Multiprocessor Systems Synthesis for Multiple Use-Cases of Multiple Applications on FPGA

AKASH KUMAR
National University of Singapore and Eindhoven University of Technology
SHAKITH FERNANDO and YAJUN HA
National University of Singapore
and
BART MESMAN and HENK CORPORAAL
Eindhoven University of Technology

Future applications for embedded systems demand chip multiprocessor designs to meet real-time deadlines. The large number of applications in these systems generates an exponential number of use-cases. The key design automation challenges are designing systems for these use-cases and fast exploration of software and hardware implementation alternatives with accurate performance evaluation of these use-cases. These challenges cannot be overcome by current design methodologies which are semiautomated, time consuming, and error prone.

In this article, we present a design methodology to generate multiprocessor systems in a systematic and fully automated way for *multiple use-cases*. Techniques are presented to merge multiple use-cases into one hardware design to minimize cost and design time, making it well suited for fast design-space exploration (DSE) in MPSoC systems. Heuristics to partition use-cases are also presented such that each partition can fit in an FPGA, and all use-cases can be catered for.

The proposed methodology is implemented into a tool for Xilinx FPGAs for evaluation. The tool is also made available online for the benefit of the research community and is used to carry out a DSE case study with multiple use-cases of real-life applications: H263 and JPEG decoders. The generation of the entire design takes about 100 ms, and the whole DSE was completed in 45 minutes, including FPGA mapping and synthesis. The heuristics used for use-case

Some results of this research were published in *Proceedings of the IEEE Conference on Field Programmable and Logic Applications (FPL)* 2007, pp. 92–97.
Authors' addresses: A. Kumar, Department of Electrical and Computer Engineering, National University of Singapore, 21 Lower Kent Ridge Road, Singapore 119077 and Department of Electrical Engineering, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven; email: a.kumar@tue.nl; S. Fernando, Y. Ha, Department of Electrical and Computer Engineering, National University of Singapore, 21 Lower Kent Ridge Road, Singapore 119077; B. Mesman, H. Corporaal, Department of Electrical Engineering, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven.

partitioning reduce the design-exploration time elevenfold in a case study with mobile-phone applications.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer Aided Engineering—*Computer-aided design (CAD)*

General Terms: Design, Algorithms, Performance, Experimentation

Additional Key Words and Phrases: FPGA, multiprocessor systems, design exploration, multi-application, multimedia systems, synchronous data-flow graphs, multiple use-cases

ACM Reference Format:

Kumar, A., Fernando, S., Ha, Y., Mesman, B., and Corporaal, H. 2008. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. ACM Trans. Des. Autom. Electron. Syst. 13, 3, Article 40 (July 2008), 27 pages, DOI = 10.1145/1367045.1367049 http://doi.acm.org/10.1145/1367045.1367049

1. INTRODUCTION

New applications for embedded systems demand chip-multiprocessor designs to meet real-time deadlines while achieving other critical design goals like low-power consumption and low cost. With high consumer demand, the time-to-market needs to be significantly reduced [Jerraya and Wolf 2004]. Multiprocessor systems-on-Chips (MPSoCs) have been proposed as a promising solution for all such problems. But one of the key design automation challenges that remain is that of allowing fast exploration of software and hardware implementation alternatives with accurate performance evaluation, also known as design-space exploration (DSE).

Further, the number of features that are supported in modern embedded systems (e.g., smart phones, PDAs, set-top boxes) is increasing faster than ever. To achieve high performance while keeping costs low, limited computational resources must be shared. The fact that these applications do not always run concurrently only adds a new dimension to the design problem. We define each such combination of applications that are active simultaneously as a *use-case* (also known as a *scenario* in literature [Paul et al. 2006]). For example, a mobile phone in one instant may be used to talk on the phone while surfing the Web and downloading some Java application in the background, and in another instant be used to listen to MP3 music while browsing JPEG pictures stored in the phone, and at the same time allowing a remote device to access the files in the phone over a bluetooth connection.

The number of such potential use-cases is exponential in the number of applications present in the system. The high demand of functionalities in such devices is leading to an increasing shift towards developing systems in software and programmable hardware in order to increase design flexibility. However, a single configuration of this programmable hardware may not be able to support this large number of use-cases with low cost and power. We envision that future complex embedded systems will be partitioned into several configurations and that the appropriate configuration will be loaded into the reconfigurable platform on-the-fly, as and when the use-cases are requested. This requires two major developments at the research front: (1) a systematic design methodology for allowing multiple use-cases to be merged on a single hardware configuration, and (2) a mechanism to keep the number of hardware configurations as small as possible. More hardware configurations imply higher cost since the configurations have to be stored somewhere in memory, and also lead to increased switching in the system.

Key Contributions. In this article, we present a solution to the aforementioned objectives. Following are the key contributions of the work.

—*MPSoC Design Flow.* We provide a systematic design methodology that generates multiprocessor systems for the desired use-cases.
—*Support for Multiple Use-Cases.* We give an algorithm for merging use-cases onto a single (FPGA) hardware configuration such that multiple use-cases may be supported in a single configuration, while minimizing hardware resources.
—*Partitioning Use-Cases.* When (FPGA) area constraints do not allow mapping of all use-cases on one configuration, we offer a methodology to partition use-cases such that the number of partitions (or configurations of FPGAs) is minimized.
—*Reducing Complexity.* Use-case partitioning is an instance of the *set-covering problem* [Cormen et al. 2001], which is known NP-hard. We propose efficient heuristics to solve this problem and compare their performance and complexity.
—*Area Estimation.* This is a technique that accurately predicts the resource requirements on the target FPGA without going through the entire synthesis process.
—*MPSoC Design Tool for FPGAs.* All of the aforesaid methods and algorithms are implemented such that the entire multiprocessor system can be generated for the given application and use-case descriptions in a *fully automated* way for Xilinx FPGAs. Besides the hardware, the required software for each processor is also generated. The tool is available at www.es.ele.tue.nl/mamps/.

The preceding contributions are essential to further research in the design automation community, since embedded devices are increasingly becoming multifeatured. Our flow allows designers to generate MPSoC designs quickly for multiple use-cases and to keep the number of hardware configurations to a minimum. Though the flow is aimed at minimizing the number of partitions, it also generates all partitions and allows the designer to study the performance of all use-cases in an automated way. The designer can then tailor the partitions to achieve better performance of all applications in a use-case.

Our flow is unique in a number of aspects: (1) It allows fast DSE by automating the design generation and exploration; (2) it supports multiple applications; (3) it supports multiple use-cases on one hardware platform; (4) estimates the area of design before the actual synthesis, allowing the designer to choose the right device; and (5) merges and partitions the use-cases to minimize the number of hardware configurations. To the best of our knowledge, there is no other existing flow to automatically map even multiple applications to an MPSoC platform, let alone multiple use-cases, except for our previous work [Kumar et al. 2007a]. The design space increases exponentially with an increasing number of use-cases, leading to increased time for DSE; our flow provides a quick solution to this.

While the flow is suitable for both design and evaluation, in this article we focus on the suitability of our flow for evaluating whether all the applications can meet their functional requirements in all the use-cases on FPGAs. We present a number of techniques to minimize the time spent in evaluation and design-space exploration of the system. We assume that applications are specified in the form of synchronous data-flow (SDF) graphs [Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2000]. SDF graphs are often used for modeling modern DSP applications and for designing concurrent multimedia applications. For the back-end design generation on the target FPGA, we use MAMPS (multiapplication and multiprocessor synthesis), as presented in our previous work [Kumar et al. 2007a].

A tool has been written to generate designs targeting Xilinx-based platform FPGAs. This tool is made available online for use by the research community at MAMPS [2007]. In addition to a Web site, an easy-to-use GUI tool is also available for both Windows and Linux. The tool is used to generate several designs with multiple use-cases that have been tested on the Xilinx University Virtex II Pro Board (XUPV2P) [Xilinx 2007]. However, the results obtained are equally valid on other FPGA architectures as well, and the tool can be easily extended to support other FPGA boards and architectures. We present a case study on how our methodology can be used for design-space exploration using JPEG and H263 decoders. We were able to explore 24 design points that tradeoff memory requirements and performance achieved with both applications running concurrently on an FPGA in a total of 45 minutes, including synthesis time. We also compare the execution time and performance of various heuristics that are used to merge and partition use-cases in a mobile-phone case study, and with randomly generated graphs.

The rest of the article is organized as follows. Section 2 reviews the related work for architecture generation and synthesis flows for multiprocessor systems. Section 3 introduces SDF graphs. Section 4 gives a short summary of the MAMPS flow. Section 5 describes our approach of merging use-cases in a single hardware description, while Section 6 explains how our partitioning approach splits the use-cases when not all can fit in one design. Section 7 describes the tool implementation, and Section 8 gives an overview of how resource utilization is estimated in this tool. Section 9 presents results of experiments done to evaluate our methodology. Section 10 concludes the article and gives directions for future work.

2. RELATED WORK

The problem of mapping an application to an architecture has been widely studied in literature. One of the recent works most related to our research is ESPAM [Nikolov et al. 2006]. This uses Kahn process networks (KPNs) [Kahn 1974] for application specification. In our approach, we use SDF [Lee and Messerschmitt 1987] for application specification instead. Further, our approach supports mapping of multiple applications, while ESPAM is limited to single applications. This difference is imperative for developing modern embedded systems which support more than tens of applications on a single MPSoC. The same difference can be seen between our approach and the one proposed in Jin et al. [2005], where an exploration framework to build efficient FPGA multiprocessors is proposed.

The Compaan/Laura design flow presented in Stefanov et al. [2004] also uses KPN specification for mapping applications to FPGAs. However, their approach is limited to a processor and coprocessor. Our approach aims at synthesizing complete MPSoC designs supporting multiple processors. Another approach for generating application-specific MPSoC architectures is presented in Lyonnard et al. [2001]. However, most of the steps in their approach are done manually. Exploring multiple design iterations is therefore not feasible. In our flow, the entire flow is automated, including the generation of the final bit-file that runs on the FPGA.

Yet another flow for generating MPSoCs for FPGAs has been presented in Kumar et al. [2007b]. However, that flow focuses on generic MPSoCs and not on application-specific architectures. Further, the work in Kumar et al. [2007b] uses networks-on-chip for communication fabric, while in our approach-dedicated links are used for communication to remove communication-resource contention altogether.

Xilinx provides a tool-chain as well to generate designs with multiple processors and peripherals [Xilinx 2007]. However, most of the features are limited to designs with a bus-based processor-coprocessor pair with shared memory. It is very time consuming and error prone to generate an MPSoC architecture and the corresponding software projects to run on the system. In our flow, an MPSoC architecture is automatically generated together with the respective software projects for each core.

The multiple use-case concept is relatively new to MPSoCs and one related research is presented in Murali et al. [2006]. However, that work focuses on supporting multiple use-cases for the communication infrastructure, in particular networks-on-chip. Our flow is mainly targeted towards supporting multiple use-cases from a computation perspective. In addition, we generate dedicated point-to-point connections for all the use-cases that are to be supported.

Our definition of a use-case is similar to what is defined as a *scenario* in Paul et al. [2006]. The authors in Paul et al. [2006] motivate the use of a scenario-oriented (or use-case-oriented, in our work) design flow for heterogeneous MPSoC platforms. Our approach provides one such design flow where designers can study the performance of all use-cases in an automated way and tune the architecture to achieve better performance of all applications in a use-case. The biggest advantage of our approach is that we provide a real synthesized MPSoC platform for designers to play with and measure performance. Further, for Paul et al. [2006], architecture is provided as an input and is static, whereas we generate platforms, given the application and use-case descriptions and provide the means to change (reconfigure) the architecture dynamically for different use-cases.

Table I. Comparison of Various Approaches for Providing Performance Estimates

	SDF^3 [Stuijk et al. 2006b]	POOSL [Kumar et al. 2006b]	ESPAM [Nikolov et al. 2006]	Our Flow
Approach Used	Analysis	Simulation	FPGA	FPGA
Model Used	SDF	SDF	KPN	SDF
Single Appl	Yes	Yes	Yes	Yes
Multiple Appl	No	Yes	No	Yes
Multiple Use-cases	No	No	No	Yes
Speed	Fastest	Slow	Fast	Fast
Accuracy	Less	High	Highest	Highest
Dedicated FIFO	N. A.	No	No	Yes
Arbiter Support	N. A.	Yes	N. A.	Yes

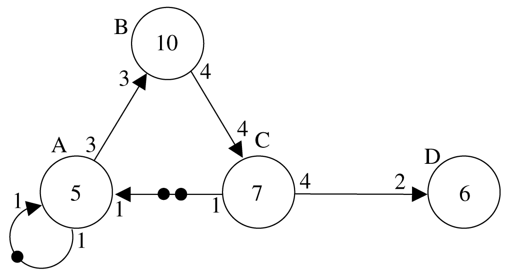


Fig. 1. Example of an SDF graph.

Table I shows various design approaches that provide estimates of application performance by various means. The first method uses SDF models and computes the throughput of the application by analyzing the application graph. However, it is only able to predict the performance of single applications. The simulation approach presented in Kumar et al. [2006b] uses POOSL [Theelen et al. 2007] for providing application performance estimates. This is more accurate than analysis, since more details can be modeled and their effects are measured using simulations. ESPAM is closest to our approach, as the authors also use FPGAs, but they do not support multiple applications. Our flow supports multiple applications and provides quick results. Further, ours is the only approach that supports multiple use-cases.

3. SYNCHRONOUS DATA-FLOW GRAPHS

Synchronous data-flow graphs (SDFGs; see Lee and Messerschmitt [1987]) are often used for modeling modern DSP applications [Sriram and Bhattacharyya 2000] and for designing concurrent multimedia applications implemented on multiprocessor platforms. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. Tasks are modeled by the vertices of an SDFG, which are called actors. SDFGs allow analysis of a system in terms of throughput and other performance properties, such as latency and buffer requirements [Stuijk et al. 2006a].

Figure 1 shows an example of an SDF graph. There are four actors in this graph. As in a typical data-flow graph, a directed edge represents the dependency between tasks. *Tasks* also need some input data (or control information) before they can start and usually also produce some output data; such terms of information are referred to as *tokens*. Actor execution is also called *firing*. An actor is called *ready* when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready.

The edges may also contain *initial tokens*, indicated by bullets on the edges, as seen on the edge from actor C to actor A in Figure 1. Buffer sizes may be modeled as comprising a back-edge with initial tokens. In such cases, the number of tokens on this edge indicates the buffer size available. When an actor writes data to such channels, the available size reduces; when the receiving actor consumes this data, the available buffer increases, modeled by an increase in the number of tokens.

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

4. MULTIAPPLICATION-FLOW OVERVIEW

Figure 2 shows an overview of the MAMPS flow that has been published in our previous work [Kumar et al. 2007a]. This flow generates multiprocessor systems from specification of multiple applications. Applications are assumed described in form of SDF graphs in xml format. A snippet of application specification of Appl0 is shown in Figure 3, corresponding to the application in Figure 2. The specification file contains details about how many actors are present in the application and how they are connected to the other actors. The execution time of the actors and their memory usage on the processing core are also specified. For each channel present in the graph, the file describes whether there are any initial tokens present on it. The buffer capacity of a particular channel is specified as well.

From these application descriptions, a multiprocessor system is generated. For a single application, each actor is mapped on a separate processor node, while for multiple applications, nodes are shared among actors of different applications. The total number of processors in the final architecture corresponds to the maximum number of actors in any application. For example, in Figure 2, a total of four processors are used in the design. This may not be the optimal way to allocate actors to processors and many optimizations remain to be performed.[1] The scope of this article, however, is to demonstrate how designs can be easily generated even for multiple use-cases and optimized for them.

For processors that have multiple actors mapped onto them, an arbitration scheme is generated. The arbitration scheme can be chosen by the designer

[1]The simplest choice, for example, would be to allow processor sharing within actors of the same application. These alternatives will be presented in later research.

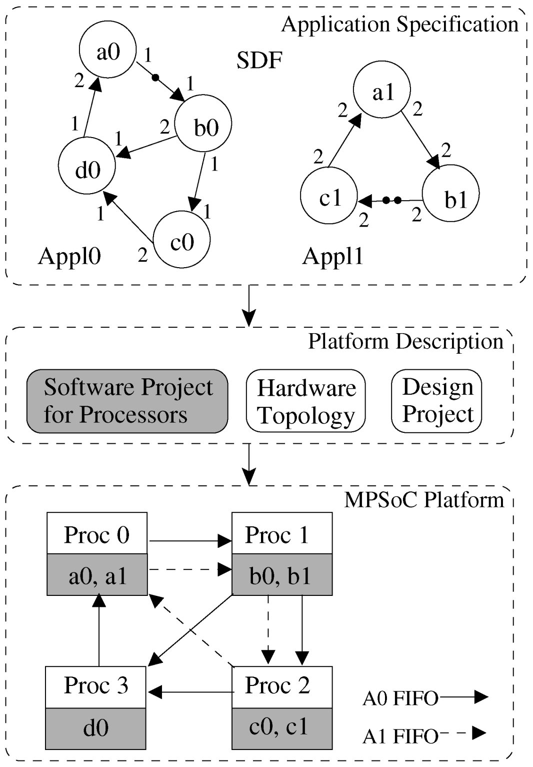


Fig. 2. Design flow.

```
<application id="Appl0">
  <actor name="a0">
    <port name="d0" type="in" rate="2"/>
    <port name="b0" type="out" rate="1"/>
    <executionTime time="1200"/>
  </actor>
  <actor name="b0">
    <port name="a0" type="in" rate="1"/>
    <port name="c0" type="out" rate="1"/>
    <port name="d0" type="out" rate="2"/>
    <executionTime time="9600"/>
  </actor>
```

Fig. 3. Snippet of Appl0 application specification.

as either round-robin or round-robin with skipping [Kumar et al. 2006a]. For the former, blocking reads and writes are used, while for the latter they are nonblocking. This allows actors to be skipped over if they are not ready, even if it is their turn in the list. However, it should be mentioned that strict round-robin often may lead to a deadlock in the system if the rates of applications are not properly assigned.

All the edges in an application are mapped on a unique FIFO channel. This creates an architecture that mimics the applications directly. Unlike processor sharing for multiple applications, the FIFO links are dedicated, as can be seen in Figure 2. As opposed to a network or bus-based infrastructure, the dedicated links remove the possible sources of contention that can limit performance. Since we have multiple applications running concurrently, there is often more than one link between some processors. Even in such cases, multiple FIFO channels are created. This avoids head-of-line blocking that can occur if one FIFO is shared for multiple channels [HOL 2007].[2]

In addition to the hardware topology, the software for each processor is also generated. The software includes the SDF model of the actor execution and the arbitration. If the source-code of an actor is available it may also be inserted in the description. Other miscellaneous files that are necessary for synthesis are also generated. An example of this in the case of FPGAs is the pin-constraints file.

5. SUPPORTING MULTIPLE USE-CASES

In this section, we describe how multiple use-cases can be merged into one design to save precious synthesis time and minimize hardware cost. When multiple use-cases are to be catered for during performance evaluation, time spent on hardware synthesis forms a bottleneck and limits the number of designs that can be explored in a typical design-space exploration. When designing systems this is even more important, as it often reduces the hardware resources needed in the final platform.

We start by defining a use-case.

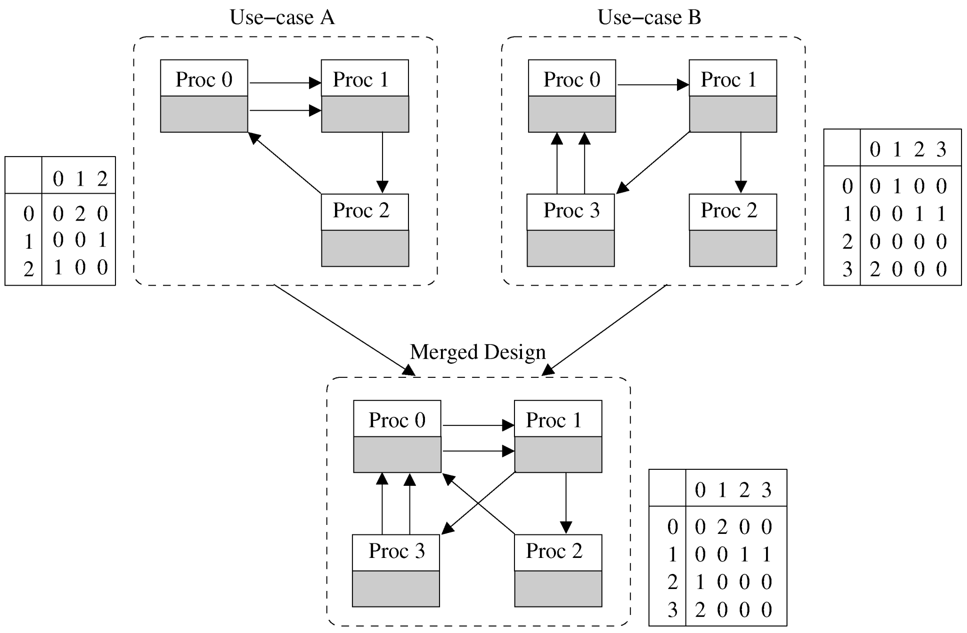
Definition 1 (*Use-Case*). Given a set of n applications $A_0, A_1, \dots A_{n-1}$, a use-case U is defined as a vector of n elements $(x_0, x_1, \dots x_{n-1})$ where $x_i \in \{0, 1\}$ $\forall i = 0, 1, \dots n-1$, such that $x_i = 1$ implies that application A_i is active.

In other words, a use-case represents a collection of multiple applications that are active simultaneously. Each application in the system requires hardware to be generated for simulation. Therefore, each use-case in turn has a certain hardware topology to be generated, as explained in Section 4. In addition, software is generated for each hardware processor in the design that models the set of actors mapped on it. The following two subsections provide details of how the hardware and software are generated.

5.1 Generating Hardware for Multiple Use-Cases

With different use-cases, since the hardware design is usually different, the entire new design has to be synthesized. Here we describe how we can merge the hardware required for different use-cases. Figure 4 shows an example of

[2]Sharing of links and memory may be possible but is left as a future work, since our ideas of use-cases merging are equally applicable with this approach as with any other communication fabric, such as networks-on-chip.



	0	1	2
0	0	2	0
1	0	0	1
2	1	0	0

	0	1	2	3
0	0	1	0	0
1	0	0	1	1
2	0	0	0	0
3	2	0	0	0

	0	1	2	3
0	0	2	0	0
1	0	0	1	1
2	1	0	0	0
3	2	0	0	0

Fig. 4. An example showing how the combined hardware for different use-cases is computed. The corresponding communication matrix is also shown for each hardware design.

two use-cases that are merged. The figure shows two use-cases A and B, with different hardware requirements that are merged to generate the design with *minimal* hardware requirements to support both. The combined hardware design is a superset of all the required resources such that all use-cases can be supported. The key motivation for the idea comes from the fact that while multiple applications are active concurrently in a given use-case, different use-cases are active exclusively.

The complete algorithm to obtain the minimal hardware to support all use-cases is described in Figure 5. The algorithm iterates over all use-cases to compute their individual resource requirements. This is, in turn, computed by using the estimates from the application requirements. While the number of processors needed is updated with a *max* operation (line 8 in Figure 5), the number of FIFOs is added for each application (indicated by line 10 in Figure 5). The total FIFO requirement of each application is computed by iterating over all the channels and adding a unique edge in the communication matrix for them. The communication matrix for the respective use-cases is also shown in Figure 4.

To compute minimal hardware requirements for the overall hardware for all the use-cases, both the number of processors and the number of FIFO channels are updated by a *max* operation (lines 13 and 15, respectively, in Figure 5). This is important because so doing generates only as many FIFO channels between any two processors as maximally needed during any particular use-case; thus, the generated hardware stays minimal. Therefore, in Figure 4, while there are in total three FIFO channels between Proc 0 and Proc 1, only two are used

```
Procedure: GenerateCommunicationMatrix
1: // Let $X_{ij}$ denote the number of FIFO channels needed from processor $P_i$ to $P_j$ overall
2: $X_{ij} = 0$                          // Initialize the communication matrix to 0
3: $N_{proc} = 0$                        // Initialize the number of processors to 0
4: for all Use-cases $U_k$ do
5:     $Y_{ij} = 0$                      // $Y_{ij}$ stores the number of FIFO channels needed for $U_k$
6:     $N_{proc,UseCase} = 0$            // Initialize processor count for use-case to 0
7:     for all Applications $A_l$ do
8:         $N_{proc,UseCase} = max(N_{proc,UseCase}, N_{proc,A_l})$     // Update processor count for $U_k$
9:         for all Channels $c$ in $A_l$ do
10:            $Y_{c_{src}c_{dest}} = Y_{c_{src}c_{dest}} + 1;$          // Increment FIFO channel count
11:        end for
12:    end for
13:    $N_{proc} = max(N_{proc}, N_{proc,UseCase})$                     // Update overall processor count
14:    for all $i$ and $j$ do
15:        $X_{ij} = max(X_{ij}, Y_{ij})$
16:    end for
17: end for
18: // $N_{proc}$ is now the total number of processors needed
19: // $X_{ij}$ is now the total number of FIFO channels needed
```

Fig. 5. Algorithm for determining minimal hardware design that supports multiple use-cases.

(at most) at the same time. Therefore, in the final design only two channels are produced between them.

5.2 Generating Software for Multiple Use-Cases

Software compilation is a lot faster as compared to hardware synthesis, in the MAMPS approach. The flow is similar to the one for generating software for single use-cases. However, we need to ensure that the numbering for FIFO channels is correct. This is very important in order to ensure that the system does not go into deadlock. For example, in Figure 4, Proc 0 in the merged hardware design has three incoming links. If we simply assign the link-ids by looking at the individual use-case, in Use-case B in the figure, the first link-id will be assigned to the channel from Proc 3. This will block the system, since the actor on Proc 3 will keep waiting for data from a link which never receives anything.

To avoid the preceding situation, the communication matrix is first constructed even when only the software needs to be generated. Link-ids are then computed by checking the number of links before the element in the communication matrix. For the output link-id, the numbers in the row are added, while for incoming links, the column is summed up. In Figure 4, for example, the communication matrix of Use-case B suggests that the incoming links to Proc 0 are only from Proc 3, but the actual hardware design synthesized has one extra link from Proc 2. The incoming link-id should therefore take this into account in this software.

5.3 Combining the Two Flows

Figure 6 shows how the hardware- and software flows come together to get quickly results for multiple use-cases. The input to the whole flow is the

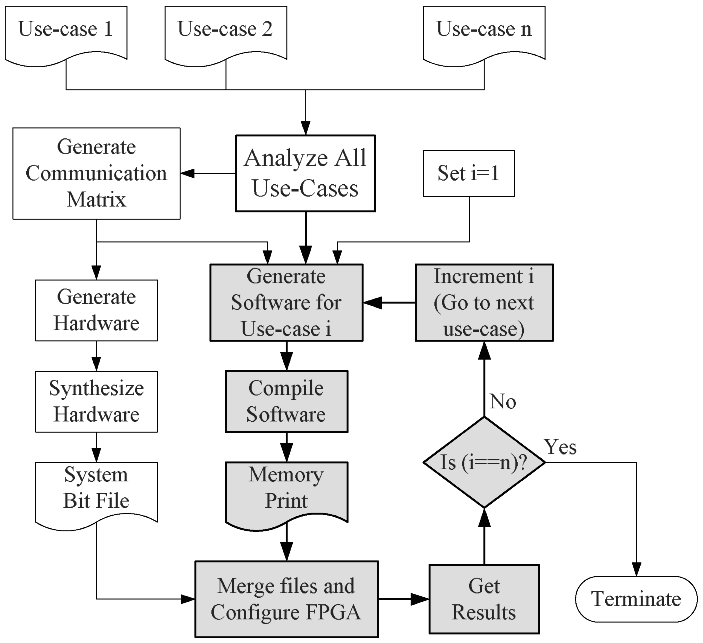


Fig. 6. The overall flow for analyzing multiple use-cases. Notice how the hardware flow executes only once, while the software flow is repeated for all the use-cases.

description of all the use-cases. From these descriptions, the communication matrix is constructed. This is used to generate the entire hardware. The same matrix is also used when software has to be generated for individual use-cases. The boxes shown in gray are repeated for each use-case. The flow terminates when all use-cases are explored. The results of each use-case are fed to the computer via the serial port and are also written out onto the compact flash card, as explained in Section 7. As can be seen in the figure, the hardware part is executed only once, whereas the software part is iterated until results for all the use-cases are obtained. This flow makes execution of multiple use-cases a lot faster, since hardware synthesis is no longer a bottleneck in system design and exploration.

The use-case analysis (see “Analyze All Use-Cases” in Figure 6) is done to find the maximum number of use-cases that can fit in one hardware design. (This is done by our ideas of area estimation that are explained in Section 8.) Formally, given a set S of m use-cases $S = \{U_0, U_1, \ldots U_{m-1}\}$, we wish to determine the biggest possible subset of S that is *feasible*, where feasibility implies that all the elements of the set can be merged into one hardware design that can fit in the given FPGA device. The next section explains what happens when not all use-cases can be merged in one hardware design.

6. USE-CASE PARTITIONING

Resources are always a constraint in an FPGA device. As the number of use-cases to be supported increases, the minimal hardware design increases as well,

Applications	Use-cases	Feasible-sets	Potential Partitions
A_0: H263 Enc	$U_0 = \{A_0, A_1, A_4\}$	$f_0 = \{U_0, U_1\}$	$\{f_0, f_1, f_4, f_5\}$
A_1: H263 Dec	$U_1 = \{A_0, A_1, A_5\}$	$f_1 = \{U_0, U_2\}$	$\{f_0, f_3, f_4\}$
A_2: JPEG Dec	$U_2 = \{A_2, A_3, A_4\}$	$f_2 = \{U_1, U_2\}$	$\{f_1, f_2, f_3\}$
A_3: MP3 Dec	$U_3 = \{A_0, A_4\}$	$f_3 = \{U_1, U_3, U_4\}$	$\{f_1, f_3\}$
A_4: Modem	$U_4 = \{A_2, A_4, A_5\}$	$f_4 = \{U_2, U_3\}$	
A_5: Voice Call		$f_5 = \{U_2, U_4\}$	

Fig. 7. Putting applications, use-cases, and feasible partitions in perspective.

and it often becomes difficult to fit all use-cases in a single hardware design. Here we propose a methodology to divide the use-cases in such a way that all can be tested, assuming that all use-cases can at least fit in the hardware resources when they are mapped in isolation.[3] Further, we wish to have as few a number of such hardware partitions as possible-since each extra partition implies extra hardware synthesis time. This is an NP-hard problem, as described next.

Problem 1. We are given $S = \{U_0, U_1, \ldots U_{m-1}\}$, where each use-case U_i is feasible in itself. Further, let us define set $\mathcal{F}$ of all feasible subsets of S. *Use-case partitioning* is finding the minimum subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of S.

Solution 1. This is clearly an instance of the set-covering problem, where the universe is depicted by S, and the subsets are denoted by $\mathcal{F}$. The set $\mathcal{C}$ we are looking for is the solution of the minimum set-covering problem, and corresponds to that of the use-case partitioning problem. Each set in $\mathcal{C}$ corresponds to a feasible hardware partition. The set-covering problem is known NP-hard [Garey and Johnson 1979; Cormen et al. 2001]. Use-case partitioning is therefore also an NP-hard problem.

The cost in both verification and design synthesis is directly proportional to the number of sets in $\mathcal{C}$. During verification, it is the time spent in synthesis which increases with partition count, while for system design more partitions imply a higher hardware cost. Since this is an NP-hard problem, in our tool we have used an approximation algorithm to solve it, called the greedy algorithm. The largest feasible subset of use-cases is first selected and a hardware partition created for it. This is repeated with the remaining use-cases until all use-cases are covered. As mentioned in Cormen et al. [2001], the maximum approximation error in using this technique over the minimal cover is $ln|X| + 1$, where X is the number of elements in the largest feasible set.

Figure 7 helps in better understanding the partitioning problem and provides a good perspective of the hierarchy of sets. The first box in the figure shows the applications that need to run on the platform. These are some of the applications that run on a mobile phone. The next box shows some typical use-cases; for example, U_1 represents a video call that requires video encoding, video decoding, and regular voice-call (as mentioned earlier, a use-case is a set of applications

[3]If an individual use-case does not fit, a bigger FPGA device is needed.

that run concurrently). The next box shows the family of sets $\mathcal{F}$, each of which is feasible. For simplicity only a part of $\mathcal{F}$ is shown in the figure. Clearly, the subsets of elements of $\mathcal{F}$ are also feasible; for example, when f_3 is feasible, so is $\{U_3, U_4\}$. As often the case, no feasible set exists which contains all the use-cases. Therefore, a subset $\mathcal{C} \subseteq \mathcal{F}$ need be chosen such that all the use-cases are covered in this subset. Few such possible subsets are shown in the last box. The last option is preferred over the rest, since it provides only two partitions.

6.1 Hitting the Complexity Wall

In order to be able to implement the greedy algorithm, we still need to be able to determine the largest feasible set. This poses a big problem in terms of implementation. The total number of possible sets grows exponentially with the number of use-cases. Suppose we have 8 applications in the system and that every combination thereof is possible, we have 255 use-cases overall. Since each use-case can either be in the set or not, we obtain a total of 2^{255} sets. Each set has then to be examined as to whether it is feasible; this takes linear time in size of set, which can in the worst case be the number of applications in the system. Thus, a system with N applications and their M possible use-cases has a complexity of $O(N.2^M)$ to find the largest feasible set. In the worst case the number of use-cases is also exponential, namely, $M = 2^N$. We see how the design space becomes infeasible to explore. In Section 9 we see some results of actual execution times.

6.2 Reducing the Execution Time

Here we see some measures to reduce the execution time. The following approaches do not reduce the complexity of the algorithm, but may provide significant reduction in execution time.

(1) *Identify Infeasible Use-Cases.* Our intention is to be able to analyze all the use-cases that we can with our given hardware resources. Identifying the infeasible use-cases reduces the potential set of use-cases. While the execution time can be significantly affected depending on their number, the worst-case complexity remains the same.

(2) *Reduce Feasible Use-Cases.* This method identifies all those use-cases that are *proper* subsets of feasible use-cases; such use-cases are defined as trivial, while those not included in any other feasible use-case are defined as nontrivial. When a use-case is feasible, all of its subsets are also feasible. Formally, if a use-case U_i is a subset of U_j, the minimal hardware needed for U_j is sufficient for U_i. (A proper subset here implies that all the applications executing concurrently in U_i are also executing in U_j, though the inverse is not necessarily true.) In other words, any partition that supports use-case U_j will also support U_i. It should be noted, however, that the performance of applications in these two use-cases may not be the same due to different sets of active applications, and therefore it might be required to evaluate performance of both use-cases.

These approaches are very effective and may significantly reduce the number of feasible use-cases left for analysis. With a scenario of 10 randomly generated

```
Procedure: FirstFitSetCoverHeuristic
 1: // Let $tmp_{ij}$ and $final_{ij}$ be two communication matrices, both initialized to zero
 2: // $final_{ij}$ stores the matrix that includes all the use-cases that fit in the current partition
 3: $UseCaseDone[\,] = 0$                                  // Initialize all use-cases as not done
 4: $UseCaseDone[i] = -1 \; \forall \; i$ when $U_i$ is infeasible            // Ignore the infeasible use-cases
 5: $UseCaseDone[i] = j + 2 \; \forall \; i$ when $U_i$ is a sub-set of $U_j$  // Reduction step
 6: $Partition[k]$ stores the use-cases that are assigned to the $k$-th partition; $k = 0$
 7: while Use-cases left (Translates to UseCaseDone[i]=0 for at least one i) do
 8:    $tmp_{ij} = 0$ and $final_{ij} = 0$
 9:    for all UseCases $U_i$ when $UseCaseDone[i] = 0$ do
10:       $tmp_{ij} = final_{ij}$
11:       Update $tmp_{ij}$ by adding UseCase $U_i$
12:       if $tmp_{ij}$ fits in device then
13:          $UseCaseDone[i] = 1$
14:          Add $i$ to $Partition[k]$
15:          $final_{ij} = tmp_{ij}$
16:       end if
17:    end for
18:    $k = k + 1$                                        // Advance partition
19: end while
20: // $Partition[\,]$ stores details of all the partitions
21: // $k$ is the number of partitions created
```

Fig. 8. Algorithm for partitioning the use-cases, with polynomial complexity.

applications and 1023 use-cases (considering all the possibilities), we found that only 853 were feasible. The reduction technique further reduced the number of nontrivial use-cases to 178. The aforesaid approaches reduce the execution time, but do not help in dealing with complexity. However, the optimality of the solution (in generation of feasible sets, not in the set-cover) is maintained.

6.3 Reducing the Complexity

In this section, we propose a simple heuristic to compute the partitions. This heuristic reduces the complexity significantly, albeit at the cost of optimality. As mentioned earlier, the greedy approach of partitioning requires to compute the largest feasible set. Since computing the largest optimal set has a high complexity, we have an alternative implementation which simply gives the first partition that includes the first nonincluded use-case, and simply scans the whole list to check which use-cases can be added such that the set remains feasible. The algorithm is as shown in Figure 8. An array is maintained to check which use-cases are not yet included in any partition (in Figure 8, UseCaseDone). Infeasible use-cases are indicated in step 4 in the figure. Use-cases are then reduced by considering only nontrivial use-cases. Trivial use-cases are assigned the identifier of its superset (step 5). This reduces the number of use-cases that are to be analyzed.

Partitions are then created on a first-come-first-serve basis. The order of use-cases in the input may therefore affect partitioning. As can be seen, once a use-case fits in a partition, it is not checked whether this is the optimal partition. In the worst-case, each use-case might result in its own partition, and the algorithm would then require $O(M)$ iterations of the while-loop, each requiring

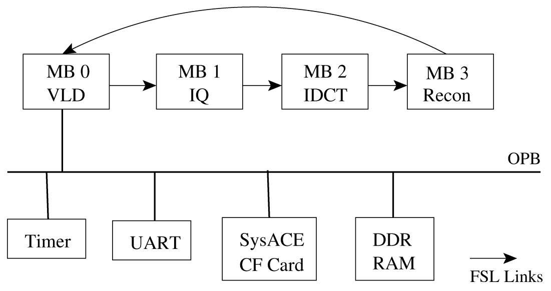


Fig. 9. Hardware topology of the generated design for H263.

M passes in the for-loop. Therefore, the total complexity of this approach is $O(M^2)$ as compared to $O(2^M)$ in the original approach. Section 9 compares the execution times of the two approaches.

7. TOOL IMPLEMENTATION

In this section, we describe the tool we developed based on our flow to target Xilinx FPGA architecture. The processors in the MPSoC flow are mapped to Microblaze processors [Xilinx 2007]. The FIFO links are mapped onto fast simplex links (FSL). These are unidirectional point-to-point communication channels used to perform fast communication. The FSL depth is set according to the buffer size specified in the application description.

Example architecture for the H263 application platform is shown in Figure 9 according to the specification in Figure 12(a). This consists of several Microblazes with each actor mapped to a unique processor, with additional peripherals such as Timer, UART, SysACE, and DDRRAM. While the UART is useful for debugging the system, the SysACE compact flash card allows for convenient performance evaluation for multiple use-cases by running continuously without external user interaction. The timer module and DDR RAM are used for profiling the application and for external memory access, respectively.

In our tool, in addition to the hardware topology, the corresponding software for each processing core is also generated automatically. For each software project, appropriate functions are inserted that model the behavior of the task mapped on the processor. This can be a simple delay function if the behavior is not specified. If the actual source-code for the function is provided, the same can be easily inserted into the project. This also allows functional verification of applications on a real hardware platform. Routines for measuring performance, as well as sending results to the serial port and CF card on-board are also generated for MB0.

Our software generation ensures that the tokens are read from (and written to) the appropriate FSL link in order to maintain progress and to ensure correct functionality. Writing data to the wrong link can easily throw the system in deadlock. XPS project files are also automatically generated to provide the necessary interface between hardware and software components.

8. AREA ESTIMATE: DOES IT FIT?

Whenever one talks about FPGA design, resource limitations are a major issue, and it is always important to know whether the desired design fits in the limited FPGA resources. Especially because hardware synthesis takes so much time, if the design finally does not fit on the target architecture, a lot of precious time is wasted and makes research considerably slower. In this section, we therefore provide the formulae that can be directly applied to compute how much area the design takes on the target platform. Our experiments were done on a Xilinx University Board containing a Virtex II Pro XC2VP30, and the same methodology can be applied to compute similar formulae for other target architectures as well. Here ISE 7.1i and EDK 7.1i were used for synthesis; however, the results are accurate for ISE 8.2i unless otherwise mentioned.

An FSL can be implemented either using block RAMs (BRAMs) or using LUTs in the FPGA. In the LUT implementation, the FIFO is synthesized using logic, while in BRAM implementation, embedded dual-port block-RAM blocks are used to synthesize these channels. Since both are crucial resources, we did the whole experiment with both these options. The following four sets of experiments were done.

—*Vary FSL, with BRAM*. This is the base design of one Microblaze and one FSL, incrementing FSL count to eight, with FSLs implemented using BRAMs.

—*Vary FSL, with Logic.* This consists of a base design of one Microblaze and one FSL, incrementing FSL count to eight, with FSLs implemented using logic.

—*Vary Microblaze, with BRAM FSL.* We use a base design of one Microblaze and eight FSLs, incrementing Microblaze count to eight, with FSLs implemented using BRAMs.

—*Vary Microblaze, with Logic FSL.* This has a base design of one Microblaze and eight FSLs, incrementing Microblaze count to eight, with FSLs implemented using logic.

Each FSL was set to a depth of 128 elements.[4] For a 32-bit element this translates to 512-byte memory. A BRAM in this device can hold 2kB of data, translating to 512 elements per BRAM. The number of slices, LUTs, and BRAMs utilized were measured for all experiments. Results of the first two sets are shown in Figure 10 and of the next two are shown in Figure 11. The increase in the total logic utilized is fairly linear, as expected. In the Virtex II Pro family each FPGA slice contains 2 LUTs, but often not both are used. Thus, we need to take slice utilization also into account. LUT utilization is shown as the measure of logic utilized in the design.

Table II shows the resource utilization for different components in the design obtained by applying a linear regression technique on the results of experiments. The second column shows the total resources present in XC2VP30. The next column shows the utilization in basic design to implement OPB

[4]It is also possible to set the FIFO depth in the specification, but we used a constant number for this study to minimize the number of variables.

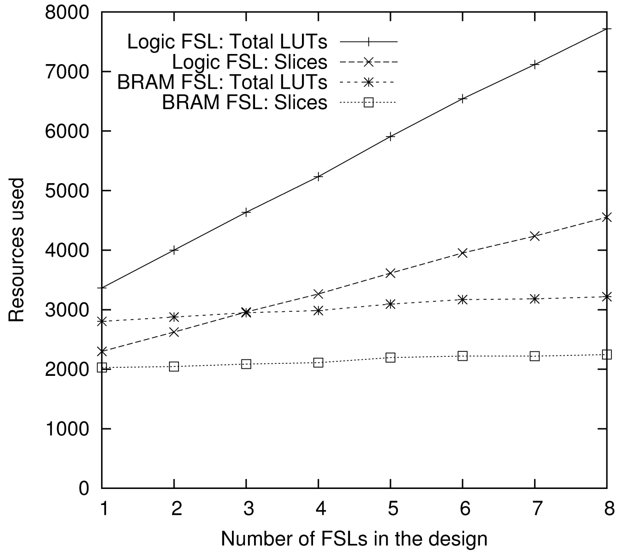


Fig. 10. Increase in the number of LUTs and FPGA slices used with changes in the number of FSLs in design.

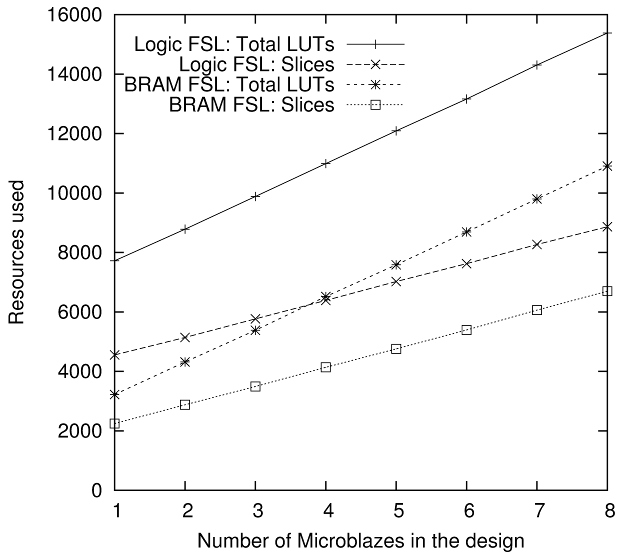


Fig. 11. Increase in the number of LUTs and FPGA slices used as the number of Microblaze processors is increased.

(on-chip peripheral bus), the CF card controller, timer, and serial I/O. The next two columns show the resources used for each dedicated point-to-point channel in the design. With the 8.2i version we observed that the minimum number of BRAM used for each FSL was 1, as compared to 2 in 7.1i. In our designs this translated to being able to accommodate a lot more FIFO channels on the same device with ISE/EDK 8.2i.

Table II. Resource Utilization for Different Components in the Design

			Each Fast Simplex Link		
	Total	Base Design	BRAM Impl	Logic Impl	Each Microblaze
BRAM	136	0	1 (2 in 7.1i)	0	4 (32)
LUTs	27392	1646	60	622	1099
Slices	13696	1360	32	322	636

The last column in Table II shows the same for Microblaze in the design. In our design, one Microblaze is assigned the task of communicating with the host and writing the results to the CF card. This core was designed with a much bigger memory for instruction and data. It uses 32 BRAMs in total, translating to 32kB memory each for data and instructions. The other cores have a much smaller memory at only 4kB each for data and instructions.

It is easy to obtain the total resource count that will be utilized upon synthesis. In our tool we also output the same and use it as a means to estimate whether the design would fit in the given resources. In all our experiments so far, our estimates have been very accurate and differ by hardly 1% or 2% when compared to actual resource utilization. For BRAM, the estimate is always exact.

8.1 Packing the Most

In our tool, we first try to assign as many FSL channels to BRAM as possible. When all the BRAMs on the device are used up, we assign them to LUT. The BRAM implementation is faster to synthesize, since only the access logic to memory has to be synthesized, while in LUT implementation the whole FIFO is constructed using logic. It should be noted, however, that since BRAM implementation assigns the whole memory block in discrete amounts, it might be a waste of resources to assign the whole block when a FIFO of small depth is needed. Currently, this tradeoff is not taken into account in our tool.

9. EXPERIMENTS AND RESULTS

In this section, we present some of the results that were obtained by implementing several real and randomly generated application SDF graphs, using our design flow described in Section 4. Here we show that our flow reduces the implementation gap between system-level and RTL-level design, and allows for more accurate performance evaluation using an emulation platform compared to simulation [Theelen et al. 2007] and analysis. In addition, we present a case study using JPEG and H263 applications to show how our tool can be used for efficient design-space exploration by supporting multiple use-cases. Further, we see how our use-case partitioning approach minimizes the number of hardware designs, by studying an example of applications running in a high-end mobile phone.

Our implementation platform is the 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 Core at 3GHz with 1GB of RAM.

Table III. Comparison of Throughput for Different Applications Obtained on FPGA with Simulation

	Appl 0			Appl 1		
Use-case	Sim	FPGA	Var %	Sim	FPGA	Var %
A	3.96	3.30	−20.05	1.99	2.15	**7.49**
B	3.59	3.31	−8.63	1.80	1.61	−11.90
C	2.64	2.74	**3.67**	1.88	1.60	−17.37
D	0.85	0.77	−10.51	3.82	3.59	−6.32
E	1.44	1.35	−6.80	4.31	4.04	−6.82
F	0.51	0.48	−5.79	5.10	4.73	−7.75
G	4.45	4.25	−4.55	1.11	0.97	−14.66
H	1.16	1.05	−10.29	4.63	4.18	−10.65
I	4.54	4.03	−12.48	2.27	2.13	−6.51
J	4.33	3.97	−8.92	1.08	1.00	−8.41
Average	–	–	−8.44	–	–	−8.29

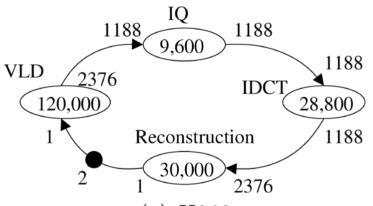
9.1 Generating Multiapplication Systems

In order to verify our design flow, we generated 10 random application graphs with 8 to 10 actors each, using the tool SDF^3 [Stuijk et al. 2006b], and generated designs with 2 applications running concurrently. Results of 10 such random combinations have been summarized in Table III. The results are compared with those obtained through simulation. We observe that in general, the application throughput measured on FPGAs is lower than simulation by about 8%. This is because our simulation model does not take into account the communication overhead. However, in some cases we observe that performance of some applications improves (shown in bold in Table III). This is rather unexpected, but easily explained when going into a bit of detail.

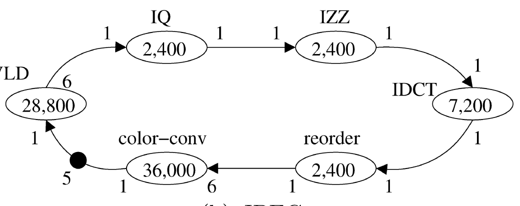
Communication overhead leads to the actor execution taking somewhat longer than expected, thereby delaying the start of the successive actor. This causes the performance of that application to drop. However, since we are dealing with multiple application use-cases, this late arrival of one actor might cause the other application to execute earlier than that in simulation. This is exactly what we see in the results. For the two use-cases in which this happens, namely A and C, the throughput of the other applications is significantly lower: 20% and 17%, respectively. This also proves that the use-cases of multiple applications concurrently executing are more complex to analyze and reason about than a single application.

9.2 Supporting Multiple Use-Cases

Here we present a case study of using our design methodology of supporting multiple use-cases for doing a design-space exploration and computing the optimal buffer requirement. Minimizing buffer size is an important objective when designing embedded systems. We explore the tradeoff between buffer size used and throughput obtained for multiple applications. For single applications, the analysis is easier and has been presented earlier [Stuijk et al. 2006a]. For multiple applications, it is nontrivial to predict resource usage and performance because multiple applications cause interference when they compete for resources.



(a) H263



(b) JPEG

Fig. 12. SDF graphs for H263 and JPEG applications.

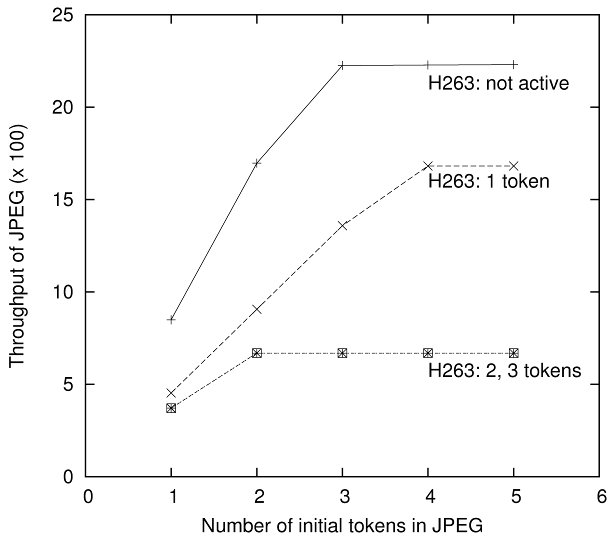


Fig. 13. Effect of varying initial tokens on JPEG throughput.

The case study is performed for JPEG and H263 decoder applications. The SDF models of the two applications were obtained from the description in de Kock [2002] and Hoes [2004], respectively, and the corresponding graphs are shown in Figures 12(a) and 12(b). In this case study, the buffer size has been modeled by the initial tokens present on the incoming edge of the first actor. The higher this initial-token count, the higher the buffer needed to store the output data. In the case of H263, each token corresponds to an entire decoded frame, while in the case of JPEG, it is the complete image. This is a special case of multiple use-case description, where the application structure remains the same but buffer size varies with each use-case. Therefore, the hardware remains unchanged, but the software is changed in each iteration.

Figure 13 shows how the throughput of the JPEG decoder varies with an increasing number of tokens in the graph. A couple of observations can be made from this figure. When the number of tokens (i.e., buffer size in real application) is increased, the throughput also increases until a certain point, after which it saturates. When the JPEG decoder is the only application running (obtained by setting the initial tokens in H263 to zero), we observe that its throughput increases almost linearly until three. We further observe that increasing the

Table IV. Throughput of the Two Applications Obtained by Varying Initial Number of Tokens (i.e., buffer size)

H263	0		1		2		3	
JPEG	H263	JPEG	H263	JPEG	H263	JPEG	H263	JPEG
0	–	–	458	–	830	–	830	–
1	–	849	453	453	741	371	741	371
2	–	1697	453	906	669	669	669	669
3	–	2226	454	1358	669	669	669	669
4	–	2228	422	1682	669	669	669	669
5	–	2230	422	1682	669	669	669	669

Table V. Time Spent on DSE of JPEG-H263 Combination

	Manual Design	Generating Single Design	Complete DSE
Hardware Generation	$\sim$ 2 days	40ms	40ms
Software Generation	$\sim$ 3 days	60ms	60ms
Hardware Synthesis	35:40	35:40	35:40
Software Synthesis	0:25	0:25	10:00
Total Time	$\sim$ 5 days	36:05	45:40
Iterations	1	1	24
Average Time	$\sim$ 5 days	36:05	1:54
Speedup	–	1	19

initial tokens of H263 worsens the performance of JPEG, but only until a certain point.

The actual throughput measured for both applications is summarized in Table IV. Increasing initial tokens for H263 beyond two causes no change, while for JPEG the performance almost saturates at four initial tokens. This analysis allows the designer to choose the desired performance-buffer tradeoff for the combined execution of JPEG and H263.

Design time. The time spent on exploration is an important aspect when estimating the performance of big designs. The JPEG-H263 MPSoC platform was also designed by-hand to estimate the time gained by using our tool. The hardware and software development took about 5 days in total to obtain an operational system. In contrast, our tool takes a mere 100 milliseconds to generate the complete design. Table V shows the time spent on various parts of the flow. The Xilinx tools take about 36 minutes to generate the bit-file, together with the appropriate instruction and data memories for each core in the design.

Since the software-synthesis step takes only about 25 seconds in our case study, the entire DSE for 24 design points was carried out in about 45 minutes. This hardware-software codesign approach results in a speedup of about 19 when compared to generating a new hardware for each iteration. As the number of design points increases, the cost of generating the hardware becomes negligible and each iteration takes only 25 seconds. The design occupies about 40% of logic resources on the FPGA and close to 50% of available memory. This study is only an illustration of the usefulness of our approach for DSE for multiprocessor systems.

9.3 Use-Case Partitioning

In this section, we show the effectiveness of our approach to partition use-cases, and the heuristics to optimize on the execution time. This is demonstrated first using some random test cases and then with a case study involving applications in a mobile phone.

Using the 10 applications we generated in the first part of the experiment, we generated all possible combinations, giving a total of 1023 use-cases. We found that only 853 of these were feasible; the rest required more resources than were present on our FPGA device. In general, most use-cases of up to 6 applications could fit on the FPGA, while only a couple of use-cases with 7 applications were feasible.

When trying to compute partitions using the greedy method directly on these 853 use-cases, the algorithm terminated after 30 minutes without any result, since there were too many sets to consider. When using the first-fit heuristic on these use-cases we obtained a total of 145 partitions in 500 milliseconds. However, since this approach is dependent on the order of use-cases, another order gave us a partition count of 126 in about 400 milliseconds. After applying our reduction technique on feasible use-cases, 178 nontrivial use-cases were obtained. The greedy approach on these use-cases terminated in 3.3 seconds and resulted in 112 partitions. The first-fit heuristic on the nontrivial cases took 300 milliseconds and gave 125 partitions, while another order of use-cases gave 116 partitions in about the same time.

A couple of observations can be made from this. Our techniques of use-case reduction are very effective in pruning the search space. Up to 80% of the use-cases are pruned away as trivial. This is essential in this case, for example, when otherwise no results are obtained for greedy. We observe that while the first-fit heuristic is a lot faster, the results depend heavily on the order of input use-cases. However, if the search space is large, first-fit may be the only heuristic for obtaining results.

Mobile-Phone case study. Here we consider 6 applications: video encoding (H263) [Hoes 2004], video decoding, JPEG decoding [de Kock 2002], mp3 decoding, modem [Bhattacharyya et al. 1999], and regular call. We first constructed all possible use-cases, giving 63 use-cases in total. Some of these use-cases are not realistic, for example, JPEG decoding is unlikely to run together with video encoding or decoding because when a person is recording or watching video, he/she will not be browsing the pictures. Similarly, listening to mp3 while talking on the phone is unrealistic. After pruning away such unrealistic use-cases we were left with 23 use-cases. After reduction to nontrivial use-cases, only 3 remained.

A greedy approach only works on the set after reduction. We observe that 23 use-cases is too many to handle if there are a lot of possible subsets. (In the previous example with 10 applications, we obtain 178 use-cases after reduction but since no partition can handle more than 4 use-cases, the total number of possible sets is limited.) After reduction, however, the greedy algorithm gives 2 partitions in 180 milliseconds. The same results are obtained with the first-fit heuristic. However, the first-fit heuristic also solves the problem without

Table VI. Performance Evaluation of Heuristics Used for Use-Case Reduction and Partitioning

		Random Graphs		Mobile Phone	
		# Partitions	Time (ms)	# Partitions	Time (ms)
Without Reduction	Without Merging	853	–	23	–
	Greedy	Out of Memory	–	Out of Memory	–
	First-Fit	126	400	2	200
With Reduction	Without Merging	178	100	3	40
	Greedy	112	3300	2	180
	First-Fit	116	300	2	180
Optimal Partitions		≥ 110	–	2	–
Reduction Factor		7	–	11	–

pruning away the use-cases. Here the order only affects which trivial use-cases are attached to nontrivial use-cases. In total, since we have only 2 partitions, performance of all the 23 use-cases is measured in about 2 hours. Without this reduction it would have taken close to 23 hours. The use-case merging and partitioning approach leads to an elevenfold reduction. The results are fed to the computer and stored on the CF card for later retrieval.

Table VI shows how well our use-case reduction and partitioning heuristics perform. The time spent in corresponding steps are also shown. Reduction to nontrivial use-cases for the mobile-phone case study takes 40 milliseconds, for example, and leaves us with only 3 use-cases. As mentioned earlier, the greedy heuristic for partitioning does not terminate with the available memory resources when applied without reducing the use-cases. The design space is too large to evaluate the largest feasible set. After reducing to nonfeasible use-cases for random graphs, we obtain 178 use-cases and at most 4 use-cases fit in any partition. Since the maximum error in using the greedy approach is given by $ln|X| + 1$, where X is the number of elements in the largest partition, we get a maximum error of $ln|4| + 1$, namely, 2.38. We can therefore be sure that the minimal number of partitions is at least 110. We see a sevenfold reduction in the number of hardware configurations in the random-graphs use-case and about elevenfold in the mobile-phone case study. We can therefore conclude that our heuristics of use-case reduction and partition are very effective in reducing the design time and number of partitions.

Reconfiguration time. The time to reconfigure an FPGA varies with the size of configuration file and the mode of reconfiguration. For Virtex II Pro 30, the configuration file is about 11Mb. The CF-card controller provides configuration bandwidth of 30Mb per second, translating to about 370 milliseconds for reconfiguration. Configuring through the on-board programmable memory is a lot faster, since it provides bandwidth of up to 800Mb per second. Thus, for the aforementioned FPGA device it takes only about 13 milliseconds. The USB connection is a lot slower, and often takes about 8 seconds. However, for the end-design, we expect the configurations to be stored in an on-board programmable memory; these are retrieved as and when use-cases are enabled. A typical mobile-phone user is unlikely to start a new use-case more than once every few minutes. Therefore, the reconfiguration overhead of 13 milliseconds is not significantly large, and is amortized over the period of use-case.

10. CONCLUSIONS AND DISCUSSION

In this article, we propose a design flow to generate architecture designs for multiple use-cases. Our approach takes the description of multiple use-cases and produces the corresponding MPSoC platform. A use-case is defined as a set of concurrently active applications. This is the first flow that allows mapping of multiple applications, let alone multiple use-cases on a single platform. We propose techniques to merge and partition use-cases in order to minimize hardware requirements. The tool developed using this flow has been made available online for the benefit of the research community [MAMPS 2007], and a stand-alone GUI tool is developed for both Windows and Linux. The flow allows designers to traverse the design space quickly, thus making DSE, even of concurrently executing applications, feasible. A case study is presented to find the tradeoffs between buffer size and the performance when JPEG and H263 run together on a platform. A multiple use-case study for a smart phone is also presented that requires only two partitions to evaluate the performance of multiple use-cases, requiring a total of two hours.

Further, we also provide techniques to estimate resource utilization in the FPGA without carrying out the actual synthesis. While the number of applications that can be concurrently mapped on the FPGA is limited by the hardware resources, the number of use-cases that can be merged in one hardware design depends on the similarity of hardware requirements in the use-cases. When synthesizing designs with applications of eight to ten actors and twelve to fifteen channels, we found it difficult to map more than six applications simultaneously due to resource constraints. A bigger FPGA would certainly allow bigger designs to be tested, and possibly all use-cases could be merged in one description.

Our technique is also capable of minimizing the number of reconfigurations in the system. The use-case partitioning algorithm can be adapted to consider the relative frequency of the use of each use-case. The use-cases should be first sorted in decreasing order of their use, and then the first-fit algorithm proposed in an earlier section should be applied. The algorithm will therefore first pack all the most frequently used use-cases together in one hardware partition, thereby reducing the reconfiguration from one frequently used use-case into another. However, for an optimal solution of the partition problem, many other parameters need to be taken into account, for example, the reconfiguration time and average duration for each use-case. We would like to extend the use-case partitioning algorithm to take the exact reconfiguration overhead into account.

Further, we would like to develop and automate more ways of design-space exploration, for example, trying different partitions of applications, and to support arbitrary mapping of actors to processors. We would also like to try different kinds of arbiters in the design to improve fairness and allow for load-balancing between multiple applications. For the tooling itself, we wish to extend MAMPS to include support for generating heterogeneous platforms in our flow.

REFERENCES

Bhattacharyya, S., Murthy, P., and Lee, E. 1999. Synthesis of embedded software from synchronous dataflow Specifications. *The J. VLSI Signal Process. 21,* 2, 151–166.

CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA.

DE KOCK, E. 2002. Multiprocessor mapping of process networks: A JPEG decoding case study. In *Proceedings of the 15th ISSS Conference,* Los, Alamitos, CA. IEEE Computer Society, 68–73.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman, New York.

HOES, R. 2004. Predictable dynamic behavior in NoC-based MPSoC. www.es.ele.tue.nl/epicurus/.

HOL. 2007. Head-of-line blocking. `http://en.wikipedia.org/wiki/Head-of-line\_blocking`.

JERRAYA, A. AND WOLF, W. 2004. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, San Francisco, CA.

JIN, Y., SATISH, N., RAVINDRAN, K., AND KEUTZER, K. 2005. An automated exploration framework for FPGA-based soft multiprocessor systems. In *Proceedings of the 3rd CODES+ISSS International Workshop on Hardware/Software Codesign*, Los Alamitos, CA. IEEE Computer Society, 273–278.

KAHN, G. 1974. The semantics of a simple language for parallel programming. *Inf. Process. 74*, 471–475.

KUMAR, A., FERNANDO, S., HA, Y., MESMAN, B., AND CORPORAAL, H. 2007a. Multi-Processor system-level synthesis for multiple applications on platform FPGA. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications*, 92–97.

KUMAR, A., HANSSON, A., HUISKEN, J., AND CORPORAAL, H. 2007b. An FPGA design flow for reconfigurable network-based multi-processor systems on chip. In *Proceedings of the Design Automation and Test in Europe (DATE)*, Los Alamitos, CA. IEEE Computer Society, 117–122.

KUMAR, A., MESMAN, B., CORPORAAL, H., VAN MEERBERGEN, J., AND YAJUN, H. 2006a. Global analysis of resource arbitration for MPSoC. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design (DSD)*. Los Alamitos, CA. IEEE Computer Society, 71–78.

KUMAR, A., MESMAN, B., THEELEN, B., CORPORAAL, H., AND HA, Y. 2006b. Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In *Proceedings of the 4th Workshop on Embedded Systems for Real-Time Multimedia (Estimedia)*. IEEE Computer Society, 33–38.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Trans. Comput. 36,* 1 (Feb.), 24–35.

LYONNARD, D., YOO, S., BAGHDADI, A., AND JERRAYA, A. 2001. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the Design Automation Conference*. ACM Press, New York, 518–523.

MAMPS. 2007. Multiple applications mutli-processor synthesis. Username: `todaes`, Password: `guest`. `http://www.es.ele.tue.nl/mamps/`.

MURALI, S., COENEN, M., RADULESCU, A., GOOSSENS, K., AND DE MICHELI, G. 2006. A methodology for mapping multiple use-cases onto networks on chips. In *Proceedings of Design Automation and Test in Europe (DATE)*. IEEE Computer Society, 118–123.

NIKOLOV, H., STEFANOV, T., AND DEPRETTERE, E. 2006. Multi-Processor system design with ESPAM. In *Proceedings of the 4th International Workshop on Hardware/Software Codesign (CODES+ISSS)*. ACM Press, New York, 211–216.

PAUL, J. M., THOMAS, D. E., AND BOBREK, A. 2006. Scenario-Oriented design for single-chip heterogeneous multiprocessors. *IEEE Trans. Very Large Scale Integr. Syst. 14,* 8 (Aug.), 868–880.

SRIRAM, S. AND BHATTACHARYYA, S. 2000. *Embedded Multiprocessors; Scheduling and Synchronization*. Marcel Dekker, New York.

STEFANOV, T., ZISSULESCU, C., TURJAN, A., KIENHUIS, B., AND DEPRETTE, E. 2004. System design using Kahn process networks: The Compaan/Laura approach. In *Proceedings of the Design Automation and Test in Europe (DATE)*. IEEE Computer Society, 340–345.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006a. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*. ACM Press, New York, 899–904.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006b. SDF3: SDF for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD).* IEEE Computer Society, 276–278.

THEELEN, B., FLORESCU, O., GEILEN, M., HUANG, J., VAN DER PUTTEN, P., AND VOETEN, J. 2007. Software/Hardware engineering with the parallel object-oriented specification langauge. In *Proceedings of the 5th ACM-IEEE International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 139–148.

XILINX. 2007. Xilinx resource page. `http://www.xilinx.com`.

Received September 2007; revised February 2008; accepted March 2008