick*.

The $list\_of\_tenants$ and the corresponding requests are provided by the request generator. The $list\_of\_fpga$ and the PRRs detailed information and the PRM-PRR mapping are obtained from the previous steps of the DSE flow. The $remove\_all\_timed\_out\_requests()$ in line 4 removes all timed-out requests that the hypervisor defers to serve because there was no suitable PRR at the time they arrived. The provided timeout periods are generated randomly by the request generator. The function $serve\_requests()$ serves the requests in the following order of *(1)* increasing $start\_time$ and *(2)* tenant's priority (the $list\_of\_tenants$ is sorted based on the priority). On the mapping aspect, the PRR assigned to the request is chosen such that *(1)* it can host the requested service and *(2)* it is the smallest available PRR. The mapper and scheduler could be developed further with a more sophisticated strategy to take other metrics into account such as application performance or energy efficiency.

4 RESULTS

4.1 Experiment setup

All of our experiments are run on a computer with CPU Intel $Core^{TM}$i5, 2.5 GHz x4 (2 physical cores with hyper-threading) and 12GB of memory. The operating system is Ubuntu 14.04 LTS 64-bit. The PR-HMPSoC template and PRFloor are provided by [19, 20]. Even though our method is made general enough for all kinds of Xilinx FPGA, the one we are experimenting with (due to the restriction of [20]) is Virtex-6 XC6VLX240T. Gurobi Solver [11] is used to solve the PRM-PRR mapper with default settings. The weight parameters in the PRM-PRR mapper objective function are $\alpha\prime = 5$, $\beta\prime = \gamma\prime = 1$ (unless stated otherwise).

The request generator generates the requests from tenants based solely on the information of the PRMs, or the IP pool, and the tenants' configuration as discussed in Section 3.4.1. In our experiments, all PRMs have equal priority unless stated otherwise. Each tenant issues 10000 requests and can use up to 3 PRRs at any one time. The PRMs are chosen randomly with uniform distribution, regardless of the application domain. The usage duration of each PRM upon successful allocation is randomly generated between 10 and 400 simulation ticks. The *window_length* parameter in Algorithm 2 is set to 500 ticks. The number of tenants in each simulation is from 2 to 12 in the increment of 2. We have two sets of tenant requests, the first one disables the timeout mechanism for the requests that cannot be served immediately as discussed in Section 3.4.2; the second one enables that with the random timeouts of upto 200 ticks.

4.2 IP Pool

We collect 50 real-world hardware accelerators, or PRMs, from CHStone [12], Opencores [21], EPFL [9] and Xilinx XPS IP core library [31]. These PRMs are categorized into 8 application domains: digital signal processing (DSP), cryptography, arithmetic, communication, soft-core processing unit, image processing, video processing and others. Figure4 depicts the resources requirements of the PRMs. As seen, the sizes and types of resources of the PRMs vary quite significantly which reflect microservice scenarios.

4.3 Accuracy of the Resource Estimation

In this section, the accuracy of the resource estimation method presented in Algorithm 1 is assessed. We randomly generate 10000 PRRs whose sizes can be up to 80% of the device. Afterwards, the mean resources occupied by each PRR are computed by the floorplanner. The error histogram of the requested resources compared against the actual occupation on the device is provided in Figure5a. The Algorithm 1 is then used to estimate the occupied resources. The corresponding error histogram is illustrated in Figure5b. The results in two figures indicate that our algorithm does offer a highly-accurate estimation on the final resource occupations. The error of the CLB and CLBM estimations are mostly within the 5% range, at most 10%. In the case of BRAM and DSP, there is a small amount of outliners that are more than 20% off from the expected values. This high error is due to the irregular distribution of the resources, especially BRAM and DSP, on the FPGA. This irregularity causes large variations in the size of the placements.

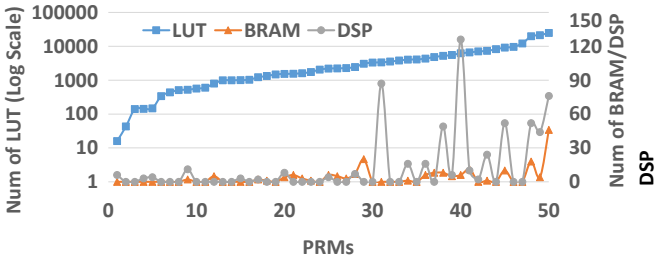
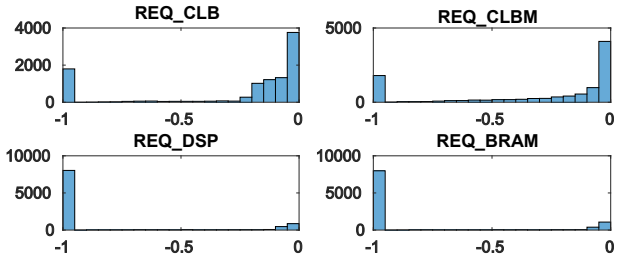
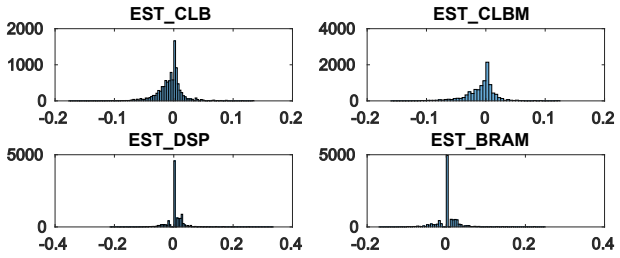


Figure 4: The resources distribution of 50 PRMs in the experiments.



(a) Requested resources vs. actual occupied resources



(b) Estimated resources vs. actual occupied resources

Figure 5: The error histograms of the requested and estimated resources compared against the actual mean resources occupied by PRRs after floorplanning.

4.4 Design Space Exploration

4.4.1 PRM-PRR Mapper. During the DSE process, the PR systems with heterogeneous PRRs are compared against the baseline. The PRRs in this baseline system are homogeneous. Each of them can host any of the PRMs presented in Section 4.2. The architecture of the baseline system is obtained naturally by just executing the DSE flow. If it is possible to fully map all PRMs to every PRR in the system, the PRM-PRR mapper will converge to that point, thanks to the formulation of the objective function shown in Eqn. 2. The baseline system returns the best possible objective value among the systems with the same number of PRRs. It is $-m * n$ in which m and n are the number of PRRs and PRMs respectively. All generated systems are later simulated with the same set of tenant requests.

We generate the PR system from the PR-HMPSoC template starting with 3 PRRs. In these experiments, the PRM-PRR mapper restricts the CLB (including CLBM), BRAM and DSP utilization of both PRRs and static modules to 85% of the Virtex 6. We keep increasing the number of PRRs until PRM-PRR mapper or PRFloor

Table 1: The mapping and floorplanning results obtained for the PR systems after the DSE.

Num PRRs	Map Time (s)	Floorplan Time (s)	Optimal	$\bar{F}(PRM)$	$\bar{G}(PRR)$
3	0.05	16	✓	3.0	50.0
4 (base)	0.04	17	✓	4.0	50.0
5	0.87	85	✓	4.9	48.6
6	1.91	29	✓	5.6	46.8
7	2.27	44	✓	6.4	45.7
8	5.01	46	✓	7.2	44.8
9	43.67	59	✓	7.6	42.4
10	101.47	175	✓	8.1	40.7
11	119.72	85	✓	8.6	39.1
12	122.84	103	✓	8.9	37.0
13	300.00	95	✗	9.2	35.2
14	300.00	108	✗	9.4	33.7
15	300.00	104	✗	9.1	30.5

fails to find a feasible floorplan. However, during the DSE, we notice a decline in the serviceability of the systems and decide to stop the DSE sooner. At this point, we have already obtained the system with up to 15 PRRs. Table 1 presents the time the PRM-PRR mapper takes for each system. The floorplanning time, the average values of $F(PRM)$ and $G(PRR)$ are also given. From the table, the baseline system has 4 PRRs. This baseline system has the same number of PRRs as our most similar related work [10]. During the DSE, there are only a few cases where the final resources occupation after floorplanning exceed the 85% constraint.

All mappings except the systems with more than 12 PRRs given by PRM-PRR mapper are proved optimal, i.e, the objective functions are mathematically proved by Gurobi that they get the lowest possible values. For the first 6 systems, it takes less than 5 seconds for PRM-PRR mapper to find an optimal solution. However, in the subsequent cases with 9 to 12 PRRs, the mapper has to spend nearly 2 minutes. For the system with 13 to 15 PRRs, the ILP solver stops exploring to find the optimal solutions because the timeout is set to 5 minutes. The thorough investigation of this issue is left for future work to optimize the ILP program.

We also run the experiments in which the resource estimation is turned off. However, PRFloor fails to floorplan the design with only 6 PRRs because the PRM-PRR mapper over-assigns the PRMs to PRRs. From our empirical results, even though each PRR can host more PRMs in these experiments, the serviceability of these systems is almost identical to the system with the same amount of PRRs with resource estimation turned on. The PRM-PRR mapper only over-assigns one or two PRMs to some PRRs. This does not have much impact on the overall serviceability.

4.4.2 The Serviceability - Without Request Timeout. The serviceability values of the systems after running the simulation are shown in Figure6. The results are for the first set of tenant request (timeout is not allowed). It can be easily drawn from the figure that increasing the number of PRRs does improve the serviceability

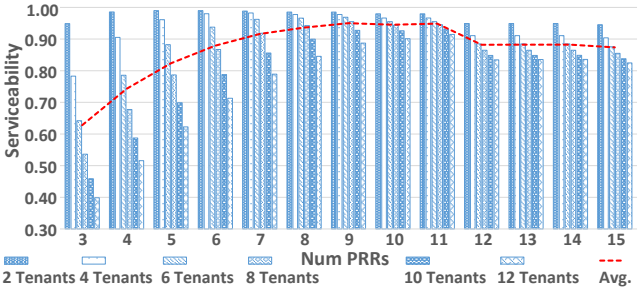


Figure 6: The serviceability of all systems with the different number of PRRs and tenants (no timeout).

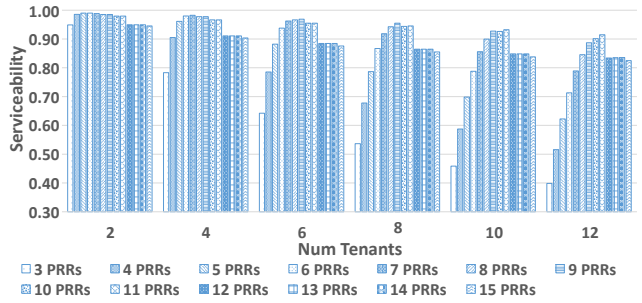


Figure 7: The serviceability of all systems shown in Figure6 but are grouped into the number of tenants.

of the systems significantly. For instance, comparing the baseline system in Figure6 with the 11-PRR system, the average serviceability increases from 0.74 to 0.95, which is 28% improvement. If we only consider the case of 12 tenants, the improvement is much higher, 75%. Additionally, having more tenants will decrease the serviceability of the baseline system drastically, about 11% for every increment of 2 tenants. With the 11-PRR system, it is just 1% .

Another interesting observation from Figure6 is the decrease in serviceability when we have more than 11 PRRs. This behavior is better observed in Figure7. In this graph, the systems are rearranged based on the number of tenants. The serviceabilities of the systems with more than 11 PRRs are even worse than the baseline when there are 2 and 4 tenants. It can be explained as follows. As the number of PRRs increases, the resources left for PRRs become smaller. Therefore, each PRR now hosts a fewer number of PRMs (Table 1). Some PRRs become hot-spots where multiple tenants try to request for the PRMs that can only be configured into those regions. As a result, the overall serviceability of these systems goes down. Figure7 also assists the architect on how to choose the best configuration based on the expected number of tenants. If there are less than 8 tenants, then the 9-PRR system delivers the best serviceability. If more tenants are needed to be served, then the 11-PRR system is the best all-around.

We also have an extended experiment to evaluate the chosen system under unknown conditions. It is to make sure that after the DSE, the system of interest can still maintain its quality of service as long as the distributions of the requested accelerators are the same. Figure8 illustrates the average serviceability offered by the same set of systems with three other different sets of requests from

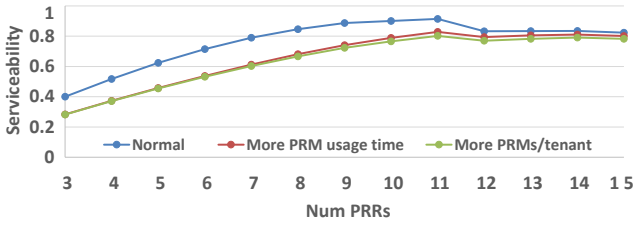


Figure 8: The serviceability of the same 11-PRR system with the other three different sets of requests from 12 tenants.

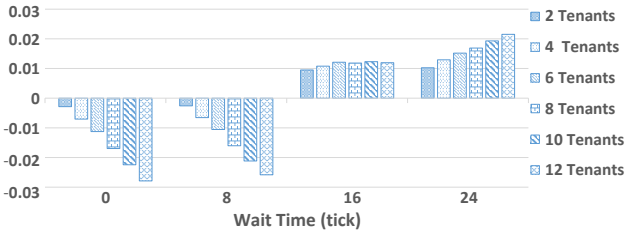


Figure 9: The serviceability differences when the reconfiguration overhead is considered versus the cases without that overhead. The requests are kept in the queue for at most "wait time" ticks.

12 tenants. The first one is created with the same configuration as stated in Section 4.1. In the second set, each tenant uses the PRMs for a longer duration, up to 800 ticks instead of 400. The third one allows the tenants to request up to 5 PRMs at any time instead of 3. Each request set is generated 5 times with different random seeds. The reported results are the average of these. As shown in the figure, the 11-PRR system still outperforms others across three new sets of requests.

4.4.3 The Serviceability - With Request Timeout. In PR systems, when a PRM is requested, there is a small reconfiguration overhead as presented in [2]. In their experiments, this overhead is about 4% of the execution of the PRM. It depends on how big the PRRs are and how long the PRMs are in use. However, the bigger PRMs tend to be used longer than the smaller ones. It also takes longer to reconfigure them. These assumption may not be entirely true in general cases; but it adequately reflects how big the latency is. Therefore, we run another set of experiments in which the reconfiguration overhead is 8 ticks, i.e. 4% of the average 200 ticks execution time of the PRMs. The requests from tenants are kept the same. We also allow the requests to stay longer in the queue for 0, 8, 16, or 24 ticks while waiting for their turn to be served. The ratio of the differences in the average serviceability of the systems is shown in Figure9. As expected, the serviceability decreases when the requests are only allowed to wait for a small amount of time. However, the change is very subtle, at most 2.7%. When the requests wait for a longer time, the systems indeed gain better serviceability. Thus, if the tenants do not strictly require their requests to be processed immediately, they will be able to have more requests served.

To assess our observation, we run the second set of tenant requests in which the larger timeout is enabled. In this time, we measure the *average wait time*, i.e, the time that each request has to wait to be served. The reconfiguration overhead is still 8 ticks. The

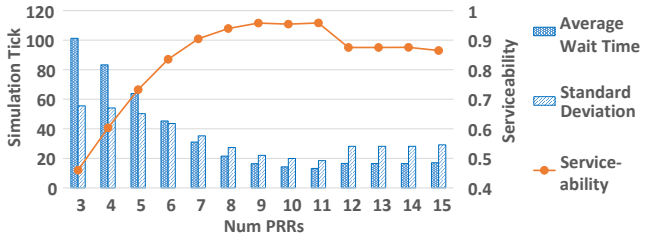


Figure 10: The average wait time for the requests and the respective serviceability of the systems with 12 tenants and a larger timeout.

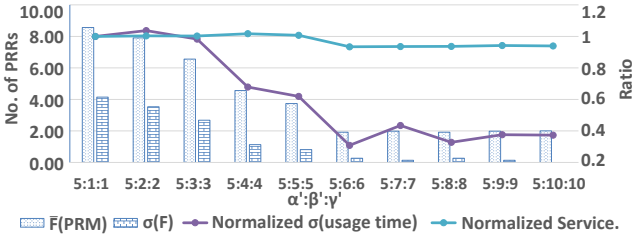


Figure 11: The effect of $\alpha\prime$, $\beta\prime$, $\gamma\prime$ on the average $F(PRM_i)$, its standard deviation, the standard deviation of the usage time and the serviceability of the 11-PRR system with 12 tenants. The standard deviations are normalized to when $\alpha\prime : \beta\prime : \gamma\prime = 5 : 1 : 1$

average wait time, as well as the corresponding serviceability of the systems with 12 tenants, are illustrated in Figure10. The graph shows that the more PRRs we have, the better wait time that the systems can deliver. The gain in the serviceability of all systems ranges from 4.8% to 17.7%. The average gain is 10%. The standard deviation of the wait time in the systems with the larger number of PRRs is also small. It means that these systems maintain good wait times for most of the requests.

4.4.4 The effect of $\alpha\prime$, $\beta\prime$ and $\gamma\prime$. In this section, the effect of $\alpha\prime$, $\beta\prime$ and $\gamma\prime$ on $F(PRM)$ is analyzed. Figure11 shows the mean of $F(PRM_i)$, $\bar{F}(PRM)$, in various configurations of the weights. The standard deviation, $\sigma(F)$, of all $F(PRM_i)$ is presented. The standard deviations of the PRRs' usuage time and the respective serviceablity are also reported. They are normalized to the case of $\alpha\prime : \beta\prime : \gamma\prime = 5 : 1 : 1$. The number of PRRs is 11. All PRMs have the same priority. Each system obtained is then simulated with 12 tenants, using the same set of requests as the experiment presented in Figure10.

As expected, $\bar{F}(PRM)$ decreases when $\alpha\prime$ decreases (with respect to $\beta\prime$ and $\gamma\prime$). When PRM-PRR mapper tries to even out the $F(PRM_i)$ in favor of high $\beta\prime$ and $\gamma\prime$, it has to lower the $F(PRM)$ of some PRMs significantly to increase the $F(PRM)$ of larger PRMs. As a result, the standard deviation becomes smaller as shown in the figure. $\beta\prime$ and $\gamma\prime$ also have an impact on the serviceability. As mentioned earlier, we include the DEV_{PRM} calculation in the objective function to provide a means to increase the chance of finding a compatible PRR for some PRMs. These PRMs initially can only be mapped to a smaller number of PRRs even though all of them have the same priority. As seen, when we increase $\beta\prime$:$\gamma\prime$ to up to 4:4, the serviceability is improved slightly, around 1.5%. After this point, the serviceability is reduced upto 8%. In this case, on average, each PRM has only 2

Figure 12: The average ratio of the number of PRRs assigned to PRMs in each domain (with different priority) versus the total number of PRRs.

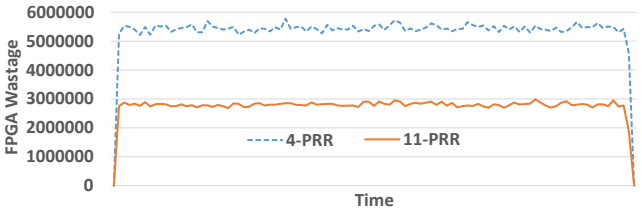


Figure 13: The weighted-sum of the wasted FPGA resource (CLB, BRAM, DSP) throughout the simulation.

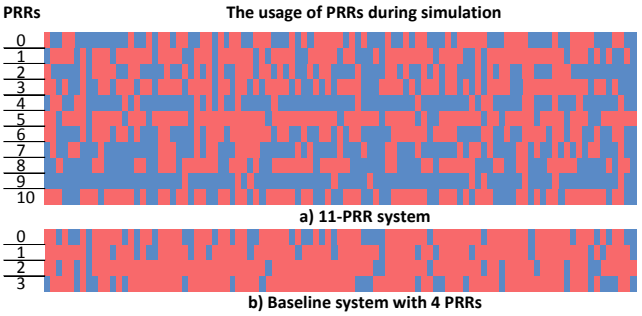


Figure 14: The utilization of each PRR throughout the simulation. For each PRR, the red/blue boxes mean that it is occupied/free.

compatible PRRs. This number is too small considering that there are 12 tenants sharing the FPGA and each tenant can request for up to 3 PRMs at a time. The DEV_{PRR} also helps reducing the hot-spot issue when too many PRMs are mapped to only a few number of PRRs. The downward trend of the normalized standard deviation of the PRRs' usage time illustrates that the PRRs are being utilized more fairly. The aforementioned effects of $\alpha\prime$, $\beta\prime$ and $\gamma\prime$) emphasize the ability of PRM-PRR maper in controlling the behavior of the system. The architect can tune the weights to suit the requirements.

4.4.5 Assigning priorities to PRMs. In our PRM-PRR mapper objective function, we consider the situation where some PRMs are used more often than the others. These common PRMs should be mapped to more PRRs. In this section, we present the effect of assigning a higher priority to some specific groups of PRMs. Our pool of PRMs is classified into application domains. These are *Arithmetic* (multiplication, addition, sine, division), *Cryptography* (SHA, MD5, AES, RC4), *Image/Video Processing* (JPEG decoder, image statistics, color filtering array, stream scaler), and many more. The experiment is set up such that, first, all PRMs under each of the three aforementioned domains will be assigned higher priority than the others. Then, the PRM-PRR mapper is executed for each case to find out the optimal mapping in the systems with 3 to 15 PRRs. Finally, the ratio of $F(PRM_i)$ over the available PRRs in each system are calculated and averaged based on the application domain across all systems. Figure12 presents the results.

It can be seen that the corresponding F_i is improved for each of the application domain (arithmetic, cryptography and image/video processing) when their priorities parameter are set higher. The F_i values of the other PRMs are automatically reduced to compensate for the PRMs under consideration. This flexibility gives the architect a freedom to adjust the mapping based on the statistics of the applications. Each system in the server farms can be tuned individually to offer even better performance for tenants with a suitable runtime mapping/scheduling strategy.

4.4.6 Wasted resources. We introduce the *wasted resource cost* metric to further inspect the usefulness of having a larger number of PRRs. If all PRRs must be large enough to accommodate the largest PRMs, which may not be used regularly, the valuable FPGA resources will be wasted. During the simulation, we compute the wasted resource cost by accumulating the difference in the resources of the requested PRMs with their allocated PRRs. Each type of resource – CLB, BRAM or DSP – is assigned different weight similar to [19]. The results are reported in Figure13. The state of the system is sampled 100 times at fixed intervals. As shown, the wasted resources in the 11-PRR system are almost half of the baseline. It implies that the 11-PRR system uses the FPGA resources much more efficiently than the baseline.

The number of PRRs and how they are being utilized during the simulation is captured in Figure14. In this experiment, $\alpha\prime : \beta\prime : \gamma\prime = 5 : 1 : 1$. The result suggests that utilizing the 11-PRR system could also improve the energy efficiency. In the figure, there are more blue *gaps* in the 11-PRR system compared to the baseline. It is possible to disable the PRRs during the free periods to save dynamic power. In the baseline system, all PRRs are active most of the time. But it can only serve half of the requests from tenants as discussed in Section 4.4.2.

5 CONCLUSION AND FUTURE WORKS

In this work, we propose a DSE process to find the best possible PR system configuration to optimize for the serviceability. The DSE process is composed of our novel ILP-based PRM-PRR mapper, the resource estimation method, request/simulator engines and the integration with an automatic floorplanner.

In the future, we will extend the ILP program and the DSE process to determine the best number of not only the PRRs, but also the FPGAs for the specific QoS requirements. The bitstream-relocation-aware approach will be explored to generate partial bitstreams more efficiently. The real-world cloud applications will also be examined.

Acknowledgments — This work is supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed) at the Technische Universität Dresden..

REFERENCES

[1] H. Artail, M. A. R. Saghir, M. Sharafeddin, H. Hajj, A. Kaitoua, R. Morcel, and H. Akkary. 2019. Speedy Cloud: Cloud Computing with Support for Hardware Acceleration Services. *IEEE Transactions on Cloud Computing* 7, 3 (July 2019), 850–865. https://doi.org/10.1109/TCC.2017.2665493

[2] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. 2016. Designing a virtual runtime for FPGA accelerators in the cloud. *IEEE 26th FPL* (2016).

[3] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. 2014. FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack. *IEEE 22nd FCCM* (2014).

[4] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the cloud. *ACM Computing Frontiers* (2014).

[5] S. Chen, J. Huang, X. Xu, B. Ding, and Q. Xu. 2018. Integrated Optimization of Partitioning, Scheduling, and Floorplanning for Partially Dynamically Reconfigurable Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[6] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2014. Architecture Support for Domain-Specific Accelerator-Rich CMPs. *ACM Transactions on Embedded Computing Systems* (2014).

[7] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio. 2015. A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6.

[8] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2016. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036* (2016).

[9] EPFL. 2017. Combinational Benchmark Suite. lsi.epfl.ch/benchmarks.

[10] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA accelerators for efficient cloud computing. *IEEE 7th CloudCom* (2015).

[11] Gurobi. 2017. Gurobi Optimization version 6.0.2. www.gurobi.com.

[12] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing* (2009).

[13] Z. István, G. Alonso, and A. Singla. 2018. Providing Multi-tenant Services with FPGAs: Case Study on a Key-Value Store. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 119–1195. https://doi.org/10.1109/FPL.2018.00029

[14] Oliver Knodel and Rainer G Spallek. 2015. Computing framework for dynamic integration of reconfigurable resources in a cloud. *DSD* (2015).

[15] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. 2008. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM TODAES* (2008).

[16] Sen Ma, Zeyad Aklah, and David Andrews. 2016. Just In Time Assembly of Accelerators. *ACM/SIGDA FPGA* (2016).

[17] Mathworks. 2017. Matlab 2016b Curve Fitting Toolbox ^{TM}. www.mathworks.com.

[18] Joel Mandebi Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, David Andrews, and Christophe Bobda. 2018. FPGAVirt : A Novel Virtualization Framework for FPGAs in the Cloud. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), 862–865. https://doi.org/10.1109/CLOUD.2018.00122

[19] Tuan D.A. Nguyen and Akash Kumar. 2016. PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems. *ACM/SIGDA FPGA* (2016).

[20] Tuan D A Nguyen and Akash Kumar. 2014. PR-HMPSoC: A versatile partially reconfigurable heterogeneous Multiprocessor System-on-Chip for dynamic FPGA-based embedded systems. *IEEE FPL* (2014).

[21] OpenCores. 2017. www.opencores.org.

[22] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *Computer Architecture, ACM/IEEE 41st International Symposium on* (2014).

[23] S.S. Sahoo, T.D.A. Nguyen, B. Veeravalli, and A. Kumar. 2019. Multi-objective design space exploration for system partitioning of FPGA-based Dynamic Partially Reconfigurable Systems. *Integration* 67 (2019), 95 – 107.

[24] Amazon Web Service. 2017. AWS FPGA Development Kit. github.com/aws/aws-fpga.git.

[25] Alan Sill. 2016. The Design and Architecture of Microservices. *IEEE Cloud Computing* (2016).

[26] Andy Singleton. 2016. The Economics of Microservices. *IEEE Cloud Computing* (2016).

[27] Naif Tarafdar, Thomas Lin, Daniel Ly-Ma, Daniel Rozhko, Alberto Leon-Garcia, and Paul Chow. 2019. *Building the Infrastructure for Deploying FPGAs in the Cloud*. Springer International Publishing, Cham, 9–33.

[28] Robert J Vanderbei. 2015. *Linear programming*. Springer.

[29] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. 2015. Enabling FPGAs in hyperscale data centers. *IEEE 12th UIC-ATC-ScalCom* (2015).

[30] T. Xia, J. C. Prevotet, and F. Nouvel. [n.d.]. Hypervisor mechanisms to manage FPGA reconfigurable accelerators. *FPT 2016* ([n. d.]).

[31] Xilinx. [n.d.]. Platform Studio (XPS). www.xilinx.com.

[32] Xilinx. 2013. Partial Reconfiguration - UG702. (2013).

[33] Q. Zhao, Hendarmawan, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi. 2017. hCODE 2.0: An open-source toolkit for building efficient FPGA-enabled clouds. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 267–270. https://doi.org/10.1109/FPT.2017.8280157

[34] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen. 2018. FPGA Resource Pooling in Cloud Computing. *IEEE Transactions on Cloud Computing* (2018), 1–1. https://doi.org/10.1109/TCC.2018.2874011