An Automated Flow to Map Throughput Constrained Applications to a MPSoC*

Roel Jordans[1], Firew Siyoum[1], Sander Stuijk[1], Akash Kumar[1,2], and Henk Corporaal[1]

1 Eindhoven University of Technology, The Netherlands
2 National University of Singapore, Singapore

Abstract

This paper describes a design flow to map throughput constrained applications on a Multiprocessor System-on-Chip (MPSoC). It integrates several state-of-the-art mapping and synthesis tools into an automated tool flow. This flow takes as input a throughput constrained application, modeled with a synchronous dataflow graph, a C-based implementation for each actor in the graph, and a template based architecture description. Using these inputs, the tool flow generates an MPSoC platform tailored to the application requirements and it subsequently maps the application to this platform. The output of the flow is an FPGA programmable bit file. An easily extensible template based architecture is presented, this architecture allows fast and flexible generation of a predictable platform that can be synthesized using the presented tool flow. The effectiveness of the tool flow is demonstrated by mapping an MJPEG-decoder onto our MPSoC platform. This case study shows that our flow is able to provide a tight, conservative bound on the worst-case throughput of the FPGA implementation. The presented tool flow is freely available at `http://www.es.ele.tue.nl/mamps`.

1998 ACM Subject Classification C.3 [Special-purpose and application-based systems]: real-time embedded systems

Keywords and phrases design flow automation, multi-processor system-on-chip, throughput constrained, synchronous data-flow graphs

Digital Object Identifier 10.4230/OASIcs.PPES.2011.47

1 Introduction

New applications for embedded systems demand complex multiprocessor designs to meet real-time deadlines while achieving other critical design constraints like low energy consumption and low area usage. Multiprocessor Systems-on-Chip (MPSoCs) have been proposed as a promising solution for such problems but the design space exploration of such systems typically involves many parameters. Higher abstraction levels, possibly combined with early and accurate performance predictions, of the designed system are therefore required to make good design choices. Several tool-flows [6, 10, 13] have been proposed to solve this problem, but these solutions still require manual design steps which are time consuming and error-prone. Combining existing tools into a common design flow has proven non-trivial [12] without careful planning and coordination of the tool development.

In this paper, we present a design flow (see Figure 1) which bundles the strengths of both the SDF^3 [14] tool set and the MAMPS [8] platform. The SDF^3 tool set supports analyzing

* This work was partly funded by the PROGRESS program of the Dutch Technology Foundation STW through the PreMaDoNa project EES. 6390.

and mapping synchronous data-flow (SDF) graphs [9]. SDF^3 uses a graph representation of the application and a set of models of the hardware platform to calculate the worst-case throughput of the application for a given mapping of tasks on the platform. MAMPS provides a tool to generate MPSoC projects for a Xilinx FPGA platform including software and hardware synthesis based on a SDF description of one or more applications and a task mapping. MAMPS has been almost completely rewritten as part of this work, new communication options have been added and the generated hardware and software have been modeled into SDF^3. This ensures that the MAMPS implementation of any mapping produced by SDF^3 can be guaranteed to meet or exceed the throughput guarantee provided by SDF^3 and thus produce a predictable system.

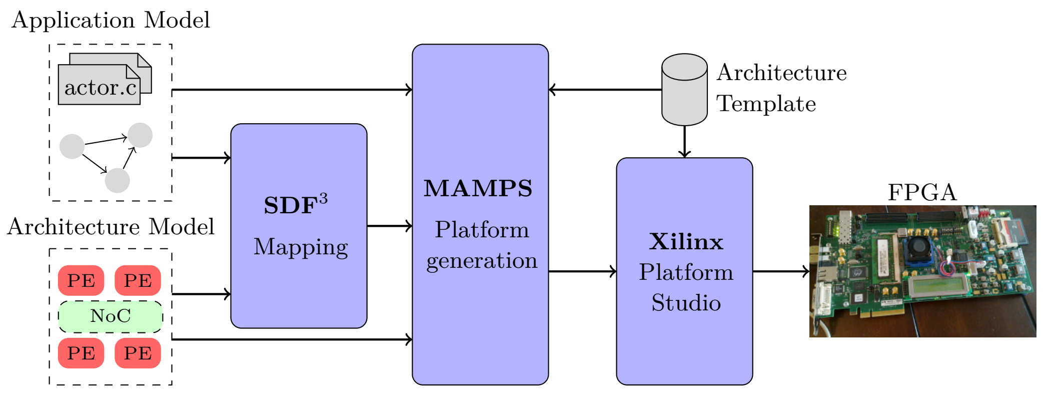


Figure 1 Design flow overview.

The remainder of this paper is organized as follows. Section 2 reviews the related work for automated MPSoC generation and performance prediction. Section 3 provides an overview of application modeling using SDF graphs. Section 4 gives an overview of the architecture of the MAMPS platform. The design flow is presented in Section 5 and the implementation changes to both SDF^3 and MAMPS are explained in this section. Section 6 presents an experiment used to validate the design flow and analyzes the design effort and design overhead of the flow. Section 7 concludes the paper and gives a direction 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 CA-MPSoC [13]. CA-MPSoC extends the MAMPS platform with a hardware communication assist (CA) which is responsible for the communication between the processing elements of the platform. The paper presents a SDF model for this CA controller and uses this model for performance prediction. However, the presented model has been simplified and lacks modeling of the communication channel. This paper improves the model by *a*) including the fragmentation of communicated tokens into words that can be sent over a network, and *b*) including a model for the communication channel on the network itself. The flow presented in [13] introduces options for deciding on a mapping of the application onto the generated platform but the method requires the user to manually translate the output format of the mapping tool into the interchange format of the platform generation tool. The flow presented in this paper automates this step by introducing a common input format for both the mapping and platform generation tools, circumventing possible user introduced errors during the translation step.

ESPAM [10, 11] presents a similar design flow as the flow presented in this paper. The ESPAM flow uses Kahn Process Networks (KPNs) to model the application. In our approach, we use SDF graphs in stead. SDF graphs are a subset of KPN graphs and therefore have a limited expressiveness when compared to KPN graphs. The disadvantage of using pure KPN for application modelling however is the limited possibilities for analyzing pure KPN graphs. It is, for example, impossible to analyze buffer requirements in a generic way when using KPN graphs but this analysis is possible for SDF graphs [2]. Another disadvantage of KPN over SDF is that KPN requires run-time buffer management and scheduling which make performance prediction difficult while SDF graphs can be completely analyzed at design time [14]. Our approach produces a predictable, throughput constrained solution whereas ESPAM is limited to an estimation of the performance. The PeaCE approach presented in [6] provides another method for hardware and software co-design. PeaCE uses two different extended versions of the SDF model and three different types of tasks for representing different parts of the application, requiring a relatively complex operating system. Our approach uses a pure SDF representation of the application and implements only a single task type resulting in a minimal implementation overhead. This comes however at the cost of a reduced expressiveness and therefore potentially an over dimensioning of our platform. The experimental results show however that this effect is limited.

3 Application Modelling

Figure 2 shows an example of an SDF graph. There are three actors in this graph. As in a typical data flow graph, a directed edge represents the dependency between actors. Actors consume input data from their input edges and/or produce output data on their output edges; such information is referred to as *tokens*. Tokens are shown in an SDF graph as dots on the edges, a number is added to these dots to show that multiple tokens are available. The number of tokens consumed by an actor is constant and can be read from the SDF graph next to the incoming vertex. An actor is called *ready* when it has sufficient input tokens on all its input edges. Actor execution is called *firing*, an actor can only fire when it is ready. An actor also produces a constant amount of tokens per firing denoted next to the outgoing end of each edge. SDF actors are stateless (i.e. no internal actor state is preserved between actor firings) so any actor state need to be modelled explicitly. Actor A in Figure 2 is an example of an actor which keeps state, implemented as the static variable in Listing 1, this state variable is modeled explicitly in Figure 2 by the self-edge of actor A.

Listing 1 Implementation of actor A

```
static int local_variable_A;

void actor_A_init(typeAtoB *, typeAtoC *) {
   local_variable_A = 0;
}

void actor_A (typeAtoB *toB, typeAtoC *toC) {
   // calculate something
   // and write the output tokens
   toB[0] = calculate_valueB1();
   toB[1] = calculate_valueB2();
   *toC = calculate_valueC(local_variable_A);
}
```

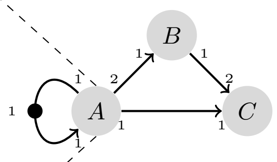


Figure 2 Example of an SDF graph together with the implementation of one of the actors.

An application is described using a graph. Edges may contain *initial tokens* as is shown on the self-edge of actor A in Figure 2. In the above example, only A can fire in the initial state, since the required number of tokens are present on all of its incoming edges. Once A has finished firing it will produce 2 tokens on its edge to B, 1 token on its edge to C and 1 token on its self-edge. B can then fire as it has enough tokens on its incoming edge to execute twice, each time producing 1 token on its edge to C.

Implementing an application using its SDF graph requires an implementation for each actor. Actor implementations consist of one actor implementation function which takes up to one parameter per edge connected to the actor. Not every edge needs to be explicitly implemented as a parameter to the actor implementation function. The self-edge of actor A is an example of an edge which is not explicitly implemented. Therefore we make a distinction between explicitly and implicitly implemented edges. Explicitly implemented edges implement connections between two actors which are transferring data. Implicitly implemented edges include, but are not limited to, the self-edges as shown above, but can also be used to model restrictions like limited buffer sizes on the edges connecting multiple actors as well as modeling a specific firing order as imposed by static order scheduling [14]. Only explicit edges are implemented as parameters of the actor implementation function. Listing 1 shows an example implementation of actor A. Two functions are created in this listing, an initialization function and the actor implementation. The actor implementation function `actor_A()` has two parameters, one for the edge to B and one for the edge to C, note that there are no parameters supplied for the implicit self-edge of A. Output tokens are written to the buffers provided as parameters. The initialization function, `actor_A_init`, is responsible for producing the initial tokens that are expected on the output edges of actor A, in this case the self-edge of A. The initialization function has the same signature as the main actor implementation but no space is reserved for edges that do not produce initial tokens and no input tokens are provided.

The application graph and the relation between the graph elements and their respective implementations are joined into the *application model*. The application model also specifies a set of metrics of the actor implementations. These metrics include the Worst-Case Execution Time (WCET), required memory sizes, and the size of communicated tokens. Memory size requirements are specified separately for both instruction and data memories in order to facilitate processing elements that use a Harvard architecture. The memory size requirement is used in the tool flow to automatically determine the memory requirements for each processing element. The WCET metrics and token sizes are used by the SDF^3 tools to calculate a lower bound on the throughput of the application. A good WCET estimate of each actor implementation is therefore important for the performance of the presented tool flow. Many different approaches exist for determining the WCET of (a part of) a program, either from the source code or some intermediate form. WCET tool challenges [5, 7] present insightful information about existing WCET analysis tools and techniques and [16] gives an in-depth analysis of the available methods as well as a survey of existing tools for WCET analysis. Any of these tools can be used to provide the WCET of actors for the presented design flow. It is possible that different (optimal) implementations of the same actor exist for the different types of processing element or tile configuration in the platform template. The application model can specify multiple implementations for each actor. Each implementation specification defines the relation between the function arguments of the implementation and the edges of the graph, the WCET and memory requirements of that specific implementation, and the type of processing element this implementation can be mapped to. This allows the tool flow to map the actors on a heterogeneous platform where actor implementations for different processing elements are likely to have different metrics.

4 Architecture Modelling

The second input of the design flow is the *architecture model* (see Figure 1). This model describes the various components available in the hardware platform and how these components are connected. The MAMPS platform allows two types of components in the architecture; tiles and interconnect. Tiles form the processing elements of the architecture and the interconnect is limited to connecting tiles together. A standardized network interface (NI) has been defined for connecting tiles to the interconnect. All tile and interconnect variants use this same network interface which makes it easy to compose a platform by using elements from an architecture template. Figure 3 shows an example of the MAMPS platform architecture. This example shows four different variations of a tile connected together through an interconnect. Tiles 1 and 2 in the example show simple tile architectures using a processing element (PE) which is connected to the network interface (NI), a local memory and some optional peripherals (i.e. IO, timers, etc.). Tile 3 shows a similar tile which has been extended with a communication assist (CA) which handles the memory management and serialization, sending, and receiving of tokens. The last tile, Tile 4, shows another option where a hardware implementation of an actor (IP) is connected directly to the interconnect using only a network interface.

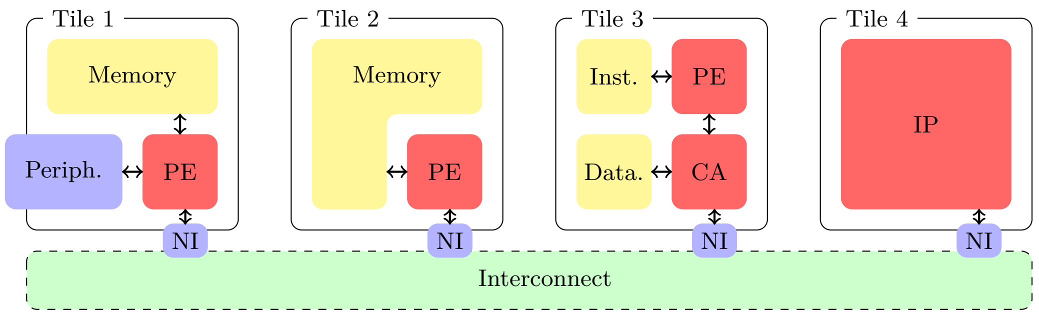


Figure 3 MAMPS platform architecture example showing different variations of a tile.

Running realistic applications on a system requires that one or more actors have access to peripherals. Predictability of the MAMPS platform is guaranteed by avoiding the sharing of peripherals over tiles. Another option for maintaining predictability while using shared peripherals is to use a predictable arbiter. [1] presents such an arbiter for SDRAM memories. The technique presented in [1] can be extended to include different types of resources and is easy to implement.

4.1 Network interface

A clear definition of the network interface is critical for the functioning of the template based architecture generation. The MAMPS platform defines the Xilinx Fast Simplex Link interface as network interface. This limits the network interface to communicating 32-bit words but also makes sure there is a trivial point-to-point solution for the interconnect by using Xilinx Fast Simplex Links (FSL) [15]. In order to translate arbitrarily sized tokens into one or more 32-bit words and back again requires serialization and de-serialization. These operations can either be performed by the processing element of the tile (i.e. the PE block in Tile 1 of Figure 3), or by the addition of some dedicated communication hardware (i.e. the CA block of Tile 3 in Figure 3).

The advantage of using the processing element for the serialization and de-serialization of tokens is the simplicity of the generated hardware. This simple hardware comes at the cost of extra processing time used on the processing element which can not be spent on running actor code. Using dedicated communication hardware like the CA described in [13] increases hardware complexity but also relieves the processing element from the serialization and de-serialization of tokens which improves the actor response-time.

4.2 Communication model

The communication introduced in MAMPS has been modeled in an SDF graph. This graph is used in SDF^3 to predict the behaviour of edges mapped to the interconnect. Figure 4 shows the parameterized model of communication via the interconnect. Three boxes divide this model into the parts representing the various phases in the communication of a token. The dashed edge in this graph shows the original connection in the SDF graph.

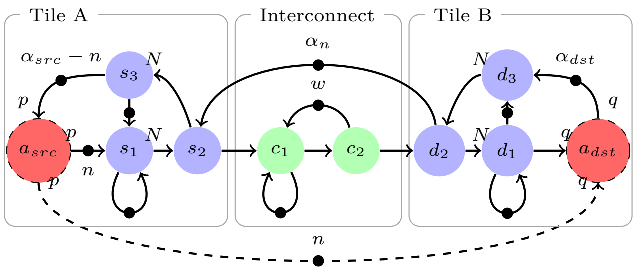


Figure 4 Parameterized model for communication over the interconnect. Missing port rates and token counts are to be interpreted as 1.

The central box models the interconnect behaviour, the model allows pipelined sending of words over the interconnect where the number of initial tokens w is equal to the maximum number of words in simultaneous transmission. The connections on the interconnect are also capable of buffering a number of α_n words in transmission. Actors c_1 and c_2 form a latency-rate model for the communication. Actors s_1, s_2 and s_3 model the serialization of the token into N 32-bit words by the network interface. The execution time of s_1 is dependant on the design of the serialization code while the execution times for s_2 and s_3 are set to 0 because these actors are only required for the modeling of the serialization of the tokens. Actors d_1, d_2 and d_3 model the de-serialization of the transmitted words into tokens at the receiving end and are assigned values in the same way as the serialization actors. Finally, α_{src} and α_{dst} model the available buffer space on the sending and receiving ends of the connection. The model in Figure 4 can be used for modeling communication over many different forms of interconnect by changing w, α_n, and the execution times of s_1, c_2, and d_1 to appropriate values.

5 Design flow

The design flow, as depicted in Figure 1 can be divided in three steps. The application should first be mapped onto the architecture. This mapping can then be combined with the original application and architecture specifications into a FPGA design which can, as a third step, be synthesized into a working system using out of the box FPGA development software. The goal of this flow is to produce a working implementation of the application on a given platform, capable of achieving the throughput as required for the application. The *throughput* of an application is defined in [3] as the long term average number of graph iterations per time unit. The long term average is used to avoid initialization effects from influencing the throughput. The design flow defines the system clock of the platform as its base time unit. This section provides a more in-depth description of the SDF^3 tool set, the MAMPS platform generation, and the currently available architecture components.

5.1 SDF^3

The SDF^3 tool set consists of several tools that allow automatic mapping of an application described as a SDF graph to a given platform. SDF^3 also verifies if such a mapping is deadlock free, calculates buffer distributions, and predicts which throughput can be guaranteed for this mapping. SDF^3 uses generic cost functions to steer the binding of the application to the architecture based on; processing, memory usage, communication, and latency. Buffer distributions, task mapping and static-order schedules are determined and gathered in the mapping output of SDF^3. The virtual platform of the SDF^3 tool set was modified to match the architecture and model of the MAMPS platform. The algorithms used during mapping have not been changed from those presented in [14].

5.2 MAMPS

The MAMPS tool set was completely rewritten as part of this research but the architecture and ideas remain the same. The platform is now generated by combining the information from the application and architecture models with the mapping output from SDF^3. Information from the architecture model and mapping are used to generate the hardware platform. Template components are instantiated and connected as required by the application. Memory sizes are calculated for each tile based on the mapped buffers, actors and the size of the scheduling and communication layer. The interconnect components are instantiated to match the specified communication architecture. Connections are routed and the VHDL code and peripheral driver for the interconnect are also generated when required. The software platform is generated next. This includes generating wrapper code for each actor, translating the static-order schedule provided by SDF^3 into C code, and generating initialization code for the communication. The generated code is combined with a template project which already includes an implementation of the scheduling and communication libraries. The XPS TCL script interface is then used to complete the project and to add the required hard and software targets for the implementation. Using the script interface ensures compatibility over many different versions of XPS and greatly simplifies the generated code.

5.3 Currently available in the architecture template

Not all blocks shown in Figure 3 are currently available in the tool flow. The MAMPS platform currently offers two forms of interconnect and two different tiles. The currently available architecture components are all targeted for the Xilinx Virtex6 FPGA using the Xilinx ML605 evaluation board. The subsections below discuss the available options.

5.3.1 Interconnect

Either point-to-point connections using Xilinx Fast Simplex Links (FSL) [15] or a Spatial Devision Multiplex (SDM) NoC based on [17] can be used for connecting the tiles. Both interconnects comply to the network interface definition but the NoC interconnect provides more flexibility at the cost of a larger implementation and a higher latency while the FSL interconnect simply uses the FSL implementation provided by Xilinx.

The NoC consists of one router per tile in the design. Each router connects through a set of wires to its neighbours. Each router can also be connected to the network interface of a single tile. The routers are arranged in a 2-dimensional mesh network. The dimensions of this network are based on the number of tiles required in the design and the network is kept as close to square as possible to reduce the maximum distance between two tiles since this distance relates directly to the latency of the network connections. The NoC allows the user to program connections on a point-to-point basis, each connection can be assigned a certain bandwidth through the number of wires assigned to that connection but wires can only be assigned to a single connection at a given time allowing an efficient usage of network resources. The original NoC presented in [17] already complied with the network interface requirements for the MAMPS platform but missed flow-control for connections in the network. Flow-control was added as part of the integration of the NoC in the MAMPS platform. The changes to the NoC required approximately 12% more slices on the FPGA when compared to the original implementation.

5.3.2 Tile template

As shown in Figure 3, a tile consists of a processing element (PE), an optional instruction and/or data memory, and a network interface (NI). MAMPS currently provides only two types of tiles. The first type is the master tile, this tile is similar to Tile 1 in Figure 3. It uses a Xilinx Microblaze soft-core as processing element, includes up to 256kB memory in a Modified Harvard configuration and has direct access to the peripherals on the FPGA board. The FSL ports of the Microblaze and a software library implementing (de-)serialization are used to implement the network interface. The second type of tile is the slave tile, this tile is the same as the master tile but does not have access to the peripherals and therefore is similar to Tile 2 in Figure 3.

6 Case study

The application used in the case study is the MJPEG decoder shown in Figure 5. The VLD actor parses the input file and decompresses the Minimal Coded Unit (MCU) blocks. MCUs consist of up to 10 blocks of frequency values, depending on the sampling settings used when creating the input file. Each block of frequency values is passed through the inverse quantization and zig-zag reordering (IQZZ) and IDCT actors which transform the frequency values into color components. The color conversion (CC) actor translates the color component blocks of one MCU to pixel values and the rasterization (Raster) actor puts the pixel values at the correct location in the output buffer. The subHeader1 and subHeader2 edges in the SDF graph forward information from the file header (i.e. frame size and color composition) to the CC and Raster actors. One graph iteration of the MJPEG decoder decodes a single MCU. This causes the throughput of the application to be defined in MCUs per clock cycle of the generated platform. A method based on [4] combined with execution time measurement was used to determine the WCET of the actors in this case study.

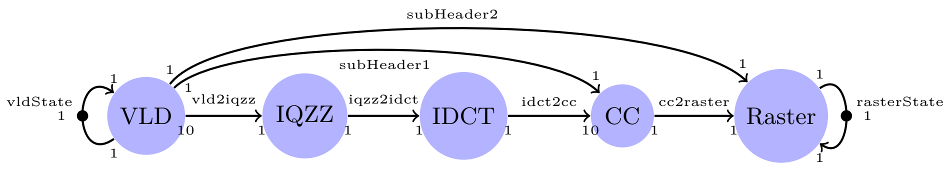
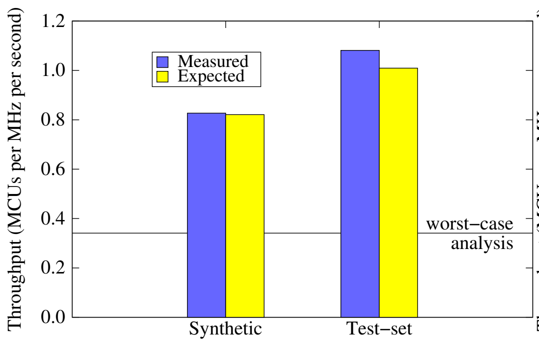


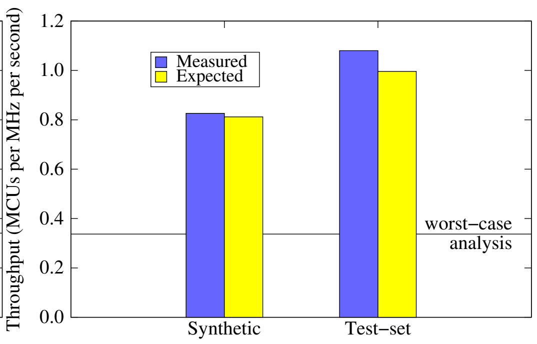
Figure 5 The SDF graph for the MJPEG decoder.

6.1 Throughput analysis

An important aspect of the presented design flow is the early throughput analysis of the designed application. The throughput of the MJPEG decoder was therefore measured on the FPGA implementation and compared to the predicted throughput of SDF^3. Figure 6 shows the worst-case throughput obtained by running the MJPEG decoder on 5 different test sequences and a synthetic sequence containing random data for two different architectures. The worst-case analysis line in both graphs shows the SDF^3 prediction based on the WCET of the actors. The expected values were calculated using SDF^3 by using WCET metrics obtained through execution time measurement of the actor code using the test-data used for the FPGA measurement. The difference between the expected throughput (blue) and measured throughput (yellow) shown in Figure 6 shows the margin of the used models (less than 1% for the synthetic data) when using actors with low variation in the execution time. Throughput at the worst-case analysis line is guaranteed by the flow.



(a) FSL interconnect



(b) NoC interconnect

Figure 6 Measured and predicted worst-case throughput for a synthetic test-sequence and a set of real-life test-sequences for two different forms of interconnect compared to the worst-case prediction of SDF^3

6.2 Designer effort

Table 1 lists the required designer effort in creating and mapping the MJPEG decoder as this was done by the authors of the paper. This implies a working understanding of the application as well as previous experience in writing applications for the design flow and platform. The top part of the table represents manual labour performed by the designer and the bottom part (marked with A) is automated by the presented design flow. Manually implementing the overall system would cost at least another 2–5 days depending on the complexity of the hardware (i.e. number of tiles) and the number of application mappings tried.

6.3 Overhead

The overhead of the generated system when compared to a manually developed system can be characterized in two categories, modeling and implementation overhead. The primary source of modeling overhead are the fixed output rates of the SDF actors. This can be seen in the MJPEG example at the output rate for the VLD actor which produces up

Table 1 Designer effort, steps marked with A are automated.

Step	**Time spent**	
Parallelizing the MJPEG code	< 3 days	
Creating the SDF graph	5 minutes	
Gathering required actor metrics	1 day	
Creating application model	1 hour	
Generating architecture model	1 second	A
Mapping the design (SDF^3)	1 minute	A
Generating Xilinx project (MAMPS)	16 seconds	A
Synthesis of the system	17 minutes	A
Total time spent	**∼ 4 days**	

to 10 frequency blocks per MCU depending on the format of the input stream. Another source of modeling overhead can be found in communicating the initialization values on the subHeader1 and subHeader2 channels in the example. A manual implementation of the algorithm could communicate these values separately from the main program flow during an initialization phase, it is not possible to model this using a single SDF graph. However, these initialization tokes are relatively small and use only 1% of the communication. The implementation overhead of SDF is also very small. Scheduling on the MAMPS platform is done through a static order schedule which reduces the scheduler to a lookup table. A manual implementation is likely to implement the same schedule in its main loop which is similar in efficiency. Communication would also be solved in a similar way and therefore does not influence the implementation overhead. The scheduling overhead will be similar for other applications but the modeling overhead and communication overhead will vary depending on the nature of the application.

A short second experiment was performed to study the overhead incurred by the (de-)serialization code in the current tile implementation. In this experiment, the worst-case execution time of the (de-)serialization functions was replaced with the execution time of the communication assist as presented in [13] and the WCET of the (de-)serialization routine was no longer counted towards the execution time of the processing element. This resulted in, according to SDF^3, an increased throughput for our case-study by up to 300% when actors were mapped to the same resources as in the original experiment. This suggests that the use of a CA will greatly improve the usability of the MAMPS platform, but this result could not be verified on hardware because there is currently no support for tiles using a CA.

7 Conclusions

In this paper, we present an automated design flow that is capable of generating an implementation of a given application on a MPSoC and correctly predicting the worst-case performance of the generated implementation. The design flow provides a method for automatically instantiating different architectures using a template based architecture model. This template based architecture is easy to extend and allows the automated selection of the correct implementation when heterogeneous systems are designed. This allows the designers to perform a very fast design space exploration for real-time embedded systems. Together with the publication of this paper the whole flow will be made publicly available to the research community at `http://www.es.ele.tue.nl/mamps`. For future work we would like to offer an improved automated design space exploration and more variation in the architecture template. Adding a predictable arbiter could enable multiple tiles in accessing peripherals while keeping a predictable system. Finally, we plan to add the communication assist presented in [13].

References

1 Benny Akesson *et al.*: *Predator: a predictable SDRAM memory controller*; in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*; p. 251–256; New York, NY, USA; 2007; ACM.

2 Marc Geilen and Twan Basten: *Requirements on the Execution of Kahn Process Networks*; in *Programming Languages and Systems*; vol. 2618 of *Lecture Notes in Computer Science*; p. 319–334; Springer Berlin / Heidelberg; 2003.

3 Amir Hossein Ghamarian *et al.*: *Throughput Analysis of Synchronous Data Flow Graphs*; in *Proceedings of International Conference on Application of Concurrency to System Design*; p. 25–36; Los Alamitos, CA, USA; 2006; IEEE Computer Society.

4 Stefan Valentin Gheorghita *et al.*: *Automatic scenario detection for improved WCET estimation*; in *Proceedings of the 42nd annual Design Automation Conference*; p. 101–104; New York, NY, USA; 2005; ACM.

5 Jan Gustafsson: *The Worst Case Execution Time Tool Challenge 2006*; in *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*; p. 233 –240; nov. 2006.

6 Soonhoi Ha *et al.*: *Hardware-Software Codesign of Multimedia Embedded Systems: the PeaCE*; in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*; p. 207–214; 2006.

7 Niklas Holsti *et al.*: *WCET 2008 – Report from the Tool Challenge 2008 – 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*; in *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*; Dagstuhl, Germany; 2008; Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

8 Akash Kumar *et al.*: *Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA*; *ACM Transactions on Design Automation of Electronic Systems*; 13(3), p. 1–27; 2008.

9 Edward A. Lee and D.G. Messerschmitt: *Synchronous data flow*; *Proceedings of the IEEE*; 75(9), p. 1235 – 1245; sep. 1987.

10 Hristo Nikolov *et al.*: *Multi-processor system design with ESPAM*; in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*; p. 211–216; oct. 2006.

11 Hristo Nikolov *et al.*: *Systematic and Automated Multiprocessor System Design, Programming, and Implementation*; *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*; 27(3), p. 542–555; mar. 2008.

12 Andy Pimentel *et al.*: *Tool Integration and Interoperability Challenges of a System-Level Design Flow*; in *Embedded Computer Systems: Architectures, Modeling, and Simulation*; vol. 5114 of *Lecture Notes in Computer Science*; p. 167–176; Springer Berlin / Heidelberg; 2008.

13 Ashan Shabbir *et al.*: *CA-MPSoC: An automated design flow for predictable multi-processor architectures for multiple applications*; *Journal of Systems Architecture*; 56(7), p. 265–277; 2010; Special Issue on HW/SW Co-Design: Systems and Networks on Chip.

14 Sander Stuijk: *Predictable Mapping of Streaming Applications on Multiprocessors*; PhD Thesis; Eindhoven University of Technology; 2007.

15 Xilinx website: *Fast Simplex Link overview*; apr. 2010; `http://www.xilinx.com/products/ipcenter/FSL.htm`.

16 Reinhard Wilhelm *et al.*: *The worst-case execution-time problem—overview of methods and survey of tools*; *ACM Transactions on Embedded Computer Systems*; 7(3), p. 1–53; 2008.

17 Zhiyao Joseph Yang *et al.*: *An Area-efficient Dynamically Reconfigurable Spatial Division Multiplexing Network-on-Chip with Static Throughput Guarantee*; in *Proceedings of International Conference on Field Programmable Technology*; p. unknown; Beijing, China; dec. 2010; IEEE; Paper accepted for publication.