An Optimal Design Flow for Hard Real-Time Streaming Systems

Mohamed A. Bamakhrama, Jiali Teddy Zhai, and Todor Stefanov
Leiden Institute of Advanced Computer Science
Leiden University, Leiden, Netherlands
Email: m.a.m.bamakhrama@liacs.leidenuniv.nl

Abstract—In this paper, we address the problem of automated design of hard real-time embedded streaming systems. To this end, we introduce the notion of optimal design flow. An optimal design flow is one that accepts, as input, a set of hard real-time streaming programs, and then produces in a fully automated manner, as output, the final system implementation, which provably satisfies the timing constraints of the input programs. We propose a realization of such an optimal design flow and implement it. This implementation is called the Daedalus\textsuperscript{RT} design flow and it is available for download, as an open-source framework, from http://daedalus.liacs.nl/

I. INTRODUCTION

The design of modern embedded streaming systems is a difficult task. The difficulty of this task stems from the fact that multiple functionalities have to be implemented on a single system while meeting stringent timing constraints. In order to tackle this difficult task, modern embedded streaming systems are often realized using Multiprocessor System-on-Chip (MPSoC) technology [1]. In MPSoCs, multiple processors, hardware accelerators, and hardware peripherals are integrated into a single silicon chip. Realizing modern embedded streaming programs on MPSoCs entails several challenges. The first challenge is expressing the parallelism in the programs in a way such that (1) we exploit efficiently the multiple processors found in MPSoCs, and (2) we can reason analytically about the performance of the programs. Streaming programs are usually specified at an algorithmic level using a high-level sequential language such as C or MATLAB. Once the correctness of these sequential specifications is verified, they are passed to subsequent design stages. Therefore, it is necessary to parallelize such programs in order to exploit MPSoC platforms efficiently.

The second challenge is how to allocate and schedule the programs on the MPSoC such that all the timing constraints of the programs are met. In order to provide such guarantees, the system must use predictable hardware and software. Predictable here means that any HW/SW operation has a bounded worst-case duration. Additionally, the OS scheduler must be capable of enforcing timing isolation between the running programs.

The third challenge is how to design a complex MPSoC with the least designer effort. Traditionally, embedded systems have been designed at the level of Register Transfer Level (RTL) for hardware and low-level Application Programmer Interfaces (API) for software. However, as embedded systems move from uniprocessor systems to multiprocessor systems, a shift in the way such systems are designed is also needed.

Based on the aforementioned three challenges, we can say that the problem of designing a hard real-time streaming system is an intersection of three sub-problems as shown in Fig. 1. The grey area represents the area to which the aforementioned design problem belongs. In this grey area, the designer must produce a hard real-time multiprocessor system that runs several streaming programs in parallel.

The three sub-problems in Fig. 1 affect each other. For example, the way in which a program is parallelized influences how it will be scheduled during the system run-time. This, in turn, has a direct impact on the timing behavior of the program. Therefore, in order to solve these sub-problems, they must be addressed simultaneously by the designer. Solving these sub-problems together can be formulated as follows:

Given a set of hard real-time streaming programs, devise a systematic way to parallelize the programs and design, at the right level of abstraction, an MP-SoC which runs the parallelized programs such that the timing constraints of the programs are guaranteed to be always met during system run-time.

II. PROPOSED SOLUTION

To address the aforementioned problem, we propose a sequence of steps that the designer should follow in order to arrive at the final MPSoC implementation. These steps constitute a design flow (also called design methodology). An optimal design flow is one which accepts, as input, a set of hard real-time streaming programs, and then produces in a fully automated manner, as output, the final MPSoC implementation. Naturally, an optimal design flow should address the challenges outlined in Section I. Therefore, an optimal design flow should consist of steps that facilitate maximum design automation and correct-by-construction design. We identify
int main() {
  while(1) {
    for(i=1;i<=10;i++) {
      for(j=1;j<=3;j++) {
        src(&img[i][j],&img1[i][j]);
        if(j<=2)
          img[i][j]=f1(img[i][j]);
        else
          img[i][j]=f2(img[i][j]);
        snk(img[i][j],img1[i][j]);
      }
    }
  }
}

Listing 1: Example of a SANLP in C

such steps for hard real-time streaming systems and call them design flow pillars.

A. Automated Parallelization and Model Construction

Recall that most streaming programs are specified as sequential programs. Most of the execution of these sequential specifications is spent in nested loops [2]. Researchers have investigated several techniques for parallelizing such programs. One particular class of nested loop programs which received a lot of attention is Static Affine Nested Loop Programs (SANLP) [3]. This class has been shown to embody a large portion of streaming programs [4]. An example of a valid SANLP is shown in Listing 1.

It has been shown in [3] that a SANLP can be automatically analyzed to construct a parallel version of it. Hence, it is important to utilize this property to relieve the designer from the burden of parallelizing such programs manually. Therefore, the first step in an optimal design flow is automated parallelization. Given a sequential program, automated parallelization tools analyze the program and construct a parallel version of it. This parallel version of the program exposes the parallelism present in the original sequential program. Several parallelizing compilers were proposed for SANLPS, such as the PNgen compiler [5].

Analysis of parallel programs is a tedious task. Therefore, it has been recognized that the designers need to abstract from the actual programs by building high-level models of them using Models of Computation (MoC) [6]. Then, these models are used to analyze the program performance under different scheduling and mapping decisions. Such design approach is often called Model-Based Design (MBD) which constitutes the second pillar in the proposed design flow. Several MoCs have been proposed in the literature such as Synchronous Data-Flow (SDF, [7]) and its generalization Cyclo-Static Dafataflow (CSDF, [8]). Under these models, a program is modeled as a directed graph, where graph nodes represent the tasks in the program and the graph edges represent the data dependencies among the tasks. According to [9], almost all streaming programs can be modeled as SDF graphs. In this work, we choose the CSDF [8] model as the model of computation since it is expressive enough to model most streaming programs as shown in [9].

B. Real-Time Scheduling Framework

As mentioned earlier in Section I, hard real-time streaming programs must be designed in a way different than the traditional RTL approach. System-Level Design (SLD, [12]) has emerged as a promising solution to tackle this problem. Under SLD, the system is designed at higher level of abstraction than the RTL level. At system-level, the engineer deals with processors, buses, peripherals, and memories as the primitive blocks that form the system. System-level design abstracts the SoC design by considering it at a high level of abstraction as shown in Fig. 2. In Fig. 2, the system design is a process of mapping a set of tasks onto a set of processing elements. Once such a mapping is determined, for example using design space exploration, Electronic System-Level (ESL) synthesis tools [13] provide a (mostly) automated procedure to generate the RTL descriptions for hardware components and the parallel software running on the processors. System-level design represents the fourth pillar in the proposed design flow.

C. System-Level Design and Synthesis

Recall from Section I that modern embedded systems must be designed in a way different than the traditional RTL approach. System-Level Design (SLD, [12]) has emerged as a promising solution to tackle this problem. Under SLD, the system is designed at higher level of abstraction than the RTL level. At system-level, the engineer deals with processors, buses, peripherals, and memories as the primitive blocks that form the system. System-level design abstracts the SoC design by considering it at a high level of abstraction as shown in Fig. 2. In Fig. 2, the system design is a process of mapping a set of tasks onto a set of processing elements. Once such a mapping is determined, for example using design space exploration, Electronic System-Level (ESL) synthesis tools [13] provide a (mostly) automated procedure to generate the RTL descriptions for hardware components and the parallel software running on the processors. System-level design represents the fourth pillar in the proposed design flow.

III. THE PROPOSED OPTIMAL DESIGN FLOW: DAEDALUSRT

Given the four pillars described earlier, they constitute together the optimal design flow shown in Fig. 3. This design flow is called DaedalusRT [14] and it consists, in total, of six steps. The step number is marked inside a circle in Fig. 3.

Step 1 accepts the SANLPS as input, and then uses the PNgen compiler to parallelize them and generate the parallel specification of these input programs. The parallel specification
consists of the Polyhedral Process Network (PPN) representation of the program. PPN is a parallel MoC that is useful for code generation and optimizations. However, it is not suitable for performance analysis. This leads us to the next step.

In Step 2, the performance analysis model (i.e., CSDF) is derived automatically from the PPNs generated in step 1. Given a PPN, we propose in [14] an algorithm to derive a CSDF graph that is equivalent to the given PPN.

In Step 3, we perform WCET analysis on the parallel specification of the program. WCET analysis can be performed by either static analysis tools or profiling the code on the target MPSoC platform [15].

In Step 4, the CSDF models generated in step 2, the WCET values generated in step 3, and the user constraints, which include for example the type of scheduler and other parameters, are fed to the scheduling framework. We propose in [16]–[18] a scheduling framework that derives, for the tasks in the CSDF model, a corresponding real-time periodic task set. The framework computes the parameters of each task (i.e., period, start time, and deadline) and the buffer size of each communication channel such that a valid schedule is guaranteed to exist. After that, the framework performs schedulability analysis based on the scheduler type given by the user. This results in: (i) the architecture specification, which describes how many processors are needed to schedule the programs, and (ii) the mapping specification, which describes how to map the tasks to the processors.

In Step 5, the PPNs together with the architecture and mapping specifications are processed by ESPAM [19]. ESPAM is a SLD synthesis tool that supports MPSoC synthesis from PPNs. We have extended ESPAM to support synthesizing the target MPSoC hardware and software. The output from this step is a full MPSoC implementation consisting of the RTL needed to perform low-level synthesis for FPGA or ASIC together with the software running on each processor in the MPSoC.

Step 6 is the last step in the design flow and consists of performing low-level synthesis for FPGA or ASIC backends together with compiling the code for each processor.

IV. Evaluation and Results

In this section, we present the results of empirical evaluation of the Daedalus flow explained in Section III. We evaluate the different steps of the flow and demonstrate their effectiveness.

A. Evaluating Automated Parallelization and Model Construction

We evaluate steps 1 and 2 by parallelizing a set of real-life programs and deriving their CSDF models. The used programs are from the PolyBench benchmark [20]. The programs are specified as SANLPS in C and vary in size and complexity. The list of programs together with the time needed to parallelize them and derive their CSDF models is shown in Table I.

<table>
<thead>
<tr>
<th>Program</th>
<th># actors</th>
<th># edges</th>
<th># lines</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter-bank</td>
<td>69</td>
<td>89</td>
<td>367</td>
<td>1.60</td>
</tr>
<tr>
<td>FM radio</td>
<td>28</td>
<td>39</td>
<td>195</td>
<td>0.66</td>
</tr>
<tr>
<td>ADI solver</td>
<td>28</td>
<td>167</td>
<td>209</td>
<td>7.26</td>
</tr>
<tr>
<td>2D FFT kernel</td>
<td>17</td>
<td>71</td>
<td>144</td>
<td>0.89</td>
</tr>
<tr>
<td>2D gauss filter</td>
<td>11</td>
<td>26</td>
<td>75</td>
<td>7.82</td>
</tr>
<tr>
<td>Gram-Schmidt</td>
<td>9</td>
<td>20</td>
<td>48</td>
<td>1.85</td>
</tr>
<tr>
<td>Regularity detector</td>
<td>8</td>
<td>11</td>
<td>54</td>
<td>2.86</td>
</tr>
</tbody>
</table>

The time reported in Table I includes: (1) the time needed by the PNgen compiler to parse the C program and generate the parallelized program, (2) the time needed to derive the CSDF model as described in [14]. We see clearly that the first two steps of the proposed flow (i.e., automated parallelization and model construction) are very fast. The fast derivation of the parallelized program and CSDF model relieves the designer from the burden of writing the parallel specifications manually. Moreover, this allows the designer to explore a large number of alternative program specifications in a short period of time.

B. Evaluating Scheduling Framework

The results of evaluating the scheduling framework are described in detail in [16], [17]. We summarize these results here as follows. For 19 real-life programs, scheduling the programs as real-time periodic task sets results in: (1) optimal throughput for 16 programs, and (2) optimal latency for 14 programs. Optimal throughput and latency of a streaming program are those obtained under self-timed scheduling. Under self-timed scheduling, an actor is fired as soon as its input data are available. The aforementioned result shows clearly that periodic scheduling of streaming programs is capable of achieving optimal performance (i.e., throughput and latency) for most programs.
TABLE II. PROGRAMS USED FOR SYSTEM VALIDATION

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th># tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG encoder</td>
<td>Image encoder from raw format to JPEG format</td>
<td>6</td>
</tr>
<tr>
<td>JPEG decoder</td>
<td>Image decoder from JPEG format to raw format</td>
<td>2</td>
</tr>
<tr>
<td>sobel</td>
<td>Sobel edge-detector filter</td>
<td>5</td>
</tr>
<tr>
<td>pipeline</td>
<td>A synthetic program with pipeline topology</td>
<td>4</td>
</tr>
<tr>
<td>split-join</td>
<td>A synthetic program with split-join topology</td>
<td>4</td>
</tr>
</tbody>
</table>

C. System Validation

In this step, we validate the systems generated by the DaedalusRT design flow. To this end, we use a set of streaming programs as shown in Table II. First, we synthesize a set of systems, where each system runs a mixture of the programs shown in Table II. Then, the synthesized systems are prototyped on two types of hardware platforms which are: (1) Xilinx ML605 FPGA board, and (2) Avnet ZedBoard with Xilinx Zynq-7000 SoC. We run each synthesized system on actual hardware and monitor its execution to detect deadline misses and/or buffer underflows/overflows. Each synthesized system was validated by running it with real input data for a duration between 1 and 12 hours. For all the synthesized systems, no deadline misses and/or buffer underflow/overflow were detected during the whole validation phase.

V. OPEN ISSUES

In this section, we list the main open issues that we plan to tackle in the future.

1) Support for more expressive MoCs: A more expressive MoC allows a more accurate performance analysis. A first step towards this goal is the work in [21]. The authors in [21] present a scheduling framework similar to ours with support for a MoC called Affine Data-Flow (ADF) graphs, which is a generalization of CSDF. Another option is to support dynamic MoCs which model programs that change their behavior during run-time.

2) Support for programs with cyclic dependencies: Currently, the scheduling framework as proposed in [16], [17] supports only programs with acyclic dependencies. Recently, Benabid et al. [22] showed that any cyclic SDF graph can be scheduled as a set of periodic tasks provided that its back edges contain sufficient amount of initial tokens. Therefore, in theory, it is possible to schedule cyclic CSDF graphs by converting them to SDF graphs. However, it remains an open issue whether or not an analysis technique like the one in [22] can be applied directly on CSDF graphs.

3) Improving the WCET by considering the effect of mapping: During the WCET analysis, we assume that the communication operations take their worst-case latency under a fully congested interconnect. However, such assumption overestimates the WCET value. In a real system, many communication streams are isolated from the others. Therefore, communication operations occur without congestion and they do not take their worst-case latency. Therefore, it is possible to reduce the WCET values if the actual mapping is taken into account.

REFERENCES