DATAFLOW PROGRAMMING: BUILDING PORTABLE AND EFFICIENT DESIGNS IN HETEROGENEOUS PROGRAMMABLE PLATFORMS

Jörn W. Janneck, Christopher H. Dick

Xilinx, Inc.
San Jose, CA, U.S.A
{jorn.janneck,chris.dick}@xilinx.com

ABSTRACT

The development of modern electronic systems increasingly faces qualitative pressures coming from a growing technical diversity and heterogeneity of the computational elements used to build them, as well as move toward parallel computing resulting from the fact that sequential processors are not becoming faster at the rate they used to. These two developments require a fundamental shift in the way systems are conceived and implemented. We propose a dataflow design methodology built around a notion of transactional execution of asynchronously communicating elements ("actors"). This model permits efficient implementation on a variety of computating platforms. It is based on simple, understandable abstractions which are a natural medium for expressing the various levels of parallelism in an application. Its parallelism naturally scales with application size, i.e. larger programs tend to include more parallel execution than smaller ones. In addition, this dataflow methodology also provides the foundation for an entirely new approach to the profiling and analysis of concurrent programs, which can be used to guide the implementation of a dataflow program and its mapping to a computing platform.

1. INTRODUCTION

The diverse set of processing tasks associated with complex wireless systems like WiMax, 3G, 3G LTE is often realized using a heterogeneous processing platform comprising DSP processors, general purpose processors (GPP) and field programmable gate arrays (FPGA). One of the challenges associated with mapping a complete wireless application comprising, among other things, of IP (internet protocol)-centric processing tasks, wireless medium access control (MAC) functions and complex real-time signal processing datapaths, is the ability to rapidly perform design space exploration in order to deliver the functionality within the required envelope of performance, power and cost.

Further, the subsequent step of taking a specific instance in the design space and generating the required combination of software for DSP processors and GPPs, and hardware description language (HDL) modules for an FPGA together with all of the implied interfaces is a complicated process using current generation design flows and tools. In practice, it is common for a set of platform specific (DSP processor, GPP, FPGA) tools to be employed for design capture and independent simulation of each of the functions to be mapped to the multiple processing domain, and it is often difficult, if not impossible, to perform a holistic simulation of the aggregate code base spanning the heterogeneous processing targets.

We observe that by employing a heterogeneous processing platform, the nature of the programming task is now one of design capture, simulation and code generation for a parallel computing platform, exploiting a hierarchy of computational and data parallelism present in the problem. Each silicon device in the system, be it a DSP processor, GPP or FPGA, is typically a concurrent computing machine in its own right. For the case of the DSP and GPP there is an trend to incorporate multiple cores in each device (multi-core processors). The FPGA is a massively parallel device comprising many hundreds of arithmetic units. So in addition to the aforementioned requirements of the programming model, it is also a requirement to generate efficient code that can be factored across the pool of parallel resources in each processing device on the platform.

This paper introduces a parallel programming model (section 2), and a typical stream-oriented application (and MPEG-4 decoder, section 3). It then presents techniques for obtaining metrics about the parallel program by *profiling* and analyzing its execution, before finally discussing its implementation in hardware and software (section 5).

2. DATAFLOW MODEL

The work in this paper is based on a *dataflow* programming model, in which (dataflow) programs consist of computational kernels, called *actors*, which are connected to each other by lossless directed FIFO channels. These channels attach to actor *(input or output) ports* and are used to send packets of data, called *tokens* (Fig. 1). This model is embodied in actor languages such as CAL [1], which is the one we used

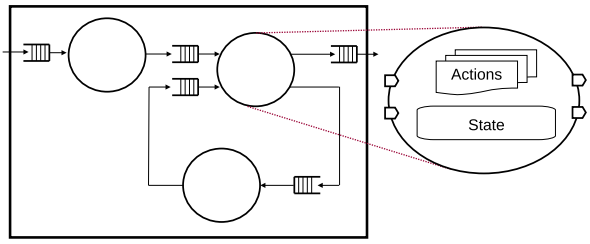

Fig. 1. A dataflow network with actors connected by FIFOs, and the inner components of one actor, its actions, ports, and state.

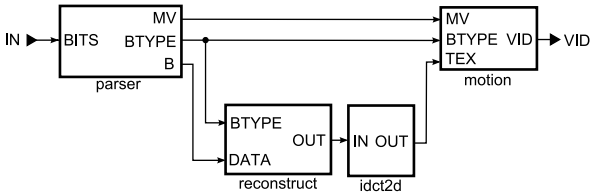

Fig. 2. Top-level view of the MPEG decoder, depicting parser, AC/DC reconstruction, IDCT, and motion compensation.

for the work presented here. Depending on the implementation platform, the FIFOs may be bounded or unbounded, and size constraints may or may not apply to individual tokens. An actor in turn consists of

- input ports,
- output ports,
- internal state,
- a number of transition rules called *actions*.

It executes by making discrete, atomic *steps* or *transitions*. In each step, it picks exactly one action from its pool of actions (according to whatever conditions are associated with that action), and then executes it, at which point it may do any combination of the following things:

- consume input tokens,
- produce output tokens,
- modify the state of the actor.

The state of an actor is strictly local, i.e. it is not visible to any other actor. This is what allows actors to run relatively independently of each other without concern for whether the order in which they end up executing their actions causes race conditions on their state.

Actors are similar to objects in object-oriented programming in the sense that they encapsulate some state and associate it with the code manipulating it (the actions). They differ from objects in that actors cannot *call* each other. There is no transfer of control from one actor to another, each actor can be thought of as its own independent thread.

3. EXAMPLE: AN MPEG-4 SP DECODER

The example used here for illustration is an MPEG-4 Simple Profile video decoder, a computational engine consuming a stream of bits on its input (the MPEG bitstream), and producing video data on its output. At 30 frames of 1080p per second, this amounts to $30 * 1920 * 1080 =$ approx. 62.2 million pixels per second. In the common YUV420 format, each pixel requires 1.5 bytes on average, which means the decoder has to produce approx. 93.3 million bytes of video data (*samples*) per second.

Fig. 2 shows a top-level view of the dataflow program describing the decoder.[1] The main functional blocks include a parser, an reconstruction block, a 2-D inverse discrete cosine transform (IDCT) block, and a motion compensator. All of these large functional units are themselves hierarchical compositions of actors—the entire decoder comprises of about 60 basic actors, which together take approximately 4,000 lines of code (LOC).

The parser analyzes the incoming bitstream and extracts the data from it that it feeds into the rest of the decoder. It is by far the most complex block of the decoder, more than a third of the code is used to build the parser. The reconstruction block performs some decoding that exploits the correlation of pixels in neighboring blocks. The IDCT, even though it is the locus of most of the computation performed by the decoder, is structurally rather regular and straightforward compared to the other main functional components. Finally, the task of the motion compensator is to selectively add the blocks issuing from the IDCT to blocks taken from the previous frame. Consequently, the motion compensator needs to store the entire previous frame of video data, which it needs to address into with a certain degree of random access.

One interesting aspect of this application is that it exercises a broad range of language features and application requirements. It contains control-dominated parts such as the parser, heavy-duty computational elements with little or no state but high throughput requirements such as the IDCT, and state-heavy parts that contain and access a large amount of memory such as the motion compensator. In the IDCT, computation and the flow of data is very regular and even statically analyzable, while in the motion compensator it is very much data dependent, and the sequence of operations in the parser is effectively entirely controlled by the incoming data.

[1] The source code for the decoder discussed in this paper is available at `http://opendf.sf.net`.

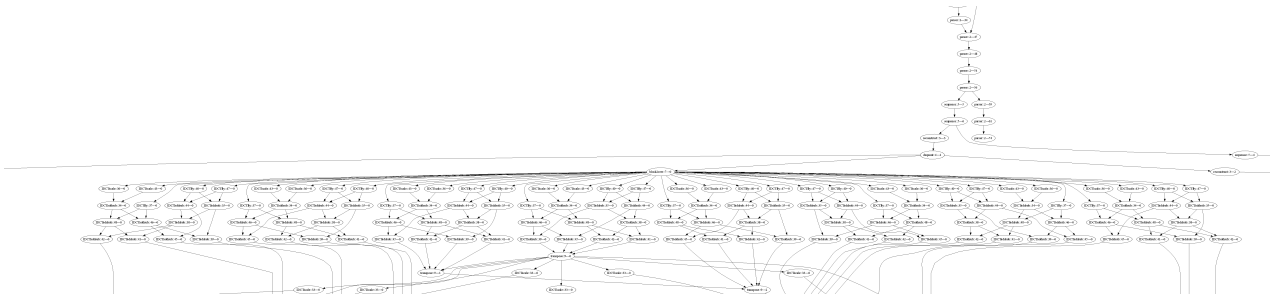

Fig. 3. Part of a trace of an MPEG-4 decoder.

4. PROFILING

This section discusses some techniques for producing quantitative data from runs of a dataflow program, which are presented in greater detail in [2].

4.1. Traces

Unlike procedures/functions in a sequential program, the basic building blocks of a dataflow program, the actions, do not transfer control to each other, there is no caller/callee relationship between them, and there is consequently no hierarchical call-graph that form the basis of most sequential profilers.

However, that does not mean that action executions are independent of each other. For instance, in a system where actor A produces tokens that actor B consumes, the actions in B are clearly dependent on those in A because for any token going from A to B, the action producing it must have occurred before the action consuming it. Every token, therefore, establishes a *token dependency* between two action *executions*, viz. the one that produced it, and the one that consumed it.[2] Typically, the actions in question will be in two different actors, but they need not be: in case of direct feedback, where an actor directly consumes tokens it produces, action executions of the same actor will be token-dependent on each other.

Two action executions within the same actor may or may not be independent. If they access shared state, or if they use the same input or output ports to communicate tokens, they are said to be related by a *state dependency* or a *port dependency*, respectively. In that case, the sequence of their execution by the actor matters, otherwise they may be executed in any order.

A *causation trace* (or simply *trace*) of a dataflow program is a directed acyclic graph such that

- every node is a step of one actor in the program,
- every edge from v_1 to v_2 is a dependency (either through a token, state or port) from v_2 on v_1, implying that therefore v_1 has to be executed before v_2.

Fig. 3 shows part of the trace of a real-world application, in this case an MPEG4 decoder, in which dependencies flow from top to bottom. The trace was extracted from an execution of a dataflow model of the decoder, by tracking the production and consumption of tokens, and the access to ports and actor state.[3] The initial segment at the top is relatively thin and linear, denoting a more sequential part of the computation (in this case, the parsing of the MPEG bitstream). Following that segment are clusters of computation that are much wider, as a result of the fact that more steps are independent of each other and can therefore be executed at the same time, permitting more parallel implementations (in this case, these clusters represent IDCTs).

Depending on the application, traces can be of considerable size—for instance, in the case of the MPEG-4 decoder that we used in our experiments, the computation corresponding to two frames of QCIF video (176x144 pixels), one i-frame and one p-frame, resulted in a trace with approximately 260,000 nodes.

Once we have obtained a trace from the execution of a dataflow program, we need to analyze it to gain insight into performance aspects of the dataflow program. The next section discusses a few basic techniques for analyzing causation traces.

4.2. Structural analysis

In media processing, many dataflow programs only have a small number of input and output ports, e.g. the MPEG-4 decoder has one of each: an input port consuming bytes of an MPEG-4 stream, and an output port generating macroblocks (16x16 blocks) of video pixels. Since the output is also subject to strict real-time requirements (a certain amount of video has to be generated in a given amount of time), we may want to investigate aspects of the computation required for each piece of output, in our case for each macroblock. A trace of

[2] If a connection fans out to several consumers, the tokens gets copied to each and thus creates a dependency of each consuming action on the same producing action.

[3] The source code of the decoder and the execution infrastructure is available at `opendf.sf.net`.

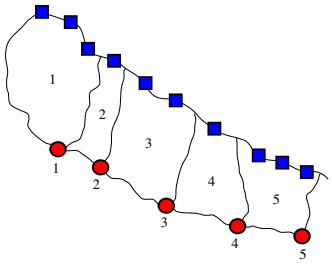

Fig. 4. Output-tagged trace.

such a system generally has the structure shown in Fig. 4. All computation happens between a sequence of input steps (the squares) and a sequence of output steps (the circles). We now *output tag* the trace by the following procedure: Starting from the first output step, we tag every step that it depends on, directly or indirectly, with 1. Then, proceeding with the second output step, we tag every untagged step it depends on with 2 and so forth. In this manner, we divide the entire computation into *output regions*, containing steps contributing to specific tokens of the output, as shown in Fig. 4.

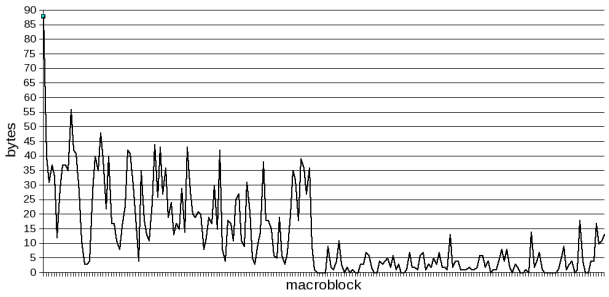

Fig. 5. Input consumed for each macroblock by the MPEG decoder, in bytes.

We can use an output-tagged trace in various ways. For instance, we can count how many *input steps* occur in each region, which provides us with a measure of how much input is required for each piece of output. The result for the first two frames on an MPEG-4 stream are shown in Fig. 5. The x-axis in this graph (and those following) represents the output tokens of the system in order, in the case of the MPEG decoder each tick is a macroblock. The y-axis represents some measure of the corresponding region in the trace. In Fig. 5 it is the number of input bytes consumed in the region. As the first frame is an i-frame and the second is a p-frame, the amount of input drops sharply in the middle of the output sequence, due to motion compensation.

The size of the regions in the output-tagged trace corresponds to the number of steps we need to perform for each macroblock. If we want to use this as a measure for the computational effort, we might want to weigh each step according

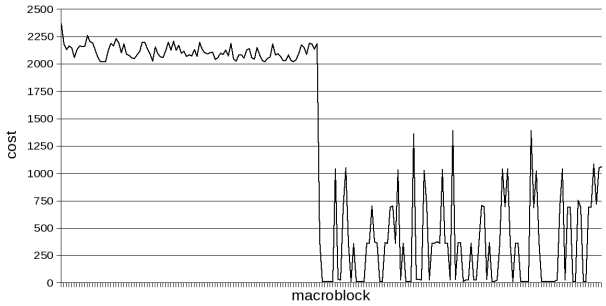

Fig. 6. Computational cost for each macroblock in an MPEG-4 decoder.

to the effort required to execute it. In the example, all weights are 1, and the resulting graph is shown in Fig. 6. Again, we can see that the motion-compensated p-frame requires much less computation, but we also see that the variability is much higher.

Looking at the output regions individually, we may determine the longest path in them. It represents the minimal latency for a particular macroblock, i.e. at maximal parallelization (which also means that no step is waiting for a resource), this is the time required for the computation. If we divide the number of steps by length of that minimal latency, we obtain a measure of the *parallelizability* of the computation—the shorter the minimal latency, and the more work is done in total, the more work can be done on average at the same time.

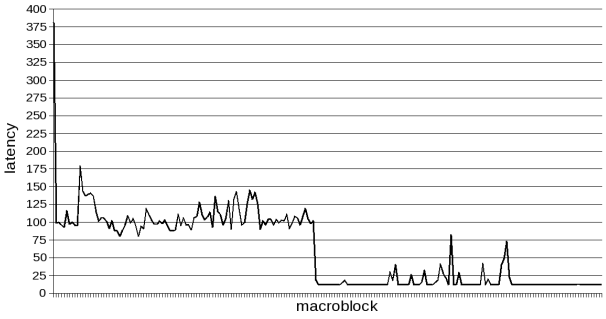

Fig. 7. Minimal additional latency per macroblock in an MPEG-4 decoder.

We can combine post-mortem scheduling with the output-tagged trace in a number of interesting ways. For instance, the longest path of a region by itself is not a very useful figure, since in a parallel implementation some of the computation usually happens concurrently with the computation of previous regions. However, for a given schedule we can determine the time between the production of macroblocks, which is a more meaningful metric. Fig. 7 shows how much

time expires between two successive macroblocks.

5. IMPLEMENTATION

This section discusses some aspects of implementing dataflow programs in hardware and in software.

5.1. Hardware synthesis

When generating hardware implementations from networks of CAL actors [3], we currently translate each actor separately, and connect the resulting RTL descriptions using synchronous or asynchronous FIFOs. Consequently, we do not employ any cross-actor optimizations.

Actors interact with FIFOs using a handshake protocol, which allows them to sense when a token is available or when a FIFO is full. We do not synthesize any static schedule between actors, which means that the resulting system is entirely self-scheduling based on the flow of tokens through it, and the content of the FIFOs.

The translation of each CAL actor into a hardware description follows a three-step process:[4]

1. instantiation
2. precompilation
3. RTL code generation

During instantiation, elaborating the network structure yields a number of actor *instances*, which are references to CAL actor descriptions along with actual values for the formal parameters. From this, instantiation computes a *closed* actor description, i.e. one without parameters, by moving the parameters along with the corresponding actual values into the actor as local (constant) declarations. It then performs constant propagation on the result.

Precompilation consists of some simple actor canonicalization steps, in which several features of the language are translated into simpler forms and other sou rce-to-source transformations, e.g. inlining procedure and function calls. Then the canonical, closed actors are translated into *XLIM*, an intermediate XML format for describing a collection of communicating threads, each of which represented as an imperative program in static single-assignment (SSA) form.

The final phase of the translation process generates an RTL implementation (in Verilog) from a set of threads in SSA form. The first step simply substitutes operators in expressions for hardware operators, creates the hardware structures required to implement the control flow elements (loops, if-then-else statements), and also generates the appropriate muxing/demuxing logic for variable accesses, including the Φ elements in the SSA form.

[4]The code for instantiation and precompilation is available on `http://opendf.sf.net`.

	Size slices, BRAM	**Speed** kMB/S	**Code size** kLOC	**Time** MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12

The above table shows the quality of the result produced by the RTL synthesis engine for the MPEG decoder, compared with a hand-written decoder in VHDL. Note that the code generated from the high-level dataflow description actually outperforms the VHDL design in terms of both throughput (in thousand macroblocks per second, kMB/s) and silicon area.

5.2. Software synthesis

In [4] Roquier et al. present *Cal2C*, a compiler translating individual CAL actors into software implementations in the C programming language. Actors are turned into individual threads, which can be scheduled with respect to one another either cooperatively or preemptively.

Translating the functions, procedures, and actions contained within the description of an actor produces a single C file. The way Cal2C uses the C language implies some limitations for the code that can be translated by this compiler, such as functional and procedural closures, which have no direct correspondence in C, and which Cal2C will not translate.

The translation process reuses many of the front end preprocessing steps of the HDL code generator. After some preprocessing, type inference is performed on the actor code, in preparation for performing a number of type-dependent transformations: functions that return lists are inlined, and list sizes are computed statically.

The resulting transformed and annotated AST is then translated into the C Intermediate Language (CIL) [5], and functional constructs of CAL are replaced by imperative ones. C code is generated by calling the pretty-printer included in the CIL framework.

For the 4,000 LOC of the CAL decoder, Cal2C generates about 10,400 LOC in C. The compiled program runs on a SystemC-based cooperative task scheduler running on a 2.4GHz Pentium at about 2 kMB/s, compared to the 290 kMB/s of an implementation that runs entirely in reconfigurable logic.

6. CONCLUSION

In this paper we have presented a dataflow programming paradigm for stream-oriented computation that permits the description of systems in such a way that they can be mapped efficiently to both programmable hardware as well as software targets. We have demonstrated this capability by translating an at-size real world application with a broad range of application characteristics to both programmable hardware as well as software.

Since a dataflow program is in essence a concurrent description of an algorithm, traditional profiling techniques are no longer applicable. We have introduced an approach to profiling based on a explicit description of the structure of a computation, and shown a analysis technique for obtaining a number of metrics from that structure.

A natural extension of this work is the implementation of dataflow systems in a combination of programmable hardware and software, asynchronously interacting through FIFO channels. We would use profiling to identify throughput-critical components, and implement those in hardware, while the rest could be realized as software on either a soft or a hard processor. The fact that we have automatic translators that can target software and hardware from the same source allows us to efficiently, and with minimal user intervention, explore different implementation options, realizing different resource cost/performance tradeoffs without rewriting the application.

7. REFERENCES

[1] Johan Eker and Jörn W. Janneck, "CAL language report," Technical Memo UCB/ERL M03/48, Electronics Research Lab, University of California at Berkeley, December 2003.

[2] Jörn W. Janneck, Ian D. Miller, and David B. Parlour, "Profiling dataflow programs," in *Proceedings of the 2008 IEEE International Conference on Multimedia and Expo (ICME)*, 2008.

[3] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet, "Synthesizing hardware from dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008.

[4] Ghislain Roquier, Matthieu Wipliez, Mickaël Raulet, Jörn W. Janneck, Ian D. Miller, and David B. Parlour, "Automatic software synthesis of dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008.

[5] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer, "CIL: An infrastructure for c program analysis and transformation," in *Proceedings of Compiler Construction (CC) 2002*.