Performance Evaluation of Concurrently Executing Parallel Applications on Multi-Processor Systems

Ahsan Shabbir Akash Kumar Bart Mesman Henk Corporaal
[a.shabbir,a.kumar,b.mesman,h.corporaal]@tue.nl
Eindhoven University Of Technology, 5600MB Eindhoven, The Netherlands.

***Abstract*—Multi-processors are increasingly being used in modern embedded systems for reasons of power and speed. These systems have to support a large number of applications and standards, in different combinations, called use-cases. The key challenges are designing efficient systems handling all these use-cases; this requires fast exploration of software and hardware alternatives with accurate performance evaluation.**
In this paper, we present a system-level FPGA-based simulation methodology for performance evaluation of applications on multi-processor platforms. We observe that for multiple applications sharing an MPSoC platform, dynamic arbitration can cause deadlock in simulation. We use conservative Parallel Discrete Event Simulation (PDES) for simulation of these use-cases. We further note that conservative PDES is inefficient so we present a new PDES methodology that avoids causality errors by detecting them in advance. We call our new approach as smart conservative PDES. It is scalable in the number of use-cases and number of simulated processors and is 15% faster than conservative PDES. We further present results of a case-study of two real life applications. We used our simulation technique to do a design space exploration for optimal buffer space for JPEG and H263 decoders.

Keywords — PDES, Simulation, FPGAs, Performance Evaluation, DSE.

I. Introduction

Modern multimedia systems support a number of applications. For example, while traditionally a mobile phone had to support only a handful of applications like a voice call and sms, modern high-end mobile devices also act as a music and video player, camera, gps, mobile TV and a complete personal digital assistant. Many of these applications also need to meet other non-functional requirements like timing and low-power. These systems are increasingly becoming multiprocessor to support the strict power and performance requirements of applications. Accurate performance evaluation of these applications on multiprocessor systems is a key challenge when designing embedded systems [4]. To make matters worse, many applications are active simultaneously. Each combination of concurrently executing applications is defined as a *use-case* [2]. The number of these use-cases is exponential in the number of applications. An embedded system with 10 applications may have over a thousand use-cases. Due to a huge number of potential use-cases, it becomes a challenge to evaluate their performance.

Software simulation is often used for performance evaluation. Unfortunately, the accuracy of simulation is often inversely proportional to the time spent on it. Further, existing techniques for performance evaluation are limited to single-application designs [15], [16]. Hardware acceleration is often used to speed simulation. However, it generally requires a high design-time effort to build a simulation model in hardware. Some techniques do exist that provide automated flows, but they only simulate the system from the perspective of architecture and not that of applications [12], [13], [17]. Therefore, an automated hardware simulation design synthesis approach is needed that can deal with the large number of applications and use-cases in modern multiprocessor systems.

In this article, we present an automated FPGA-based simulation methodology for performance evaluation of multiple applications executing concurrently on multi-processor systems. We specify applications using synchronous dataflow (SDF) graph model of computation, although can easily handle any other dataflow model. The architecture description is specified including the desired mapping of tasks to processors, and the arbiter type for each processor. The target multiprocessor platform is generated in hardware, where each processor is simulated using a Xilinx Microblaze processor. The properties of each task (timing,repetition rate) are preserved during software generation for each processor. The desired arbiter for each processor is also generated automatically.

Further, we observe that dynamic arbiters in a processor lead to deadlock in simulation. In order to prevent this, parallel discrete event simulation (PDES) principles are used [5]. Typically PDES is used to accelerate sequential program execution on parallel machines. In our approach, we use PDES for simulating multiple applications – each consisting of parallel tasks – executing on multiple processors. Most PDES approaches fall under one of the two categories – *conservative* and *optimistic*. We propose and use a *smart conservative* approach that is intelligent to figure out when the sequential program execution can be compromised for improved efficiency. We have developed a mechanism which on every simulation step checks whether proceeding the simulation on incomplete information can result in causality errors. If we figure out that following the time-stamp ordering is imperative then we switch to conservative PDES, otherwise we turn to *smart conservative* PDES. Based on our technique, we have developed a tool for performance evaluation of applications on multi-processor platform. The tool is available at [7] for on-line use.

Following are major contributions of this paper.

- Identification and solution of problems faced during simulation of multiple applications on a MPSoC platform.
- A new technique for PDES is presented that is 15% faster than conservative PDES.
- An automated tool flow for the performance evaluation of multiple applications executing concurrently on an FPGA based MPSoC.

We validate our smart conservative PDES technique by a case-

study to optimize the buffer-space needed in an MPSoC design given the performance constraints of the applications. The case study shows that our FPGA-based simulation methodology is at least 3 times faster than an optimized software simulation. The solution is also scalable with the number of processors, applications, and use-cases, as shown by our results. Further, we compare the effectiveness of the smart conservative PDES over the traditional conservative PDES.

The paper is organized as follows. Section 2 explains related work in this direction. In section 3, we perform a review on SDF graphs. Section 4 presents our simulation methodology. Section 5 is about our newly developed PDES technique. Section 6 discusses details about implementation of our methodology and presents results of case-study which we performed to validate our approach. Section 7 concludes the paper.

II. Related Work

FPGAs can be used as simulation platforms; the reason being their flexibility, relatively low cost, and reconfigurability. Today, FPGAs are fast and big enough to provide scalable alternative to software simulation. FPGA simulation efforts such as RPM [12] have produced a system level multiprocessor emulator. Designers can choose from a library of architectural components and evaluate their performance. FAST [13] is an FPGA based platform for modeling multi-processor system with MIPS cores. It is suitable for MPSoC memory system research. RAMP [17] is a cycle accurate, distributed concurrent event simulator. It claims to provide the user with the flexibility to configure all components of multiprocessor systems like processing elements, communication infrastructure, programming model etc. ATLAS [14] uses the BEE2 boards from RAMP and its main emphasis is software research for transactional memory. The prototype runs the GNU/linux operating system and runs multi-threaded applications that use transactional memory.

All the above mentioned platforms simulate architectural components of the multi-processor research. On the contrary, our platform performs performance evaluation of multiple applications on a multi-processor fabric. There are also some analysis tools like SDF3 [15] for single applications, and they use heuristics to find the actor-processor mappings. In [16], authors have presented a scheduler for SDF graph simulation on multi-processor platforms but it does not support multiple applications. As far as we know, our approach is the first to use FPGAs as simulation platform for performance evaluation of multiple applications running concurrently on MPSoC platforms.

In [4], the authors have presented a simulation model for multiple applications. The tool helps in finding the throughput of applications but it is not scalable and like any software simulation environment it gets slower and slower as we increase the number of simulated processors. Simflex [8] is a micro-architectural simulator for multi-processors. It accelerates simulation by exploiting homogeneity of application behaviors that repeat millions of times. It analyzes the application and chooses the size of sample in such a way to capture the behavior of the model completely. This technique has a very high overhead due to creation of complete state before execution of each sample. MAMPS [2] is a tool flow for mapping multi-media applications on FPGA. It evaluates the performance of applications by executing their models on the FPGA fabric, however we only forward the time-stamps of execution. Forwarding time stamps takes few cycles as compared to execution of the program model. This makes our approach faster than MAMPS.

A-ports [9] is another architectural simulator on FPGA. Like our approach, it is also a graph of connected nodes. A node may execute a simulation cycle when all of its inputs are ready. However there are some key differences. A-Ports nodes are required to send a message every cycle; even if the message indicates that no change in the state of node has happened. However, we only forward the control tokens if there is a chance of deadlock in simulation.

MPARM [10] is an environment for MPSoC design space exploration using SystemC. It is complete platform solution for MPSoC simulation composed of processor models (ARM) bus models (AMBA), memory models, hardware support for SMP (hardware semaphores), and a software development toolset including a C compiler and an operating system. It provides several performance statistics, such as cache miss/hit rate and bus contention and average waiting time. In [11] authors have presented a methodology for performance analysis of processor at different abstraction levels, offering different trade-offs between estimation speed and accuracy. Neural networks are used to provide software performance estimation. The neural networks are first trained on benchmarks of specific processors. The methodology is targeted towards processor selection.

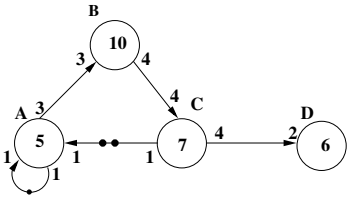


Fig. 1: Example of an SDF Graph.

III. Synchronous Data Flow Graphs

DSP and multi-media applications are of streaming nature. Data flows through the processing nodes. These nodes perform their computations on chunks of data and forward it to the next processing node. SDF Graphs [20] can model these applications efficiently. SDF graphs are used extensively, to find the expected performance of application on the computation platform.

Figure 1 shows an example of an SDF Graph. There are four actors/tasks in this graph (we will use terms actor and task interchangeably). As in a typical data flow graph, a directed edge represents the dependency between actors. Actors also need some input data (or control information) before they can start and usually also produce some output data; such information is referred to as tokens. The edges may also contain initial tokens, indicated by bullets on the edges, as seen on the edge from actor C to A in Figure 1. Required number of tokens at input edge, and number of tokens produced after execution (input and output token rates), are also visible in Figure 1. Actor execution is also called firing. An actor is

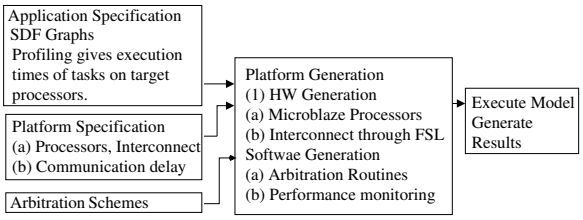


Fig. 2: Simulation methodology

called ready when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output edges; an actor can only fire when it is ready.

In the above example, only A can start execution from the initial state, since the required number of tokens is present on its only incoming edge. Once A has finished execution it will produce 3 tokens on the edge to B. B can then proceed as it has enough tokens and upon completion produce 4 tokens on the edge to C. Since there are two initial tokens on the edge from C to A, A can again fire as soon as it has finished the first execution, without waiting for C to execute.

We choose SDF graphs as they are very helpful in finding the throughput and buffer requirements of applications running on multi-processor platforms. This brings analyzability and predictability in the design which is of great importance in embedded and non-embedded domains. Our approach is also applicable to more general data flow models.

IV. Simulation Platform Generation

In an MPSoC architecture, where processors are already selected and hardware and software components have been defined, a system-level simulation model can be used for performance analysis. MPSoC architectures are composed of multiple processors, hardware IPs, memories and peripherals. Evaluation of individual components is not sufficient to analyze the system performance. The situation is further complicated if the platform has to support multiple applications simultaneously.

Our simulation platform is a network of nodes/processors connected to each other through First-in-First-out (FIFOs) buffers. These nodes continuously check their input ports and wait for arrival of tokens. The nodes then process these tokens and forward the results to succeeding nodes. FIFOs are placed between the nodes so that synchronization signals between the producing and consuming nodes are not required. We simulate a system of K applications having N_k actors. E.g., "$a_0, a_1, ..., a_{N_{a-1}}$" are "$N_a$" actors from application A and "$b_0, b_1, ..., b_{N_{b-1}}$" are "$N_b$" actors from application B. Actors from the same application communicate with each other exclusively through messages going through edges "c". We assume that the executions times of the actors on the target processors are known by profiling. Along with the application information, we also require the platform information; e.g, the number of processors, their interconnect topology, communication delay. At present we support point-to-point networks, but our methodology is also applicable to shared networks. Actors/Tasks executing on our platform are non-preemptive.

Figure 2 shows our simulation platform generation methodology. Applications are specified as SDF Graphs. Actor execution times, platform specification, and scheduling policy at each processor is input to our tool. The tool then generates platform hardware and software (for Xilinx FPGA). Software includes arbitration routines for schedulers in each processor. The model is then executed and results are processed.

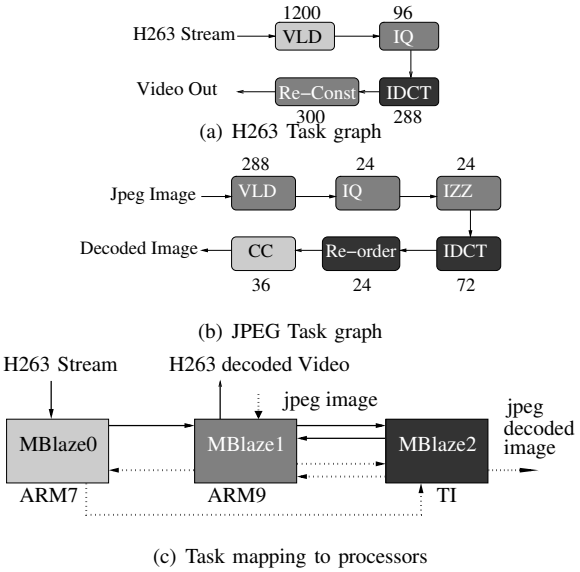


Fig. 3: Simulation platform generation

TABLE I: Processor-Actor Assignment.

Proc.	H263	JPEG	Sched. pol.
ARM7	VLD	CC	FCFS
ARM9	IQ,Re-const	VLD,IQ,IZZ	RRWS
TI	IDCT	IDCT,Re-order	FCFS

Figure 3 shows an example of our simulation technique. H263 and JPEG are two applications to be simulated on a three processor platform. Assume that ARM7, ARM9 and a DSP from TI constitute a multi-processor platform. The task graphs of applications are shown in Figure 3(a) and Figure 3(b). Table I shows a mapping of tasks from the two application on each processor and also shows the task scheduling policy at each processor. The execution times (in time units) of the graphs are also shown in Figure 3. The platform is simulated on a 3 microblaze system which has one-to-one correspondence with the target platform as shown in Figure 3(c). Each processor can have actors from different applications and maintains a local time-stamp "ts_{proc}" of its progress. In addition each actor also carries with it its own time-stamp "ts_{actor}". The time-stamps indicate how far in simulation time the processor or the actors have progressed. For a processor, the time-stamps are updated when it executes an actor or when it is idle. The same goes for actors. The time at which an actor is ready, is determined by the time-stamp of the latest available token from all input edges of an actor. Following are important definitions/rules used in our approach.

Definitions:

- ts_{actor} is earliest possible task/actor execution time at which an actor can execute.

- The **time-stamp** of a processor ts_{proc} is the latest time to which this processor has simulated. It is updated as follows

$$ts_{proc_{new}} = max(ts_{proc}, ts_{actor}) \quad (1)$$

- ts_{token} is the processor time $ts_{proc}(i)$ at which token is produced in $proc_i$.
- Before an actor can fire, it has to ensure that no other inputs can cause any change in the decision to execute the actor. This is also called **safe to execute**. When an actor executes, its execution time $T_{exec}(actor)$ is added to its actor time.

$$ts_{actor_{new}} = max(ts_{actor}, ts_{proc.}) + T_{exec}(actor) \quad (2)$$

- For processors having a single-actor, the processor can fire the actor when it is ready.
- Each FIFO buffer has a time-stamp "ts_{fifo}" associated with it. When empty, it is equal to ts_{token} of the last received token; otherwise it is equal to ts_{token} of the token at the head of the FIFO.

THEOREM 1. *Successive time-stamps $ts_{fifo_{(i)}}$, $i \in \mathbb{N}$, on a FIFO (edge) are guaranteed to be non-decreasing i.e. $ts_{fifo_{(i)}} \geq ts_{fifo_{(j)}}$ for $i > j$.*

Proof: Let p_0 and p_1 be two processors connected to each other through a FIFO edge. Let $ts_{fifo_{(1)}}$ and $ts_{fifo_{(2)}}$ be time-stamps of successive tokens sent from p_0 to p_1. We know that the token received in the FIFO is the processor time of previous processor, and according to Equation 1 the processor time can not decrease. Therefore $ts_{fifo_{(2)}} \geq ts_{fifo_{(1)}}$. Hence time-stamps on a FIFO edge are guaranteed to be non-decreasing. ■

Algorithm 1 Determining the ready time of all the actors.

1: {ts_{actor} is ready time-stamp of an actor a.}
2: **for all** Incoming edges c where $sink_c = a$ **do**
3: Read required input tokens on c
4: Let ts_c be the time-stamp of the last read token on c
5: ts_{actor} = max(ts_{actor}, ts_c)
6: **end for**

Algorithm 1 updates the time for each actor. The algorithm reads the input tokens from all the incoming edges of the actor. Since we are only simulating the performance, the real data is not important and we are only interested in the time-stamps of when the data was produced. The actor is ready to fire at the time the last token is available. Since we are sure the tokens in any FIFO are in non-decreasing time-stamps, we can simply check the time-stamp of the last token read on all the edges to determine the ready time of the actor.

The same is illustrated by means of an example in Figure 4. The actor has two incoming edges and the number of tokens needed from each edge is shown on the respective FIFO. The time-stamps of the last tokens read on the left and on the top edge are 10 and 9 respectively. Since the actor needs 3 time-units to execute, the tokens produced on the edge have the time-stamp of 13. Right side of Figure 4 shows the actor after its execution. This assumes that there is no contention on the processing node and that the actor can start execution as soon as it is ready.

For static scheduling methodologies like Round Robin (RR) and static order [4], the actor execution order is fixed and does not depend on the arrival of actor time-stamps. In such deterministic cases, there is no problem of ordering of actor execution in simulation model. For dynamic arbitration schemes like FCFS and Round Robin with Skipping (RRWS), the simulation model should behave the same way as the physical system. However, this can lead to deadlock in the system. To remove deadlocks, we have implemented the well know technique used in PDES. More details are in the next section.

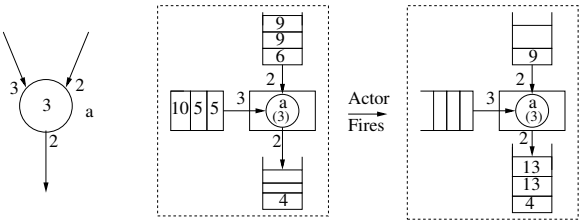


Fig. 4: **FIFO** contains time-stamps of the tokens and determine when an actor fires.

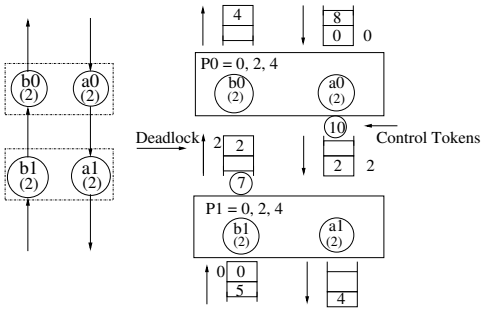


Fig. 5: System deadlock.

V. PDES For Multiple Applications

A. Deadlocks

An important issue in distributed simulation is deadlock. According to [19], a deadlock condition is defined as

1) not all the processes in a network of processes have terminated and
2) no process is executable.

In dynamic scheduling like FCFS, the ready actor with the smallest time-stamp is executed. However if one of the input FIFOs of this actor is empty; it blocks. If a cycle of interdependent empty FIFOs arises that has small time-stamps, then each actor in that cycle must block and the simulation deadlocks (more is described in following example). These deadlocks can be avoided by a mechanism of look ahead (in our case we call them "control tokens") described in [5]. Readers can refer to [5] for further details and proofs.

Figure 5 shows one such situation. In this example, both processors P0 and P1, have two actors each from different applications A and B. Their respective actor execution times (2 units each) are also shown in the figure. We assume that at the beginning of simulation, actor a0 and b1 execute due to initial tokens at their inputs queues. Processor times of both P0 and P1 are incremented to 2 as shown in Figure 5. Now assume actors a0 and b1 receive time-stamps 8 and 5 at their queues,

respectively. Note that tokens removed from the queues during previous execution are shown beside the queues. Actors a1 and b0, both have data in their input queues and their time-stamps are lower than that of a0 and b1 (2 each for a1 and b0 against 8 and 5 of a0 and b1, respectively). So both a1 and b0 fire resulting time stamps of 4 each, as shown in their output queues (Figure 5). Next, the time-stamps at input FIFOs of actors a1 and b0 are still lowest but they are waiting for their data. There is a cycle of dependency between all four actors and the simulation deadlocks and can not proceed any further. To rectify this problem, we forward the expected arrival time of tokens to the next processor. We call these tokens "control tokens". These control time-stamps require the system to be predictable, i.e. we are able to predict the output of an actor from the knowledge of its previous executions. In case of SDF graphs we can predict the finishing time of an actor execution by Equation(3).

$$t_{cntrl}(actor) = T_{exec}(actor) + max(ts_{proc}, ts_{actor}(actor)) \quad (3)$$

Here t_{cntrl} is the time-stamp of the control token to be sent to the next processor. This time-stamp is a message to the receiving processor that it will not receive any output from sending processor before time t_{cntrl}.
In case of above example, if actor b1 forwards control token (encircled in Figure 5) having time-stamp of 7 and actor a0 forwards control token of time-stamp 10 (according to Equation (3)), the deadlock resolves and simulation continues. To avoid sending too many control tokens, we only consider sending one when the local processor time passed the time of the previously sent control token.

B. Smart Conservative PDES

PDES, also called distributed simulation refers to the execution of a single discrete event simulation program on a parallel computer. All algorithms for parallel simulation either fall into the **conservative** or **optimistic** class.

- Parallel simulations are conservative if they satisfy the property that no process receives information from any other process that **predates** the current simulation time of the receiving process [6].
- They are optimistic if the processes can act on incomplete information thus admitting the case where messages may arrive **in the past**.

Optimistic methods exploit more parallelism; however they require some sort of roll back mechanism to an earlier valid state. To achieve this synchronization, each process must checkpoint its state and event information, which requires storage space [6]. In multi-application systems when dynamic scheduling is used, an optimistic approach generates a lot of correction traffic due to close dependence of time-stamps on the successive actors in an application. This increased traffic will affect performance monitoring of the application.
So we propose a new PDES approach which we name as **smart conservative** PDES. Our approach is different from conservative PDES as we develop a mechanism to find out the cases when violation of event list order can not produce causality errors. In these cases we proceed the simulation and do not wait un-necessarily for information which will not affect the simulation order. If our mechanism finds out that there is a possibility of causality error, we switch to conservative PDES. An example in the next subsection will further clarify our approach. Our approach borrows many concepts from PDES work by Chandy [5] as we are using event driven simulation. PDES simulates a single program onto a multi-processor platform. However, we use PDES to simulate multiple applications modeled as SDF Graphs. So not only the actors of an application are to be executed in a sequence, scheduling of actors from different applications should also follow some scheduling policy. Our methodology also does not require any central scheduler to synchronize the processing nodes, which makes our approach more scalable.

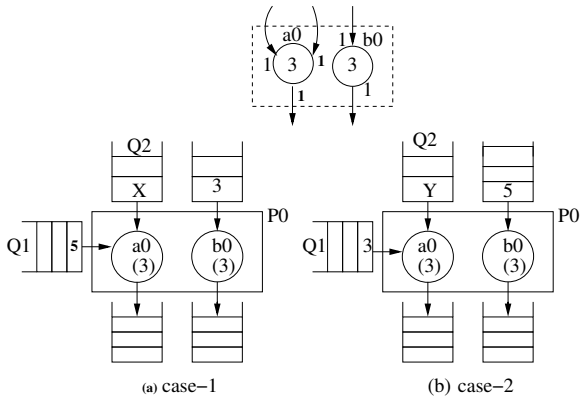


Fig. 6: Smart conservative PDES example

C. Motivating Example

In this subsection, we show that for multi-application simulation, our smart conservative PDES provides an efficient solution. Figure 6 shows two actors from two applications mapped onto one processor P0. Their corresponding SDF graph is shown on top of the figure. Actor a0 has two incoming edges and actor b0 has one. Rate at all edges is 1. Figure 6(a) shows the case when actor b0 has a lower time-stamp than actor a0 and second edge of actor a0 is empty (so the token X in queue Q2 has not yet arrived). For this case, if we schedule these actors with the FCFS, we can ignore the token at Q2 as actor time of a0 is $ts_{a0} = max(5, X)$ (according to algorithm 1). Even if the time-stamp of the received token at Q2 (which is X), is lower than 3 we will execute b0 (as for FCFS we are looking for the actor having minimum time-stamp value given by $min(max(5, X), 3)$). This property of not waiting for the arrival of token "X" at Q2 is **smart conservative** scheduling decision.
Figure 6(b) shows the case, when one of the queues of a0 has a lower time-stamp than that of b0 and the other queue of a0 is empty. Let "Y" be the expected time-stamp value of token at Q2. The value of Y can effect the scheduling decision by the relation $min(5, max(3, Y)$. It can be lower or higher than 5, in this case we use conservative approach and wait for the arrival of time-stamp at Q2.
Now consider we apply conservative PDES for both cases. Then in the first case, we are waiting unnecessarily for the token in Q2 of actor a0.

Algorithm 2 FCFS arbitration algorithm for PDES.

```
1: FCFS_arbitration()
2: {Check which of the actors are ready to execute.}
3: for all actors a do
4:     actor_ready[a] = is_actor_ready(a)
5: end for
6: for all actors a do
7:     if (ts_actor[a] < min_time) then
8:         min_time = ts_actor[a];actor = a;flag_min_time = 1;
9:     end if
10: end for
11: {If all the actors get ready after the CPU was idle then we set the CPU
    time equal to the minimum time of the actors.}
12: ts_proc = min_time;
13: if (actor_ready[actor] && flag_min_time == 1) then
14:     execute the actor
15: else
16:     send_control_tokens(ts_proc); return FCFS_arbitration()
17: end if
```

Algorithm 3 RRWS arbitration algorithm for PDES.

```
1: RRWS_arbitration(actor)
2: for all actors a do
3:     if (ts_actor[a] < minimum_time) then
4:         minimum_time = ts_actor[a]
5:     end if
6: end for
7: {If all the actors get ready after the CPU was idle then we set the CPU
   time equal to the minimum time of the actor}
8: ts_proc = min_time;
9: {Check the next actor from the list if it is ready}
10: if (is_actor_ready(actor)) then
11:     if (ts_proc < ts_actor[actor]) then
12:         Skip the actor;RRWS_arbitration(next_actor);
13:     else
14:         return; Execute the actor
15:     end if
16: else
17:     if (ts_proc >= ts_actor[actor]) then
18:         send_control_tokens(ts_proc);
19:         Wait for the actor to get ready;
20:         RRWS_arbitration(actor)
21:     else
22:         Skip the actor;return RRWS_arbitration(next_actor);
23:     end if
24: end if
```

D. Dynamic Actor Arbitration

Algorithm 2 shows our smart conservative FCFS arbitration. The algorithm waits for any actor to get ready for execution (line 4). Then it checks the minimum time of the actors. Next, the algorithm verifies that the processor time is not less than the minimum actor time. This can happen if all the actors get ready during the time the processor was idle. In this case we increase our processor time to minimum actor time (line 12). The actor with smallest time stamp is executed if it is ready. Control time-stamps are sent only if the actor is not ready or its time is not minimum. Algorithm 3 shows the pseudo code for RRWS arbitration under PDES. Like FCFS, the algorithm ensures that if the processor time is less than actor times of all actors then it sets the processor time to minimum actor time. Then it picks an actor from the list of actors and checks if it is ready (line 10). If the actor is ready then it determines if it is safe to execute this actor. E.g. if the processor time is less than the actor time, then we can not execute this actor, because there is a possibility that an actor from other application may arrive before the ready time of this actor. So we skip the actor (line 12). If this is not the case, we execute the actor and pick the next actor from the list. On the other hand if the actor is not ready yet and processor time is more than this actors' ready time, then we will have to wait for the arrival of tokens for this actor and we can not skip it, as shown on line 19 of Algorithm 3. Similarly, if the actor is not ready and its time is greater than the processor time then the tokens for this actor will arrive in future and we can safely skip to the next actor (line 22).

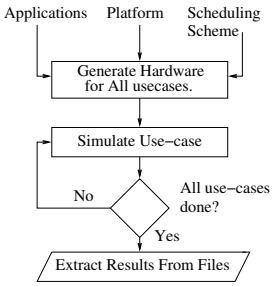


Fig. 7: Software for each use-case is loaded one by one

VI. FPGA implementation, Experiments and Results

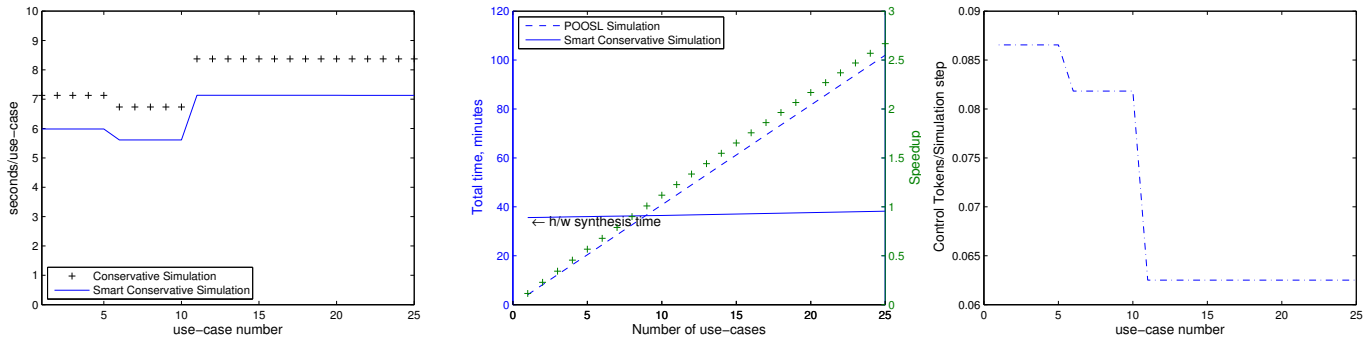
We have implemented our simulation tool onto Xilinx FPGA. The tasks are mapped to microblaze processors. The FIFO links are mapped to FAST Simplex Links (FSL). Additional peripherals such as timer, UART, and SysAce are also used in the design. UART is useful for printing debugging information of the system. Performance results of each use-case are stored in the SysAce Compact Flash card. A Timer is used for profiling the application.

The tool is fully automated and generates Xilinx HW/SW files from XML input files. Our implementation platform is Xilinx XUP Virtex II Pro Development Board with an xc2vp30 FPGA on-board. Xilinx EDK 8.2i and ISE 8.2i were used for synthesis and implementation. All tools run on a Pentium dual core at 2.0GHZ with 2.0GB of RAM.

A. DSE Case Study

We present a case-study to use our simulation methodology for performing a design space exploration to compute the optimal buffer requirement for two applications running concurrently on a multi-processor platform. Minimizing buffer size is an important objective when designing embedded systems. We explore the trade-off between the buffer-size used and throughput obtained for multiple applications. Increasing buffer space exploits more parallelism in the platform [21]. For single applications, the analysis is easier and has been presented earlier [21]. For multiple applications it is non-trivial to predict resource usage and performance because multiple applications cause interference when they compete for resources [2].

To predict performance of applications for different buffering options, time spent during hardware synthesis is the limiting factor. The solution is to synthesize a super-set hardware for all use-cases in a typical design space exploration problem,



(a) conservative vs smarter conservative (b) POOSL vs smart conservative (c) Number of control tokens/simulation step

Fig. 8: Comparison of smart conservative with POOSL simulation.

and only change the software for each point in the design space as shown in Figure 7. Interested readers are requested to read [2] for more details about the technique.

The case study is performed for JPEG and H263 applications. The descriptions are obtained from [22] and [23] respectively. Both applications are mapped onto a 3 processor platform. Figure 3 shows the task graphs and actor mappings of both applications. The actors are mapped on processors by equally distributing the compute load. In this case study the buffer size has been modeled by the initial tokens present on the incoming edge of the first actor. The larger this initial-token count, the larger the buffer needed to store the output data. In case of H263, each token corresponds to an entire decoded frame, while in the case of JPEG, it is a complete image. The design space consists of 25 points as for both applications, the number of initial tokens are varied from 0 to 4. Hardware synthesis time of the whole design on the FPGA was about 35 minutes. This is a one time overhead and after that we only compile the software for each design point and download that to get the results. We compare the performance of our tool with a POOSL model [24]. POOSL is a very expressive modeling language with a small set of powerfull primitives and completely formally defined semantics. It furthermore serves as a basis for performance analysis.

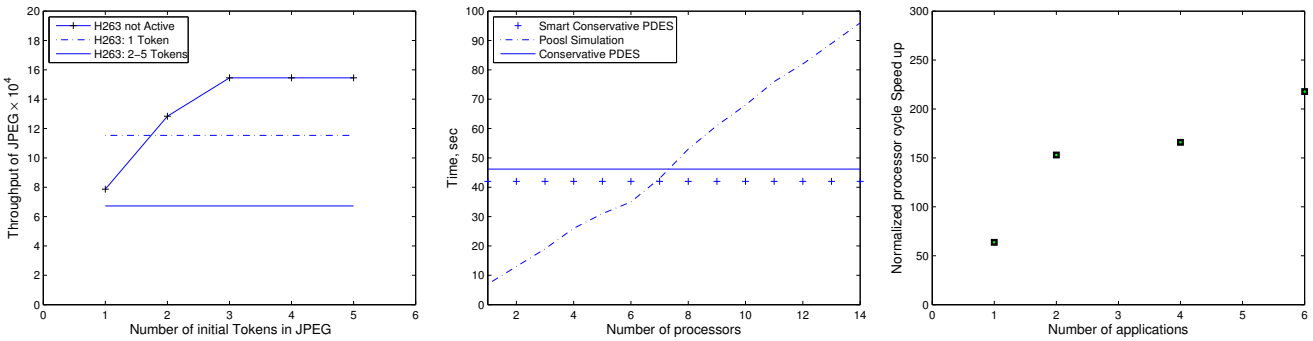
Figure 8 presents results from the case-study. Figure 8(a) shows time taken by smart conservative approach at each design point. Smart conservative approach is 15% faster than the conservative PDES. In [4], authors have created a tool to find application throughput and buffer requirement using POOSL. We compare our FPGA simulation platform results with this POOSL model. Total time taken for POOSL simulation is 95 minutes where as the FPGA simulation took 41 minutes only (including the hardware synthesis time). The speed up gained for FPGA simulation platform for two applications is around 3 as shown in Figure 8(b). This may seem like a small improvement, but since our approach is more scalable, speedup increases with increasing number of use-cases. E.g, if we have to perform same DSE for 4 applications, the design space will contain 625 use-cases. POOSL will simulate these use-cases in about 2500 minutes where as our FPGA based technique will require 110 minutes resulting in a speedup of around 23. Figure 8(c) shows number of control tokens per simulation step at each DSE point. Techniques like [9] require one control token at each simulation step where as in our case required number of control tokens/simulation step are very low. Figure 8(c) also shows a decrease in number of tokens with each DSE point. This is because from use-case 1-5, initial H263 tokens are zero so only one application is executing in the platform, resulting in more control tokens. As number of initial tokens of H263 are increasing from use-case 6-25, Both applications are active and fewer control tokens are required due to increased exploitable parallelism.

Figure 9(a) shows how the throughput of JPEG decoder varies with increasing number of tokens in the graph. When the number of tokens (i.e. buffer-size in the real application) is increased, the throughput also increases until a certain point after which it saturates. When the JPEG decoder is the only application executing (obtained by setting the initial tokens in H263 to zero), we observe that its throughput increases almost linearly till 3 initial tokens. We further observe that increasing the initial tokens of H263 worsens the performance of JPEG.

B. Scalability

Figure 9(b) shows the scalability of FPGA simulation as compared to POOSL. In this experiment we increased the number of processors in a single application by one processor at each experiment step. We simulated 500,000 iterations of each resulting graph. It is evident from figure 9(b) that for fewer number of processors, POOSL simulation is faster than FPGA. It should be noted that the FPGA platform is running at 50MHz and POOSL at 2.0 GHz. However, as we increase the number of processors, POOSL simulation gets slower and the simulation time keeps on increasing. On the other hand, FPGA simulation takes almost the same time, as we keep on increasing the number of processors. We increased the number of processors up to 14. This is the highest number of microblaze processors which we can synthesize on Xilinx xc2vp30 FPGA, used for our experiments.

Figure 9(c) shows the normalized processor cycle speed up of FPGA simulation against the POOSL simulation. POOSL and FPGA simulation are running at different frequencies using different type of processors, so to have a comparison we convert their simulation time to normalized processor cycles. Our POOSL simulation is running on Intel dual core 2.0 GHZ processor (only one of the processor being used) and our FPGA simulation is mapped on 6 microblaze processors



(a) Effect of varying initial Tokens on JPEG throughput (b) Scalability with #Proc. (c) Speedup with number of applications

Fig. 9: Figures comparing FPGA vs software simulation

running at 50 MHz each, as shown in Table II. Normalized processor cycles are obtained by first multiplying the simulation time, number of processors, and processor frequency for both FPGA and POOSL simulation and then dividing the corresponding products of POOSL by the FPGA products. Our normalized processor cycle speed up for 6 applications is as high as 218.

TABLE II: Normalized processor cycles.

# of Appl.	cycles POOSL $\times 10^9$	cycles FPGA $\times 10^9$	# of proc. POOSL	# of proc. FPGA	Tot. Cycles POOSL $\times 10^9$	Tot. Cycles FPGA $\times 10^9$	Speed up in FPGA
1	280	0.73	1	6	280	4.39	64
2	830	.90	1	6	830	5.45	153
4	1120	1.125	1	6	1120	6.75	165
6	1710	1.307	1	6	1710	7.84	218

VII. CONCLUSIONS

In this paper, we propose a novel technique to accelerate simulation of multiple applications on FPGAs. Our technique is scalable as larger FPGAs are available to simulate designs with a large number of applications and use-cases. The largest Vertex-4 device [25] from Xilinx can be configured to have about 97 microblaze processors. Support for such a large number of processors makes our approach very attractive. We also showed for two applications our simulation technique is at least 3 times faster than an efficient software solution. This speedup further increases for large number of applications, use-cases and processors. Another contribution of our paper is highlighting and solving the problems identified during the PDES simulation for multiple applications. We present efficient algorithms for three scheduling policies, RRWS, RR and FCFS, in combination with our new smart PDES simulation.

In the future, we intend to extend our work by supporting more scheduling techniques. We also plan to include hardware synthesis and performance evaluation options to our framework. This will allow us to evaluate performance of hardware modules like accelerators for multi-media applications. HDL descriptions will be integrated in our simulation platform to get feedback on performance of hardware modules.

REFERENCES

[1] W. Wolf, The future of multiprocessor systems-on-chips. Proc. DAC, pp 681-685, 2004.

[2] Akash Kumar *et al*. Multi-processor Systems Synthesis for Multiple Use-Cases of Multiple Applications on FPGA. In ACM TODEAS. pp 1-27,July, 2008.

[3] A. Jerraya and W. Wolf, eds., Multiprocessor Systems-on-Chips, Morgan Kaufman/Elsevier, 2004.

[4] Akash Kumar *et al*. Analyzing Composability of Applications on MP-SoC Platforms. In JSA Vol 54, 2008, pp 369-383.

[5] K. M. Chandy and J. Mishra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Trans. Soft. Engg., 1979.

[6] F. Richard. Parallel Discrete Event Simulation. Communications of ACM, 33(10), October 1990.

[7] www.ics.ele.tue.nl/~akash

[8] T. F. Wenisch *et al*. Simflex:Statistical Sampleing of Computer System Simulation. IEEE Micro, vol.26, issue 4. pp 18-31, July 2006.

[9] M. pellauer *et al*. A-Ports: an efficient abstraction for cycle-accurate performance modles on FPGAs. FPGA '08: Proceedings of 16th international ACM/SIGDA, New York USA.

[10] L. Binini *et al*. "MPARM": Exploring the Multi-processor SoC Design Space Exploration with SystemC" JVSP Vol. 41,2005.

[11] M.Oyamada *et al*. Software Performance Estimation in MPSoC design. DATE, 07. pp 38-43.

[12] K. Oner *et al*. The Design of RPM: An FPGA Based Multiprocessor Emulator. In *Proceedings of* Intl. Symp. on FPGA. pp 60-66, 1995.

[13] J. D. Davis *et al*. A Chip prototyping Substrate: The Flexible Architecture For Simulation ans Testing (FAST). HPCA, 2005.

[14] W. Sewook*et al*. A Practical FPGA-Based Framework for Novel CMP Research. Intl. Symp. on FPGA. pp 116-125, 2007.

[15] S.Stuijk. Predictable Mapping of Streaming Applications. Ph.D. Thesis, Eindhoven University of Technology, 2007.

[16] H. Chia-jui *et al*. Multithreaded Simulations for Synchronous Dataflow Graphs. DAC, pp 331-336, 2008.

[17] C. Chang *et al*. BEE2: A high-end Reconfigurable Computing System. IEEE Design and Test of Computers, pp 114-125, 2005.

[18] M. Flynn, Some Computer Organizations and their Effectiveness. IEEE Transactions on Computers. Vol. C-21, page 948, 1972.

[19] K.M. Chandy and J. Mishra. Deadlock Absence Proofs for Networks of Communicating Processes. IEEE Trans. Soft. Engg., 1981.

[20] E.A Lee and D. G. Messerschnitt. Static Scheduling of synchronous dataflow programs for digital signal processing. IEEE Transactions on Computers, Feb 1987.

[21] S. Stuijk et al., Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. 43rd DAC, pp. 899-904, 2006.

[22] D. Kock, E. 2002. Multiprocessor mapping of process networks: a JPEG decoding case study. In *Proc.* of 15th ISSS, pp 68-73.

[23] R. Hoes. Predictable Dynamic Behavior in NOC-based MPSoC. Available from: www.es.ele.tue.nl/epicurus/ ,2004

[24] Available from: http://www.es.ele.tue.nl/poosl.

[25] http://www.xilinx.com