CA-MPSoC: An Automated Design Flow for Predictable Multi-processor Architectures for Multiple Applications

A. Shabbir[a,1], A. Kumar[a,b,1], S. Stuijk[a,1], B. Mesman[a,1], H. Corporaal[a,1]

[a]*Eindhoven University of Technology Eindhoven, The Netherlands*
[b]*National University of Singapore, Singapore*

Abstract

Future embedded systems demand multi-processor 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 semi-automated, time consuming and error-prone.

In this paper, we present a fully automated design flow to generate *communication assist (CA)* based multi-processor systems (CA-MPSoC). A worst-case performance model of our CA is proposed so that the performance of the CA-based platform can be analyzed before its implementation. The design flow provides performance estimates and timing guarantees for both hard real-time and soft real-time applications, provided the task to processor mappings are given by the user. The flow automatically generates a super-set hardware that can be used in all use-cases of the applications. The software for each of these use-cases is also generated including the configuration of communication architecture and interfacing with application tasks.

CA-MPSoC has been implemented on Xilinx FPGAs for evaluation. Further, it is made available on-line for the benefit of the research community and in this paper, it is used for performance analysis of two real life applications, Sobel and JPEG encoder executing concurrently. The CA-based platform generated by our design flow records a maximum error of 3.4% between analyzed and measured periods. Our tool can also merge use-cases to generate a superset hardware which accelerates the evaluation of these use-cases. In a case study with 6 applications, the use-case merging results in a speed up of 18 when compared to the case where each use-case is evaluated individually.

Key words: Multi-processor, Multiple Applications, Performance Analysis, Automated Design Flow, Communication Assist

1. Introduction

Modern multimedia embedded systems have to support a large number of independent applications. In the area of portable consumer systems, such as mobile phones, the number of applications doubles roughly every two years and the introduction of new technology solutions is increasingly driven by applications [18]. Tile-based multi-processor platforms [47, 23, 24, 12, 39] are increasingly being used in modern embedded systems to meet tight timing and high performance requirements of these large number of applications and their *use-cases*. A use-case is a combination of concurrently executing applications. The number of such potential use-cases is exponential in the number of applications that are present in the system.

In general, mapping applications onto tile-based platforms is considered difficult. However, streaming applications can be described in a data flow like manner and the computational kernels of this flow can be easily mapped to suitable processing elements. In essence, these systems trade architectural complexity for communications, spreading work across a number of sparsely connected small tiles rather than among richly connected functional units of a monolithic, wide core. In order to make use of tile-based platforms easier, inter-tile communication for these architectures should be predictable, fast and easy to program.

In [9], a multi-processor platform is introduced that decouples the computation and communication of applications through a hardware *communication assist* (CA). This decoupling off-loads the communication load from the processor, thereby improving the performance significantly. Further, this makes it easier to provide tight timing guarantees on the computation and communication tasks that are performed by the applications running on the platform. Several CA architectures [33, 4, 35, 37] have been presented in the literature. However, it is very time consuming to map applications on these platforms due to the unavailability of *platform generation tools*. Furthermore, it is very difficult to *program* them as the user has to configure the communication infrastructure in addition to the application functionality.

Manual design efforts are error prone and consume a lot of time. To worsen matters, most of these devices have very short product life-cycle; shorter time-to-market for these systems poses a challenge for the designers. The designers have to

Email addresses: `a.shabbir@tue.nl` (A. Shabbir), `akash@nus.edu.sg` (A. Kumar), `s.stuijk@tue.nl` (S. Stuijk), `b.mesman@tue.nl` (B. Mesman), `h.corporaal@tue.nl` (H. Corporaal)

verify each use-case. For example, Bluetooth 2.5 has to meet its specification during each combination of applications. It should perform while receiving a call or sending text messages or even taking a picture. So there is a need for automated tools which can reduce the design generation and verification time.

There are some multi-processor design tools [20, 31, 37, 44], but most of them lack support for multiple applications let alone multiple use-cases, and require manual steps. There is a tool described in [26] that supports platform generation for multiple applications and their use-cases but it does not support *CA-based platforms*. Automated platform generation reduces errors in the design and thus saves time for design iterations.

Automatic platform generation is very helpful for the designers but often they are also interested in knowing about the expected performance of the applications before the actual synthesis of the platform. This allows the designers to choose the design which meets their requirements. There are some *performance evaluation* tools [22, 29, 46, 48], but most of them are for single application. There is a tool [28] for performance analysis for multiple applications but it does not take into account the *communication architecture details*.

In this paper, we present a design flow (CA-MPSoC) that takes models of multiple applications and their task to processor mappings, as input and gives expected performance of the applications. Synchronous Data Flow graphs [30] (SDFGs) are used to model the applications. These application models are refined with the details of the communication architecture and actor-to-processor mappings. The refined graphs are used to predict the performance of multiple applications. If the designer is satisfied with the performance estimates, he/she can generate CA-based platform by using our CA-MPSoC. As far as we know, this is the first design flow which can generate a CA-based platform. Following are the *key contributions* of the paper.

Performance analysis: The flow provides the expected performance of applications on the platform, given the fact that the mappings of the tasks on the processors is already provided. The applications are presented as SDFGs and architecture details are added to these graphs. An SDF model of CA has been introduced and it is used to generate *architecture aware* SDFGs. The tool provides both the worst case and average case performance results from these graphs. Worst case results can be used for hard real-time applications whereas the average case can be used for soft real-time applications.

Automatic CA-based multi-processor generation: An automated design flow that generates multi-processor systems, directly from the architecture aware application graphs. The flow also generates the communication infrastructure so that the designer does not worry about it. It generates a super-set hardware which can be used for all the use-cases. The software for each use-case is generated individually. This reduces the verification time of all the use-cases of the applications. The designer can verify that their applications will meet the required performance in all possible combinations of applications.

SDF Task Interface: Another contribution of this work is definition of an interface for the tasks such that the semantics of SDF behaviour are maintained during execution. So when an application specification includes high-level language code corresponding to tasks in the application, the source code is automatically added to the desired processor.

Software generation: The software for all the processors is automatically generated in the flow. Further, the required communication APIs are also generated. This includes configuration of communication channels, setting up connections, and management of memory used for communication. The programmer does not bother about these configurations and can concentrate on the functionality of the applications.

The above contributions are essential to further research in design automation community since the embedded devices are increasingly becoming multi-featured. Our flow allows designers to evaluate the performance of applications on the architecture before actually synthesizing it. It also allows the designers to generate the platform for either hard real-time or soft real-time systems with given sets of actor to processor mappings. CA-MPSoC is evaluated on two real life applications Sobel and JPEG Encoder. The maximum error between estimated and measured periods of these applications is about 3.4% for soft real-time analysis. Furthermore, platform generation for multiple uses-cases is evaluated with a mobile phone case study consisting of 6 applications. The merging of use-cases gives a platform which supports all the use-cases. This merging results in a speed up of 18 as compared to the case where the use-cases are evaluated individually. The tool is made available on line [7] for the benefit of the research community.

The rest of the paper is organized as follows. Section 2 reviews the related work for existing CA architectures, performance analysis and automatic platform generation tool flows. In Section 3 we describe our architecture template. Section 4 introduces SDFGs. Section 5 presents SDF model of our CA. In Section 6, we show how the SDF model of CA can be incorporated in the application model and how performance of applications can be predicted. Section 7 gives details of the steps performed in our design flow to generate the platform. Section 8 describes details of tool implementation. Section 9 presents results of the experiments performed to evaluate our design flow. Section 10 concludes the paper and gives directions for future work.

2. Related Work

2.1. Communication Assist

The communication controller presented in [37] implements FIFO based communication between tasks. Writes to the FIFOs are always local to a processor whereas reads are always remote (from the FIFO memory of a producer). The programming model is based on Kahn Process Network [21] (KPN).

Due to FIFO based communication, out-of-order access, rereading, and skipping is only possible after storing the data locally in the consuming task. In our CA-based platform, all the reads/writes to the memory are local to the producer/consumer resulting in saving of the memory space.

In [32], the authors have presented SystemC model of a CA, but there are some key differences with our CA. They propose separate communication and computation memories whereas in our case, the data memory is also used as communication memory. In [13], the authors have presented a synchronization scheme for embedded shared memory systems. They propose channel controllers for synchronization of data between tasks. They have channel controllers per channel; our implementation has one controller for all the channels, resulting in area efficient implementation. Authors in [6] describe communication between Nested Loop Programs (NLP) in multi-processor systems. The algorithm is implemented in software and can handle out-of-order access to the buffer. Both producer and consumer have their respective write and read windows for mutually exclusive access. However, the algorithm is limited to single assignment codes. Our CA does not impose such restrictions.

A KPN is derived from NLP in [49]. In KPN communication between the tasks is arranged via FIFO buffers. When the consuming task has to read a location multiple times, the consumer stores the array in an additional buffer. Instead of FIFO buffers, we use circular buffers and also there is no need to copy values in an additional buffer. The work by [17] is quite similar to [49] and uses a read and write window.

CELL BBE [15] implements communication between processing elements (SPEs) and the external memory through DMA controllers called Memory Flow controller (MFC). The key difference between MFC and our CA is the fact that in MFC the synchronization between the memories has to be performed explicitly by the SPEs. In case of CA the synchronization is taken care of by the CA itself and the processor is freed from the synchronization overhead.

In the KPN model of computation, processes communicate with each other by sending data to each other over edges. A process may write to an edge whenever it wants. When it tries to read from an edge which is empty, it blocks and must wait till the data is available. The amount of data read from an edge may be data-dependent. This allows modeling of any continuous function from the inputs of the KPN to the outputs of the KPN.

It has been proved in literature that it is not possible to analyze properties like the throughput or buffer requirements of a KPN at design time [14]. On the other hand, SDF is a more restrictive model. A task can only execute if it has input data and space available at the output. The size of input and out data is also fixed so throughput analysis and buffer capacity analysis of SDF graphs is possible statically, which makes SDF more attractive than the KPN.

Note that others in fact impose restrictions on the KPN graphs that are accepted by their tools. These constraints turn these graphs into cyclo-static dataflow graphs. Such a cyclo-static dataflow graph can always be coverted into an SDF and mapped using our flow. Hence it may seem that others use a more flexible model, but in fact their restrictions imply that use the same model as we do.

2.2. *Design Flows for Platform Generation*

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 [37]. This uses Kahn process networks (KPNs) [21] for application specification. In our approach, we use SDFGs for application specification instead. Further, our approach supports mapping of multiple applications, while ESPAM is limited to single application. 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 [20], where an exploration framework to build efficient FPGA multi-processors is proposed.

The Compaan/Laura design flow presented in [44] 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 [31]. 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 [27]. However, that flow focuses on generic MPSoCs and not on application-specific architectures. There is also a tool described in [26] that supports platform generation for multiple use-cases but it does not support *CA-based platforms*.

Xilinx provides a tool-chain as well to generate designs with multiple processors and peripherals [50]. 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.

Finally, none of the above flows support a CA-based platform. In fact our flow is the first to generate CA-based multiprocessor platforms. Communication plays an important role in the parallelization of applications. The communication to computation ratio determines the justification of splitting task between the processors. Our CA in turn exposes more parallelism in the applications.

In [8], the authors present a design flow that generates a multi-core system for multimedia applications. Their work is quite similar to ours. However, there are some key differences. Firstly they use mesh network for interconnection whereas we use point-to-point networks. Secondly, they use profiling to dimension their system. We, on the other hand use static analysis techniques. Profiling based techniques are significantly slower than analysis based techniques. Also their synthesis flow generates platforms for average case performance whereas our flow can generate platforms for both worst case and average case performance. Lastly, our flow supports multiple applications

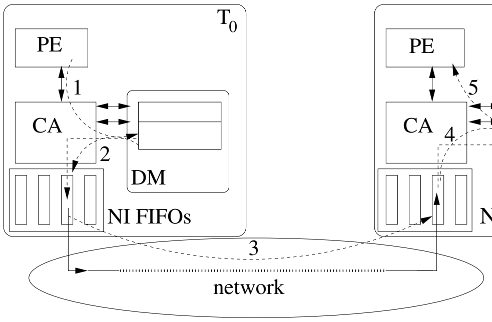


Figure 1: Proposed CA-based platform.

concurrently executing on the platform while [8] is for single application.

2.3. Performance Analysis

In [34], the authors propose to analyze the performance of a single application modeled as an SDFG by decomposing it into a homogeneous SDF graph (HSDFG) [43]. The throughput is calculated based on analysis of each cycle in the resulting HSDFG [10]. However, this can result in an exponential number of vertices [38]. Thus, algorithms that have a polynomial complexity for HSDFGs have an exponential complexity for SDFGs. This approach is not practical for multiple applications.

For multiple applications, an approach that models resource contention by computing worst-case-response-time (WCRT) for TDMA scheduling (requires preemption) has been analyzed in [3]. A similar worst-case analysis approach for round-robin is presented in [16], which also considers non-preemptive systems, but suffers from the same problem of lack of scalability. Real-time calculus has also been used to provide worst-case bounds for multiple applications [22, 48, 29]. The analysis is very intensive and requires a very large design-time effort. On the other the worst-case-waiting-time analysis used in our tool is very fast and simple.

A common way to use probabilities for modeling dynamism in application is using stochastic task execution times [1, 41, 42]. The probabilistic approach [25] used by us uses probabilities to model the resource contention and provides estimates for the throughput of applications. This approach is orthogonal to the approach of using stochastic task execution times. To the best of our knowledge, there is no efficient approach of analyzing multiple applications on a non-preemptive heterogeneous multi-processor platform. A technique has been presented in [28] to also model and analyze contention, but the approach used in this paper is much better. The technique in [28] looks at all possible combinations of actors blocking another actor. Since the number of combinations is exponential in the number of actors mapped on a resource, the analysis has an exponential complexity. The approach used in this paper has linear complexity in number of actors.

3. Architecture Template

The architecture template used in our platform is depicted in Figure 1. It consists of a processing element (PE), a communication assist (CA), Data memory (DM) and Network interface FIFOs (NI FIFO). The CA transfers data between the DM and the NI FIFO. The NI FIFOs are connected through a partial point-to-point network. The structure of the network is out of the scope of this paper.

Scalability of partial point-to-point networks has been an issue as they require storage to deal with bursts. FSL buses from Xilinx is one example. However, the point-to-point networks used in our template do not require storage. This means that cost of a connection is not very high. The CAs can transfer the data directly from the data memory of sending tile to the data memory of the receiving tile, i.e. they do not require storage in the point-to-point network itself.

3.1. Processing Element

The processing elements used in our template are simple RISC based processors. RISC processors are the processing element of choice for tile-based platforms [47]. No caches are attached to the processor to have predictable execution trace. The PE has local instruction and data memories. The instruction memory is connected to the PE through a bus whereas the access to the data memory is through the communication assist. Note that we chose microblaze processors from Xilinx whereas there is work [2] where picoblaze processors are used. Our synthesis flow is not restricted to any one processor type so choice of processor is not important.

The PE is non-preemptive and can execute only single thread. This simplifies the architecture of the PE. Preemption requires extra hardware and is costly in terms of area. Furthermore, non-preemptive scheduling algorithms are easier to implement as compared to their preemptive counter parts and have dramatically lower overhead at runtime [19]. In high performance embedded processors (like SPEs in Cell Broad Band Engine and graphics processors), non-preemptive systems are preferred over preemptive systems.

3.2. Memories

We use a single port instruction memory, which is directly connected to the PE. The data memory (DM) used in our template is a dual ported memory as depicted in Figure 1. The CA has exclusive access to one port of this memory. The second port is connected to the PE through the CA. The choice of dual ported memory may seem expensive, however we use it to make the access of the memory to CA and PE as fast as possible. The other option could be an arbiter to resolve the access between the two but for predictable performance, we preferred dual ported memory over a combination of an arbiter and a single ported memory. Single ported memory can introduce stall cycles for the processor which inturn makes the execution time of the task executing on the processor, unpredictable. Further, it is very difficult to model an unpredictable arbiter so we decided to use dual ported DM. Next subsection will clarify this configuration.

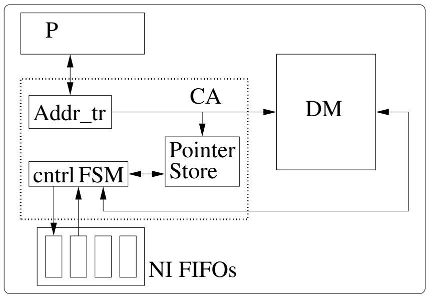


Figure 2: CA architecture.

3.3. Communication Assist

Figure 2 shows the global view of our CA (more details about the architecture can be found in [40]). It performs the following basic functions

1. It configures NI FIFO channels and their corresponding buffers in DM.
2. It accepts data transfer requests from the attached PE and splits them into local memory requests and remote requests (to other tiles). The address translation unit "Addr_tr" shown in Figure 2 performs this task.
3. Local memory requests are simply bypassed to the data memory.
4. Remote memory requests are handled through a round robin arbiter. Every two cycles, a 32 bit word is transferred from the buffer in the memory to NI FIFO channels and vice versa.
5. The buffers implemented in the memory are circular buffers. The pointers needed for circular buffer management are updated and stored in the CA. The number of NI FIFO channels can be greater than or equal to number of buffers in the data memory.

Our communication assist acts as an interface that provides a link between the NoC and the sub systems (PE and memory). It also acts as memory management unit that helps a processor keep track of its data structures. As a result, it decouples communication from computation and relieves the processor from data transfer functions. Our programmable CA uses a shared data and buffer memory. This leads to lower memory requirement for the overall system and to a lower communication latency.

Figure 1 shows two CA-based multi-processor tiles and demonstrates the steps involved during data transactions between the tiles. Assume tile T_0 is executing a producer task and tile T_1 is executing a consumer task. The primitives used for communication are known as the C-HEAP [36] protocol. The producer task executing on tile T_0 requests space. The CA returns a pointer to the buffer in the memory (step 1 in Figure 1). The PE processes the data as local memory access. It then requests the CA that it wants to release the space. The CA transfers the data to the designated NI FIFO (step2). The data is transported through the network (step 3). The CA of the consumer task executing in tile T_1 receives the data and places that in the memory (step 4). The consumer task requests the CA about the availability of the data. The CA sends the pointer to this data and the PE can access it like a local memory request (step 4). The consumer task processes the data and releases the space so that the CA can use this space for future data receptions (step 5).

Figure 2 depicts the hardware components of our CA. The pointers used for circular buffer management are stored in a pointer store unit "Pointer Store". Every clock cycle, the CA checks whether there is data to be transferred between the DM and the NI FIFOs. The monitoring of the NI FIFOs is round robin, which makes the architecture predictable. This predictability allows us to give tight bounds on the reported performance of the platform.

Before we can demonstrate how the communication between the tiles and the timing behaviour of task execution can be analyzed in terms of timing, first we need to introduce SDFGs in the next section.

4. SDF Graphs

Synchronous data flow graphs are often used for modeling modern DSP applications [43] and for designing concurrent multimedia applications implemented on multi-processor 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 [45].

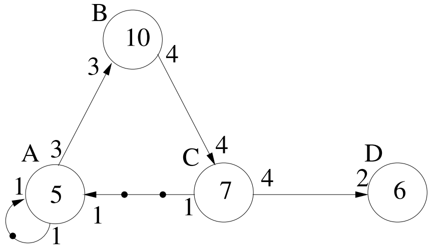


Figure 3: Example of an SDF graph.

Figure 3 shows an example of an SDFG. 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 3. Buffer sizes may be modeled as 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.

One of the most interesting properties of SDFGs relevant to this paper is *throughput*. Throughput is defined as the inverse of the long term *period*, i.e. the average time needed for one iteration of the application. An *iteration* is defined as the minimum non-zero execution such that the original state of the graph is obtained. This is the performance parameter we use in this paper.

One of the methods to find the throughput of an SDFG is to convert it into HSDF graph and then find the throughput of the resulting graph. An HSDF graph is a special kind of SDFG in which execution of an actor results in consumption of one token from every incoming edge of the actor and production of one token on every outgoing edge of the actor. The throughput is calculated based on the analysis of each cycle in the resulting HSDFG. The maximum period of these cycles is the inverse of throughput and is called MCM, given by

$$MCM(G) = max_{c \in G} CM(c) \tag{1}$$

$$CM(c) = \sum_{v\ on\ c} WCET(v)/tokens(c) \tag{2}$$

here WCET is worst case execution time of each actor v, c is one of the cycles in the graph and *tokens* are the number of initial tokens in the cycle.

The CA is also modeled as an SDF actor so that methods like MCM can be used to measure the performance of these combined graphs of applications and architectural components. In the next section, we present SDF model of our CA.

5. SDF Model of CA

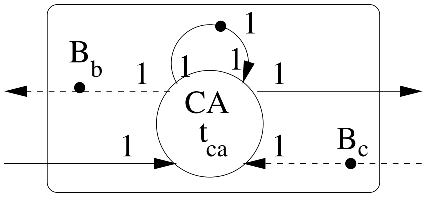


Figure 4: SDF model of CA.

A predictable system allows the derivation of a conservative lower bound on the throughput and a conservative upper bound on the end-to-end latency. To achieve this goal, accurate analytical models of applications and architectural components are necessary so that performance estimates can be made before synthesis of the platform. In multimedia applications, tasks can be modeled as actors of SDFG. Tasks like Descrete cosine transform (DCT), Color Conversion (CC) are some of the examples. The synchronization between these actors takes place on token granularity. A token can be a pixel, a macro-block or a frame.

The application SDF model can be refined to include the mapping decisions, buffer sizes and the timing impact of architectural components. This results into a combined SDFG of the application and the architecture with a predictable behaviour. We call it an *architecture aware* SDFG. Our CA can be modeled as an actor with a self edge (see Figure 4). The self edge is given one initial token such that the next execution of the actor can not start before the previous execution has finished. As described earlier, the CA polls the NI FIFO channels in a round robin fashion. Every channel requires two cycles. During the first cycle the CA checks whether there is a word to be transferred from the output buffer to the channel or from channel to the input buffer. The second cycle is required for the transfer. As the number of channels per CA increases, the response time of the CA gets larger. The execution time of CA actor t_{ca} can be calculated using equation 3:

$$t_{ca} = 2 \times NC \tag{3}$$

where NC are the number of NI FIFO channels the CA has to manage. Each channel takes 2 cycles so we multiply it with number of channels.

In CA-based platform, the CA lies between NI FIFO channels and data memory of the processor. The CA transfers data between NI FIFO channels and buffers in the memory. Similarly, in SDF model, each CA actor is connected with task actor while the other side of CA actor is connected with the NI FIFO channels. The depth of the NI FIFOs is modeled with the initial tokens B_c as shown in Figure 4. The rate at this edge is one word as each execution of the CA actor transfers one word from buffer to the NI FIFO or vice verse. Note that the direction of this edge will reverse in case of an input buffer. Similarly B_b models the buffer space claimed by the processor for reading or writing. The rate at this edge is also one as one word space is released with each execution of CA.

The application model is transformed into an architecture aware SDF model. The architecture aware SDF model enables us to predict the performance of applications before actually implementing them in hardware. Now that we have the SDF models in place, we demonstrate the design time timing analysis of these models.

6. Performance Analysis

The transformation of an SDFG into HSDFG can result in an exponential number of vertices in the resultant graph. Each actor in HSDFG consumes and produces one token during each execution. If an actor in SDFG consumes n number of tokens, then the resulting HSDFG will model this with n actors, each consuming one token. Thus algorithms (for finding the throughput) that have polynomial complexity for HSDFGs will have exponential complexity for SDFGs. The situation gets worse when mapping and other architectural details are added to the graphs. When modeling communication, CA actors are added and for modeling resource dependencies extra edges are added. For multiple applications, the graphs become very complex and MCM based methods cannot work. A technique has been proposed in [45] to compute throughput directly on SDFGs. For real life application graphs, this technique is faster than the MCM based methods.

We will use JPEG encoder as a running example throughout this paper, to show the transformation into our architecture

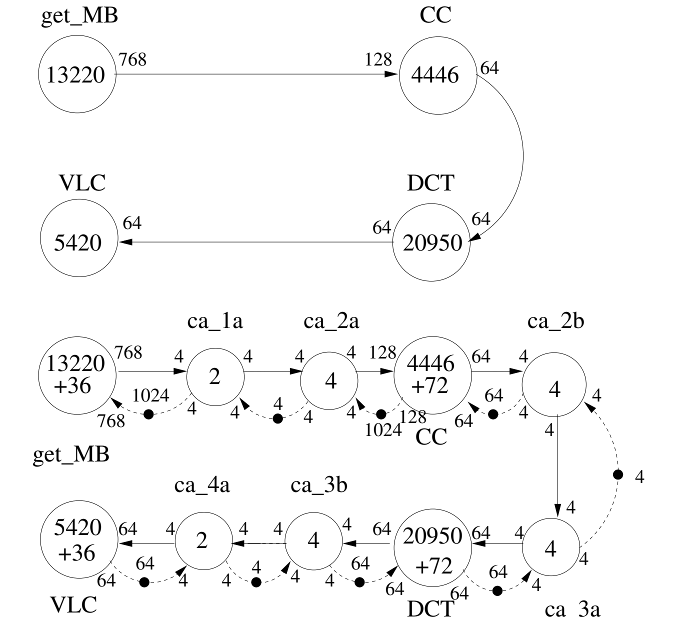


Figure 5: SDF graph of JPEG encoder and its transformation into CA-based platform.

aware SDFG and to show how we estimate the performance of these graphs using the analysis techniques.

The upper part of Figure 5 shows SDF model of JPEG encoder. It is split into four actors. Each actor is mapped on one processor of the platform. The four actors are macro-block sampling (*get_MB*), color conversion (*CC*), discrete cosine transform (*DCT*) and variable length coding (*VLC*). The first actor *get_MB* parses the input BMP file and sends macro-blocks to the *CC* actor. Each macro-block is 16×16 pixels and 3 such macro-blocks are sent to the *CC* (one each for R, G and B pixels). This equates to 768 pixels. The *CC* actor converts the RGB format into 4 luminance Y, and two Cr, Cb chrominance macro-blocks. These 8×8 macro-blocks (384 pixels) are fed to the *DCT* actor which is the most compute intensive task of JPEG encoder. The *DCT* actor sends these 6 macro-blocks one by one (64 pixels each time) to the *VLC* actor where each of these macro-blocks is variable length encoded. The worst-case-execution times of the actors (in number of clock cycles) are obtained through profiling and are also shown in the graph. Note that this graph does not model communication delay and only the execution times of the actors are modeled here.

To analyze this application when mapped on our CA-based platform, the graph is transformed into the one shown on the bottom of Figure 5. Every channel in the upper graph has been mapped to an independent CA actor. The execution time of each CA actor is calculated by equation 3. For example, the CC actor sends 64 pixels to the DCT actor. The CA attached to CC actor has two channels (ca_2a and ca_2b). So the execution time of each CA actor is 4 cycles. Every 4 cycles, 4 pixels (1 word=4 pixels) are transferred as shown in Figure 5.

After this graph transformation, we can use either average case analysis for soft real-time systems or use the worst-case-waiting time analysis for the hard real-time systems.

6.1. Average-case Analysis

In [25], the author presents a technique for performance analysis of multiple applications executing concurrently on a multiprocessor platform. The technique named *Iterative Probabilistic Performance Prediction* (*IP^3*) is particularly suitable for non-preemptive PEs and is based on a probabilistic model of the contention on the shared resources. When actors from different or same applications share a processor, they are executed in an orderly fashion depending upon the scheduling policy. Each actor has to wait for its turn before it can execute. The time spent by an actor in contention is added to its execution time, and the total gives its response time:

$$t_{resp} = t_{exec} + t_{wait} \tag{4}$$

The t_{wait} is the time that is spent in contention when waiting for a processor resource to become free. (This time may be different for different arrivals of a repetitive task.) The response time, t_{resp} indicates how long it takes to process an actor after it arrives at a PE. When there is no contention, the response time is simply equal to the execution time. The authors of [25] use probabilities to predict the waiting times of the actors on the PEs and provide quite accurate results. However, use of probabilistic model means that they can not give guarantees on their performance estimates.

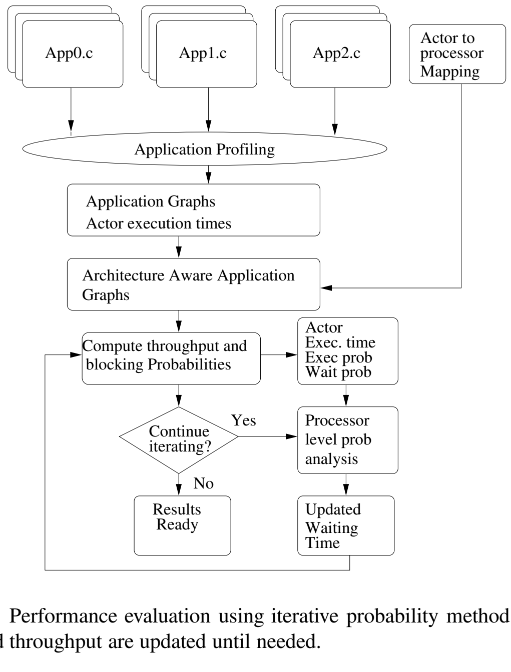


Figure 6: Performance evaluation using iterative probability method. Waiting times and throughput are updated until needed.

We use the IP^3 to predict the performance of applications mapped on our CA-based platform. Figure 6 shows our performance evaluation methodology. Application code is profiled and xml files containing the actor execution times are obtained. These files are updated with mapping information and architecture aware SDFGs are obtained. Additional actors are added to model the communication. The architecture aware SDF model is input to the tool. The CA is modeled as an independent actor so the CA actors are not shared in the IP^3. The execution times of the actors in the applications are replaced with the response times calculated with the iterative probabilistic prediction. These application models are then fed to *SDF*^{3} [46] tool to compute the throughput of the individual graph. The updated actor execution times, execution probabilities and waiting probabilities are used to find the new processor level probabilities. Waiting times are updated and the loop continues until the number of iterations are finished.

As stated earlier, this technique is based on probabilistic waiting times so it can not provide guarantees on its timing results. It is however quite fast and can also be used for run-time analysis.

6.2. Worst-case Analysis

Besides the iterative technique, the worst-case-waiting-time approach [16] is also used to give guarantees on the performance. The worst-case-waiting-times for non-preemptive systems for FCFS as mentioned in [16] are computed by using the following formula

$$t_{wait} = \sum_{i=1}^{n} t_{exec(a_i)} \tag{5}$$

where actors a_i for $i = 1, 2, 3, ...n$ are mapped on the same resource (i.e processor). The waiting times are added to the execution times of the application actors in the architecture aware application graphs. The execution times of the CA actors are left unchanged because the CA actors are not shared. These updated architecture aware graphs are then used to find the throughput using *SDF*^{3}. It is intuitive to judge that this method will give pessimistic results for large number of applications. However, the results can be used for hard real-time applications.

7. Design Flow

```
<sdf name="jpeg" type="G">
  <actor name="CC" type="A0">
    <port name="in0" type="in" rate="128" datatype="char"/>
    <port name="out0" type="out" rate="64" datatype="char"/>
    <executionTime time="4446"/>
    <processor type="proc_0" default="true">
    <functionName funcname="CC"/>
  </actor>
  <actor name="DCT" type="A1">
   <port name="in0" type="in" rate="64" datatype="char"/>
   <port name="out0" type="out" rate="64"datatype="short"/>
   <executionTime time="20950"/>
   <processor type="proc_1" default="true">
   <functionName funcname="DCT"/>
  </actor>
    ...
<channel name="ch0" srcActor="CC" srcPort="out0"
dstActor="DCT" dstPort="in0"/>
    ...
```

Figure 8: Snippet of JPEG application specification.

Once the user is satisfied with the performance analysis results, he/she can generate the complete CA-based platform using our design flow. We present CA-MPSoC, a design flow that takes in application(s) specifications and generates the entire CA-based MPSoC, specific to the input application(s) together with corresponding software projects for automated synthesis. This allows the design to be directly implemented on the target architecture. Figure 7 depicts our system design methodology. The application-descriptions are specified in the form of SDFGs, which are used to generate the hardware topology. Figure 8 shows an example of application description. It forms an important part of the flow. While the specification shown in Figure 8 is obtained through application profiling, it is also possible to use tools to obtain the SDF description for an application from its code directly. Compaan [44] is one such example that converts sequential description of an application into concurrent tasks. These can be then converted into SDFGs easily.

The application-descriptions, mapping information (actor-to-processor) and source code of each application are input to our tool. The source code is already partitioned and each actor is in the form of a function call with arguments being the input and output to the actor.

7.1. H/W Generation

During hardware generation, the IP cores of the processor, CA, and memories are connected according to the mapping information. A CA is connected with each processor to take care of the communication between the processors. The number of NI FIFO channels and the number of buffers (the CA has to manage) are also generated according to edges in the architecture aware SDF graphs.

As the generated hardware supports multiple use-cases, so we employ the use-case merging technique [26] and modify certain parts to incorporate CA buffers. Each use-case requires a certain hardware topology to be generated. In addition to that, software is generated for each processor. Figure 9 shows an example of 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 super-set of all the required resources such that all the use-cases can be supported. The reason to use a super-set hardware is the fact that while multiple applications are active concurrently in a given use-case, different use-cases are active exclusively.

The algorithm to obtain the minimal hardware to support all use-cases is described in Algorithm 1. 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 and CA buffers needed is updated with a max operation (line 10 and line 11 in Algorithm 1), the number of CA channels is added for each application (indicated by line 13 in Algorithm 1). The total CA channel requirement of each application is computed by iterating over all the buffers and adding a unique edge in the communication matrix for them. The communication matrix for the respective use-cases is also shown in Figure 9.

While there are in total three CA channels between CA 0 and CA 1, only two are used (at most) at the same time. Therefore, in the final design only two CA channels are produced

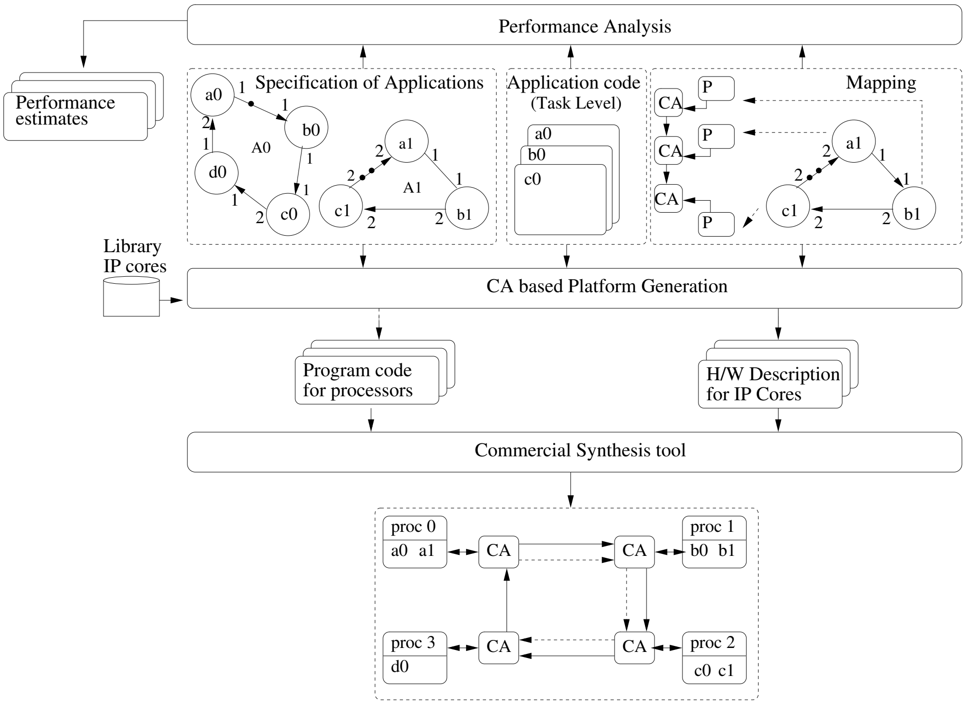


Figure 7: Design flow.

between them. The number of CA buffers required are maximum needed for all the use-cases. For example, CA 2 requires 2 buffers for use-case A and one in use-case B however, in the super-set hardware two buffers are reserved for CA 2. Note that the CA can use the same buffer as input or output. The configuration of CA binds a buffer in the memory with a NI FIFO channel. There are limits to the number of use-cases that can be mapped to hardware and to avoid these limits certain heuristics have been proposed in [26].

7.2. S/W Generation

```
// Functional definition of an SDF-actor
void <functionName>(datatype *in0, datatype *in1,
                    ....., datatype *inN,datatype *out0,
                    datatype *out1,....., datatype *outM){
...
...
}
```

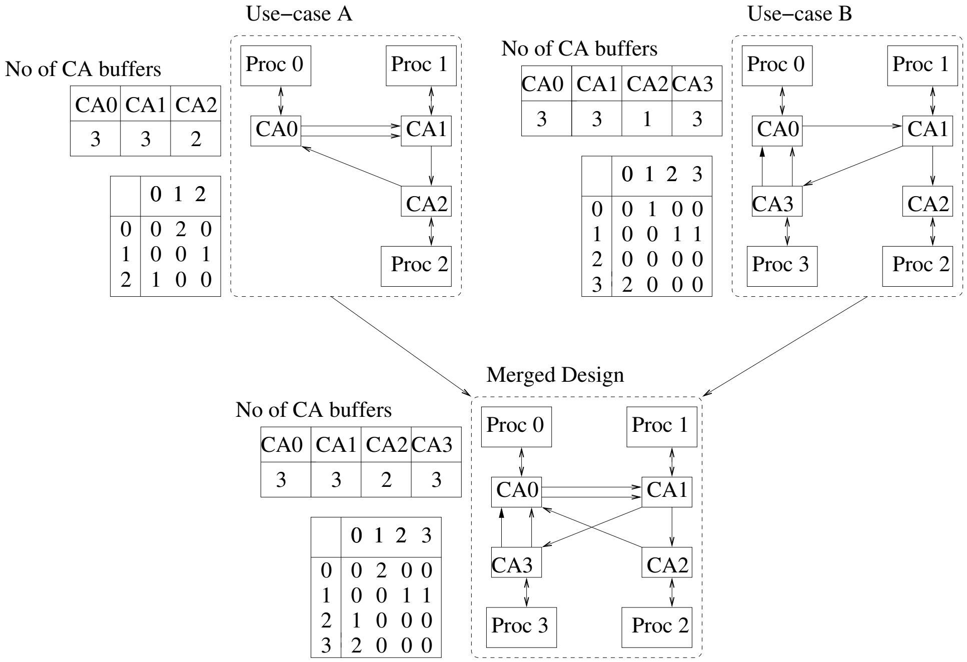
Figure 10: The interface for specifying functional description of SDF-actors.

Software generation includes configuration of buffers between the actors, data type declarations of the ports of the actors and code needed for SDF actor execution. The software project for each core is produced and the task files are copied into the project folder. The *xml* file also specifies the processor on which the actor has been mapped.

If an application specification also includes high-level language code corresponding to actors in the application, this source code can be automatically added to to the desired processor. To realize this, we have defined an interface such that the SDF behaviour is maintained during execution. The number

Algorithm 1 Generate communication matrix for CA channels and number of CA buffers.

```
{// Let $X_{ij}$ denote the number of CA channels needed for processor $P_i$ to $P_j$ overall}
$X_{ij} = 0$ {//Initialize the communication matrix to 0}
$N_{proc} = 0$ {//Initialize number of processors 0}
$N_{ca-buffers} = 0$ {//Initialize number of CA buffers 0}
for all Use-cases $U_k$ do
    $Y_{ij}$ {//$Y_{ij}$ stores the number of CA channels needed for $U_k$ }
    $N_{proc,UseCase} = 0$ {//Initialize processor count for use-case to 0}
    $N_{ca-buffers,UseCase} = 0$ {//Initialize CA buffers for use-case to 0}
    for all Applications $A_l$ do
        $N_{proc,UseCase} = \max(N_{proc,UseCase}, N_{proc,A_l})$ {//Update processor count for $U_k$}
        $N_{ca-buffers,UseCase} = \max(N_{ca-buffers,UseCase}, N_{ca-buffers,A_l})${//Update CA buffers count for $U_k$}
        for all Channels c in $A_l$ do
            $Y_{c_{src}c_{dst}} = Y_{c_{src}c_{dst}} + 1$ {//increment CA channel count}
        end for
    end for
    $N_{proc} = \max(N_{proc}, N_{proc,UseCase}$ {//Update overall processor count}
    $N_{ca-buffers} = \max(N_{ca-buffers}, N_{,UseCase}$ {//Update overall CA buffer count}
    for all i and j do
        $X_{ij} = \max(X_{ij}, Y_{ij})$
    end for
end for
{//$N_{proc}$ is now the total number of processors needed}
{//$X_{ij}$ is now the total number of CA channels needed}
{//$N_{ca-buffers}$ is now the total number of CA buffers needed}
```



No of CA buffers

CA0	CA1	CA2
3	3	2

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

No of CA buffers

CA0	CA1	CA2	CA3
3	3	1	3

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

No of CA buffers

CA0	CA1	CA2	CA3
3	3	2	3

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

Figure 9: An example of showing how the combined hardware for different use-cases is generated. The corresponding communication matrix and no. of buffers are also shown for each hardware design.

of input parameters of an actor function is equal to the number of incoming edges and the number of output parameters is equal to the number of output edges. The interface is shown in Figure 10. The array $*in_i$ is for input tokens consumed from *i*-th incoming edge. where the array length is equal to the size of buffer associated with the edge. Similarly, $*out_i$ is an array of output tokens that are written during one execution of the an actor. The application *xml* file indicates the function name that corresponds to application actor.

Figure 8 shows an example for the DCT actor of JPEG encoder application. The function has an input channel from the CC module and the data produced during execution is written to the output channel to VLC module. Therefore the function definition of this actor only has one input and one output parameter as shown in Figure 11.

Figure 11 shows the c-code generated automatically from our tool. Both actors are executing on different processors. The data types specified in the *xml* file are used to determine the buffer space needed for the particular buffer. Buffers are configured for each channel. The size for each buffer inside the data memory is determined by multiplying the data type and rate associate with the port. For example, the size of output buffer in CC task is 64 bytes ($64 \times 1 bytes$). Configuration of buffer also includes the direction of the buffer, the NI_FIFO_ID number and the physical address of the buffer inside the memory.

The *claimwritespace* command looks for available space in the output buffer. Similarly the *claimreadspace* checks whether the required number of tokens are available for processing. The buffers are identified by their *ids*. The reason to check the availability of output space before the input space is because our SDF model of execution is conservative. Both commands are non-blocking. So an actor might not be able to execute if any of its incoming buffers does not have sufficient tokens. The same holds when the output buffers of an actors are full. While this does not cause any problem when only one actor is mapped on the processor, in the case of multiple actors, the other possibly ready actors might not be able to execute while processor sits idle. To avoid this, *claimreadspace and claimwritespace* commands have been implemented as non-blocking so that if any of *claimspace* commands is unsuccessful, the processor is not blocked. Note that the command overhead is fixed and is added to the execution time of the actors. It is implementation dependent and we explain more about it in Section 9.

After the function processing, the *releasewritespace* command indicates the CA to transfer the data to the next actor. The release commands update the read/write buffers so that they can be used for further receive/send operations.

8. Tool Implementation

In this section, we describe the tool we developed based on our flow to target Xilinx FPGA architecture. The processors in the CA-MPSoC are mapped to Microblaze processors [50]. The communication 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 to one as this is the minimum depth available for these buses. As explained earlier, we do not require any storage in the point-to-point networks in our proposed design. However, it is not possible to have FSL links with zero storage so it is an implementation dependent restriction.

Example architecture for the JPEG application platform is shown in Figure 12 according to the specification in Figure 8. This consists of several Microblazes with each actor mapped to

```
-----------------------------------------------------
//Code generated for Color conversion task
-----------------------------------------------------
char* in0;int size_out=rate_out*sizeof(char);
char* out0;int size_in=rate_in*sizeof(char);

Config(buffer_id0,base_addr_out,size_out,out,ni_fifo_id_out);
Config(buffer_id1,base_addr_in,size_in,in,ni_fifo_id_in);

out0=claimwritespace(buffer_id_0,size_out);
in0=claimreadspace(buffer_id_1,size_in);
CC(in0,out0);
releasewritespace(buffer_id0);
releasereadspace(buffer_id1);
---------------------------------------------------------
//Code generated for DCT task
---------------------------------------------------------
char* out0;int size_out=rate_out*sizeof(short);
char* in0;int size_in=rate_in*sizeof(char);

Config(buffer_id0,base_addr_out,size_out,out,ni_fifo_id_out);
Config(buffer_id1,base_addr_in,size_in,in,ni_fifo_id_in);

out0=claimwritespace(buffer_id0,size_out);
in0=claimreadspace(buffer_id1,size_in);
DCT(in0,out0);
releasewritespace(buffer_id0);
releasereadspace(buffer_id1);
```

Figure 11: Snippet of c-code generated from architecture aware SDFG of JPEG encoder.

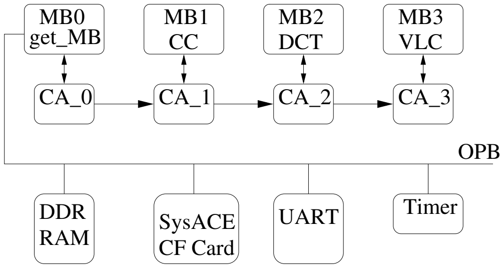


Figure 12: Generated hardware from the example xml file.

a unique processor, with additional peripherals such as Timer, UART, SysACE, and DDR RAM. 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. 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.

9. Experiments and Results

In first part of this section, we evaluate our tool flow with two real life applications. A CA-based platform is generated to run these applications concurrently. The period of these applications is compared with the period computed through analysis techniques described earlier. In the second part, we evaluate our tool with a mobile phone case study consisting of 6 applications. In each use-case we enable a subset of these applications. We also show how our tool generates a super-set hardware that supports large number of use-cases. The software for each use-case is generated at run-time, and enables us to verify these use-cases in very short time.

9.1. Real Life Applications

We have implemented two real life applications (JPEG encoder, Sobel) to evaluate our tool. A CA-based platform consisting of 4 microblaze processors and 4 CAs is generated. Both JPEG encoder and the Sobel models are based on pixel level granularity. Details about the JPEG encoder have been given in previous sections. Now we briefly describe the Sobel filer.

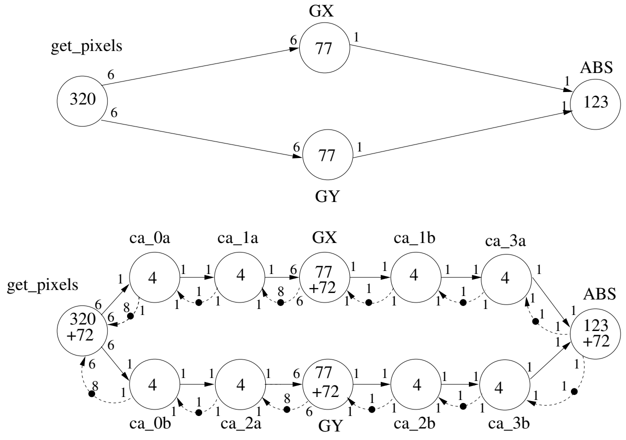


Figure 13: SDFG of Sobel and its transformation into CA-based platform.

Sobel is extensively used in image processing, particularly within edge detection algorithms. Technically it is a discrete differentiation operator and computes the approximation of the gradient of the image. The reference implementation of Sobel is mapped on a 4 microblaze platform. Figure 13 shows the SDF model of Sobel. The first actor (*get_pixels*) opens the input file stored in the CF card and loads it into the data memory. It then forwards 6 pixels each to the connected actors. These actors (*GX*,*GY*) find the gradient of the image in x and y direction respectively. Finally the fourth actor (*ABS*) finds the absolute value of the gradients computed by the preceding actors.

Both applications are concurrently executed on the platform. The application graphs along with mapping decisions, buffer sizes and communication actors for JPEG encoder and Sobel are shown in Figure 5 and Figure 13 respectively. Worst-case-task-execution times (WCET in clock cycles) of the actors are specified inside the circles in the graphs. Self edges are removed for more visibility in the Figures. The response time of each CA is calculated using equation 3.

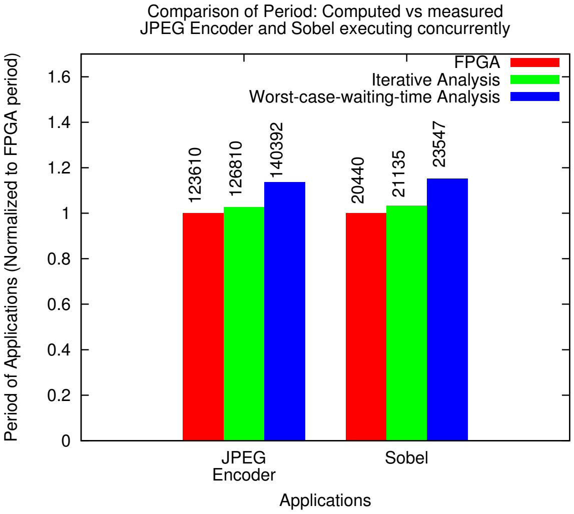


Figure 14: The period of concurrently executing Sobel and JPEG encoder applications as measured and analyzed.

As described earlier, the CA manages the buffer memory for the tasks. The processor asks for pointers to these buffers through commands (*claimreadspace* or *claimwritespace*). It takes certain time for the CA to update the pointers and send them to the processor. This overhead is implementation dependent. A command overhead of 36 cycles has been added to the execution time of the actors. This overhead is multiplied with 2 for CAs having two channels.

The period (in clock cycles) of these applications for one iteration is calculated using the IP^3 and worst-case-waiting-time techniques. In one iteration, the JPEG encoder encodes 3 macro-blocks (R,G,B) and Sobel filters one pixel. The measured period (in clock cycles) from FPGA implementation is also shown in Figure 14 (at 50 MhZ clock frequency, this equates to the encoding of 4 QCIF frames/second). We can increase the clock frequency of microblaze to support low resolution video also.

The predicted periods are normalized with the measured period from the FPGA implementation. Our predicted period using $IP3$ is very close to the measured one whereas worst-case-waiting-time technique is about 15% higher than the measured period. We define *error* as the difference between predicted and measured periods. For JPEG and Sobel applications the maximum error between the corresponding predicted and measured periods for IP^3 is 3.4%. In this particular example, the difference between predicted and measured periods for worst-case-waiting technique is quite low. This means that the same hardware can be used for both average case and worst case performance.

We implemented our CA-based platform on an XUP Virtex II Pro Development Board with an xc2vp30 FPGA. Xilinx EDK 8.2i and ISE 8.2i were used for synthesis and implementation. All tools run on a dual core 2.0 GHz with 1GB of RAM. Table 1 shows the resources claimed by a four channel CA. The CA takes only 5% of the resources of this medium sized FPGA. The synthesized frequency of CA is 108 MHz. The area consumed by the CA-based platform is shown in Table 2. The platform consists of 4 microblaze processors and four CAs. Figures 15, 16 show the output of the JPEG encoder and Sobel filter respectively.

Table 1: FPGA resources for a four channel CA.

	Proposed design	used resources of xc2vp30
No. of Slices	624	5%
No. of Slice flip flops	452	1%
No. of 4 input LUTs	1846	6%

Figure 15: JPEG encoded image.

Figure 16: Output of Sobel Filter.

Table 2: Area overhead for CA-based platform executing Sobel and JPEG encoder applications on Xilinx FPGA (xc2vp30)

Architecture	Slices	Bram	LUT	Used FPGA resources
CA-based	6,779	56	11,212	49%

9.2. *Support for Multiple Use-cases & use-case merging*

In this case study we consider 6 applications - video encoding (H.263) [16], video decoding [45], JPEG decoding [11], mp3 decoding [45], modem [5] and regular call. We first constructed all possible use-cases giving 63 use-cases in total. However, some of these use-cases are not realistic. For example, JPEG decoding is unlikely to run simultaneously with video encoding or decoding, because when a user is recording or viewing video, it is not possible to browse through pictures. Similarly it is also not possible to listen to mp3 songs while talking to some body on phone. This gives us 23 realistic usecases as shown in Table 3. Each active application in a use-case is represent with a "1" at its position.

In this experiment, our tool generates a platform that can support all of these 23 realistic use-cases. The platform consists of 5 microblaze processors and 5 communication assists. The platform occupied 97% of the available FPGA resources.

Our approach is very fast and is further optimized by modifying only the relevant software and keeping the same hardware design for different use-cases. The software synthesis includes configuration of all CA channels, buffer sizes, and incorporation of appropriate task calls. Since software synthesis step takes only about 25 sec in our experiment, the entire experiment for 23 design points takes only about 9 minutes.

Manual design effort will involve separate hardware generation and software configuration for each use-case. In contrast,

Table 3: Realistic use-cases for mobile phone case study.

usecase number	H263 decoder	H263 encoder	JPEG decoder	modem	Phone call	mp3 decoder
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	1	1	0	0	0	0
4	0	0	1	0	0	0
5	0	0	0	1	0	0
6	1	0	0	1	0	0
7	0	1	0	1	0	0
8	1	1	0	1	0	0
9	0	0	1	1	0	0
10	0	0	0	0	1	0
11	1	0	0	0	1	0
12	0	1	0	0	1	0
13	1	1	0	0	1	0
14	0	0	1	0	1	0
15	0	0	0	1	1	0
16	1	0	0	1	1	0
17	0	1	0	1	1	0
18	1	1	0	1	1	0
19	0	0	1	1	1	0
20	0	0	0	0	0	1
21	0	0	1	0	0	1
22	0	0	0	1	0	1
23	0	0	1	1	0	1

our tool takes a mere 100 ms to generate the complete design. The Xilinx tool takes about 36 minute to generate the bit file together with the appropriate instruction and data memories for each core in the design. The time spent on the exploration is an important aspect when estimating the performance of big designs. The 6 application system is also designed by hand to estimate the time gained by using our tool. The hardware and software development took about 4 days in total to obtain an operational system.

Table 4: Time taken for platform generation in the experiment with ten applications.

	Manual Design	Generating Single Design	Complete Experiment
Hardware Generation	2 days	40ms	40ms
Software Generation	2 days	60ms	60ms
Hardware Synthesis	36:00	36:00	36:00
Software Synthesis	0:25	0:25	09:34
Total Time	4 days	36:25	45:34
Iterations	1	1	23
Average Time	4 days	36:25	1:59
Speedup	-	1	18.36

This hardware/software co-design approach results in a speed-up of about 18 when compared to generating a new hardware for each iteration. As the number of design points are increased, the cost of generating the hardware becomes negligible and each iteration takes only about 25 seconds. This study shows the usefulness of our use-case merging approach for problems like DSE for multi-processor systems.

10. Conclusion

In this paper, we present a design flow to generate multiprocessor platforms for multiple applications. We also provide analysis techniques to predict the performance of the applications before the genration of the platform. The design flow can cater for both hard and soft real time applications, given the fact that the mappings of actors to processors are provided by the user. CA-MPSoC allows performance exploration of the applications and their use-cases. It is fully automated and requires minimal manual effort. It also generates the configuration software for the communication infrastructure.

The design flow is evaluated on two real life applications Sobel and JPEG Encoder. The maximum error between estimated and measured periods of these applications is about 3.4%. Furthermore, platform generation for multiple uses-cases is evaluated with 6 applications from a mobile phone case study. The platform generation takes milliseconds in contrast to days needed for manual platform genration. The use-case merging evaluates all the 23 realistic use-cases of the case-study by using a single hardware platform. This results in a speed up of 18 when compared to the case where hardware for each use-case is generated individually and then evaluated. The tool is made available on line [7] for the use by the research community.

One of the limitations of the design flow is that it does not include Network-on-chip (NoC) based designs. It is worth mentioning that our CA can easily be integrated in a NoC. In the future, we intend to include an NoC also in our design flow. We also want to extend the design flow with automated mapping decisions, so that mapping of the actors to the processors can also be optimized.

References

[1] Abeni, L., Buttazzo, G., Superiore, S., Anna, S., 1999. Qos guarantee using probabilistic deadlines. In: In Proceedings of the 11th Euromicro Conference of Real-Time Systems. pp. 242–249.

[2] Astarloa, A., Zuloaga, A., Bidarte, U., Martín, J. L., Lázaro, J., Jimenez, J., 2007. Tornado: A self-reconfiguration control system for core-based multiprocessor csopcs. Journal of Systems Architecture 53 (9), 629–643.

[3] Bekooij, M., Hoes, R., Moreira, O., Poplavko, P., Pastrnak, M., Mesman, B., Mol, J. D., Stuijk, S., Gheorghita, V., van Meerbergen, J., 2005. Dataflow analysis for real-time embedded multiprocessor system design. Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, pp. 91–108.

[4] Bekooij, M., Moreira, O., Poplavko, P., Mesman, B., Pastrnak, M., van Meerbergen, J., 2004. Predictable embedded multiprocessor system design. In: Proc. of SCOPES, LNCS. Vol. 3199. Springer, pp. 77–91.

[5] Bhattacharyya, S. S., Murthy, P. K., Lee, E. A., 1999. Synthesis of embedded software from synchronous dataflow specifications. J. VLSI Signal Process. Syst. 21 (2), 151–166.

[6] Bijlsma, T., Bekooij, M., jansen, P., Smit, G., 2008. Communication between nested loop programs via cicular buffers in an embedded multiprocessor system. In: Proc. of 11th SCOPES. Vol. 296. pp. 33–42.

[7] CA-MPSoC, 2009. Please email at a.kumar@tue.nl for username and password.
URL http://www.ics.ele.tue.nl/˜akash

[8] Cho, M. H., Cheng, C.-C., Kinsy, M., Suh, G. E., Devadas, S., 2008. Diastolic arrays: throughput-driven reconfigurable computing. In: ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design. IEEE Press, Piscataway, NJ, USA, pp. 457–464.

[9] Culler, D., Singh, J., Gupta, A., 1999. In: Parallel Computer Architecture: a hardware/software approach. Morgan Kaufmann Publishers, Inc.

[10] Dasdan, A., 2004. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. ACM Trans. Des. Autom. Electron. Syst. 9 (4), 385–418.

[11] de Kock, E. A., 2002. Multiprocessor mapping of process networks: a jpeg decoding case study. In: ISSS '02: Proceedings of the 15th international symposium on System Synthesis. ACM, New York, NY, USA, pp. 68–73.

[12] Ferrari, A., Sangiovanni-Vincentelli, A., 1999. System design: Traditional concepts and new paradigms. In: Proc. of ICCD. pp. 2–12.

[13] Ganwal, O. P., Niewland, A., Lippens, P., 2001. A scalable and flexible data synchronization scheme for embedded HW-SW shared memory systems. In: Proc. of ISSS. pp. 1–6.

[14] Geilen, M., Basten, T., 2003. Requirements on the execution of kahn process networks. In: Proc. of the 12th European Symposium on Programming, ESOP 2003. Springer Verlag, pp. 319–334.

[15] Gswind, M., 2006. Chip multi-processing and the cell broad band engine. In: Proc. of CCF. pp. 1–8.

[16] Hoes, R., 2005. Predictable dynamic behaviour in noc-based multiprocessor system-on-chip. Master's thesis, Eindhoven University of Technology, Eindhoven (The Netherlands).

[17] Huang, K., D.Grunert, Thiele, L., 2007. Windowed FIFOs for FPGA-based multiprocessor systems. In: Proc. of IEEE 7th ASAP. pp. 36–41.

[18] ITRS, 2007. International techniology reoad map for semiconductors 2007. system drivers.

[19] Jeffay, K., Stanat, D. F., Martel, C. U., 1991. On non-preemptive scheduling of periodic and sporadic tasks. pp. 129–139.

[20] Jin, Y., Satish, N., K. Ravindran, K. K., 2005. An automated exploration frame work for fpga based soft multiprocessor systems. In: Proc. of 3rd CODES+ISSS CA, USA. Vol. 3199. pp. 273–278.

[21] Kahn, G., 1974. The semantics of a simple language for parallel programming. In: Proc. of IFIP Congress. North-Holland Publishing Co., pp. 471–475.

[22] Kai, R., Marek, J., Rolf, E., 2003. A formal approach to mpsoc performance verification. Computer 36 (4), 60–67.

[23] Karthikeyan, S., Ramadass, N., Haiming, L., Changkyu, K., Jaehyuk, H., Doug, B., W., K. S., R., M. C., 2003. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. SIGARCH Comput. Archit. News 31 (2), 422–433.

[24] Ken, M., Tim, P., Nuwan, J., Ron, H., J., D. W., Mark, H., 2000. Smart memories: a modular reconfigurable architecture. SIGARCH Comput. Archit. News 28 (2), 161–171.

[25] Kumar, A., 2009. Analysis, design and management of multimedia multiprocessor systems. Ph.D. thesis, Eindhoven University of Technology, Eindhoven (The Netherlands).

[26] Kumar, A., Fernando, S., Ha, Y., Mesman, B., Corporaal, H., 2008. Multiprocessor systems synthesis for multiple use-cases of multiple applications on fpga. ACM Trans. Des. Autom. Electron. Syst. 13 (3), 1–27.

[27] Kumar, A., Hansson, A., Huisken, J., Corporaal, H., 2007. An fpga design flow for reconfigurable network-based multi-processor systems on chip. In: Proc. of Design Automation and Test in Europe. Los Alamitos, CA. IEEE Computer Society, p. 117122.

[28] Kumar, A., Mesman, B., Corporaal, H., Theelen, B., Ha, Y., 2007. A probabilistic approach to model resource contention for performance estimation of multi-featured media devices. In: DAC '07: Proceedings of the 44th annual Design Automation Conference. ACM, New York, NY, USA, pp. 726–731.

[29] Kunzli, S., Poletti, F., Benini, L., Thiele, L., 2006. Combining simulation and formal methods for system-level performance analysis. In: DATE '06: Proceedings of the conference on Design, automation and test in Europe. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 236–241.

[30] Lee, E., Messerschmitt, D. G., jan 1987. Static scheduling of synchronous dataflow programs for digital signal processing. In: Proc. of IEEE Transactions on Computers. Vol. 36. IEEE, pp. 24–35.

[31] Lyonnard, D., Yoo, S., Baghdadi, A., Jerraya, A., 2001. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In: Proc. of Design Automation and Test in Europe. ACM Press, New York, p. 518523.

[32] Moonen, A., Bekooij, M., van den Berg, R., van Meerbergen, J., 2007. Decoupling of computation and communication with a communication assist. In: Proc. of DSD. pp. 63–68.

[33] Moonen, A., Berg, R. V., Bekooij, M., Bhullar, H., van Meerbergen, J., 2005. A multi-core architecture for in-car digital entertainment. In: Proc. of GSPx Conference.

[34] Neal, B., Vida, K., Mukul, K., Shuvra, B. S., 2003. Intermediate representations for design automation of multiprocessor dsp systems 7 (4), 307–323.

[35] Niewland, A., Brockmeyer, E., Corporaal, H., 2007. The impact of higher communication layers on NOC supported MP-SoCs. In: Proc. of NOCS. IEEE, pp. 107–116.

[36] Niewland, A., Kang, J., Gangwal, O., Sethuraman, R., Busa, N., Goosens, K., Llopis, R. P., Lippens, P., 2002. C-HEAP: A heterogeneous multiprocessor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. In: Proc. of DAC. pp. 233–270.

[37] Nikolov, H., Stefanov, T., Deprettere, E., 2006. Multi-processor system design with ESPAM. In: Proc. of CODES+ISSS. pp. 211–216.

[38] Pino, J. L., Lee, E. A., 1995. Hierarchical static scheduling of dataflow graphs onto multiple processors. In: IEEE International Conference on Acoustics, Speech, and Signal Processing. pp. 2643–2646.

[39] Sangiovanni-Vincentelli, A., Martin, G., Nov/Dec 2001. Platform-based design and software design methodology for embedded systems. In: IEEE Design and Test of Computers. pp. 23–33.

[40] Shabbir, A., Stuijk, S., Kumar, A., Theelen, B., Mesman, B., Corporaal, H., may 2010. A predictable communication assist. In: Accepted for publication in ACM International Confence on Computing Frontiers. ACM.

[41] Shaoxiong, H., Gang, Q., S., B. S., 2007. Probabilistic design of multimedia embedded systems. ACM Trans. Embed. Comput. Syst. 6 (3), 15.

[42] Sorin, M., Petru, E., Zebo, P., 2004. Schedulability analysis of applications with stochastic task execution times. ACM Trans. Embed. Comput. Syst. 3 (4), 706–735.

[43] Sriram, S., Bhattacharyya, S., 2000. Embedded multiprocessors; scheduling and synchronization. marcel dekker, new york, usa.

[44] Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprette, E., 2004. System design using kahn process networks: The compan/laura approach. In: Proc. of Design Automation and Test in Europe. pp. 340–345.

[45] Stuijk, S., Geilen, M., Basten, T., 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous data flow graphs. In: Proc. of Design Automation Conference. Vol. 3199. ACM press, New York, USA, pp. 899–904.

[46] Stuijk, S., Geilen, M., Basten, T., 2006. SDF^3 : SDF for free. In: Application of Concurrency to System Design, ACSD 06, Proceedings. IEEE. pp. 276–278.

[47] Taylor, M. B., Kim, J. S., Miller, J. E., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S. P., Agarwal, A., 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. IEEE Micro 22 (2), 25–35.

[48] Thiele, L., Chakraborty, S., Naedele, M., 2000. Real-time calculus for scheduling hard real-time systems. In: in ISCAS. pp. 101–104.

[49] Turjan, A., Kienhuis, B., Deprettere, E., 2004. An integer linear programming approach to classify the communication in process networks. In: Proc. of SCOPES. pp. 62–76.

[50] Xilinx, 2007. Resource pages[online].
URL Available from: http://www.xilinx.com