PROTEUS: AN ARCHITECTURAL SYNTHESIS TOOL BASED ON THE STREAM PROGRAMMING PARADIGM

Nikolaos Bellas[1] Sek M. Chai[2] Malcolm Dwyer[2] Dan Linzmeier[2] Abelardo Lopez-Lagunas[3]

Computer Engineering and Communications Department[1]
University of Thessaly, Volos, Greece
nbellas@uth.gr

Motorola, Inc.[2]
Schaumburg, IL
USA
sek.chai@motorola.com

Departamento de Mecatrónica[3]
ITESM-Toluca, Mexico

ABSTRACT

The problem of automatically generating hardware modules from a high level representation of an application has been at the forefront of EDA research in the last few years. Such an EDA methodology would potentially enable the large pool of software engineers and algorithm IP experts without architectural and hardware expertise to design and implement platform systems, thus dramatically reducing time to market.

This paper makes the argument that such a methodology requires a programming model beyond the sequential semantics of languages like C/C++. We argue in favor of the streaming programming model in which computation and data communication are explicitly separated and optimized. Our architectural synthesis tool, Proteus, processes stream programs that partition the application into a series of streaming kernels that operate on streams of data elements. Proteus produces efficient hardware accelerators that provide orders of magnitude higher throughput than a software implementation, at an area cost very close to manual HDL implementation.

1. INTRODUCTION

Programming models based on von Neumann architectures would naturally map to a centralized processor and memory subsystem. These traditional architectures with centralized compute structures and large caches do not map well on an FPGA where the underlying logic and memories are small and distributed throughout the chip. To properly capture the programmer's intent, the programming model must express the data access and communication explicitly for the compiler to generate the appropriate memory structures such as buffers and bus networks that implement the memory subsystem.

Therefore, a new programming paradigm is required that exposes data communication to the programmer and enables the structuring of bandwidth-efficient software and hardware systems. Stream programming is an example of such programming model and is ideal for architectures synthesized onto FPGAs because the model explicitly defines the communication patterns and the nature of the computation, enabling the synthesis of unique memory subsystems and computation units.

This paper presents Proteus, an architectural synthesis CAD tool, which produces synthesizable HDL code for a network of hardware accelerators given a high level streaming representation. First, in Section 2, we detail the advantages of the streaming programming model to express low level and high level parallelism and its suitability to express computation that will be implemented into hardware, and we explain how we use streaming as the underlying technology of Proteus.

In our work, we use a well-crafted architectural template that can be instantiated to match the performance requirements, and available FPGA resources for a particular application [1]. Section 3 provides details of the optimizing compiler and hardware generator that translates the streaming kernels into synthesizable Verilog.

There has been a substantial amount of research on hardware generation starting from a high level language [2],[3],[4],[5],[6]. Most prior research focuses on the computational aspect of the application. The streaming architectures generated by the tools presented in this paper are different because we focus not only on the computation but also on the memory subsystem. We developed a benchmark suite of streaming DFGs to further evaluate our approach and to show the trade-offs involved between performance and area. These results are shown in Section 4.

2. STREAMING PROGRAMMING MODEL

This section presents the details of the streaming programming paradigm. In the stream programming model, a kernel of computation is formed as a set of

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

localized processor operations that are independent and self contained. The processing in each computation kernel is regular and repetitive, which often comes in the form of a loop structure. These computation kernels can be mapped onto a separate hardware accelerator without frequent interaction with the processor.

Traditional compiler optimizations for instruction and data level parallelism such as loop unrolling and modulo scheduling [7] can be applied not only inside a computation kernel but also across kernels.

Computation kernels consume and produce a uniform sequence of data elements (stream records) since the kernel operations are regular. Stream data appear to be sequential to the computation kernels even though they may be scattered throughout memory. Global variables are usually not referenced in a kernel. Instead, the stream and other scalar values, which hold persistent state, are identified explicitly as variables in a data stream or as signals between kernels.

The design process starts with the application programmer describing the application in a high level language such as C, or a combination of C and Data Flow Graphs (Fig. 1). The computation kernels and their associated memory accesses are then used to generate hardware accelerators to lift the heavy computation load from the main processor. Scalar processors are reserved for normal conditional code which is not easily parallelizable.

A task or computation kernel is expressed using a streaming data flow graph (sDFG) language, shown in Fig. 1(b)-(c). A sDFG consists of nodes, representing basic arithmetic and logical operations, and directed edges representing the dependency of one operation on the output

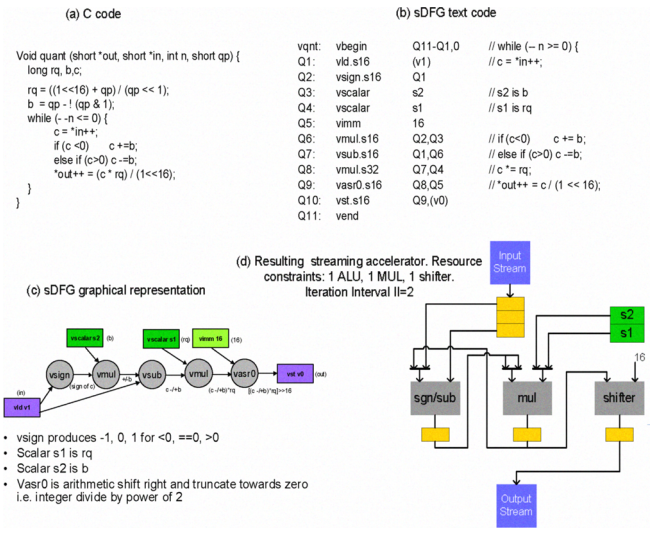

Fig. 1. The C quant function is transformed to a sDFG and finally to a streaming accelerator. Note that only the code within the while loop is mapped into hardware.

of a previous operation.

In this DFG language, all dependencies are explicitly stated. This simplifies the scheduler's task of identifying dependencies and determining which operations can be scheduled in parallel, resulting in schedules that are often close to optimal, given the functional unit and interconnect limits.

Fig. 1 shows a sequence of representations of a quantization function which may be part of a video compression algorithm like MPEG-4. As shown in Fig. 1(b), the input and output streams are loaded in the kernel and stored to the memory with the *vld* and *vst* operations, respectively. Internal operation nodes represent computation such as extracting the sign of a number, subtraction, multiplication, and arithmetic shift. Note that the kernel simply operates on incoming streams and produces outgoing streams, and that there is no dynamic address calculation of memory accesses in the kernel.

In this simple example, if we assume that the final implementation includes only one ALU, one multiplier, and one shifter, the organization of these resources could be depicted as shown in Fig. 1(d). Each functional unit is supported with output queues of depth 1 used to temporarily store the produced data. The shared resources have multiplexers at their input ports. The implementation of Fig. 1(d) follows the template architecture shown in Fig. 2. The template consists of two parts: data path and stream unit. The data path is generated based on a computation kernel (such as the one of Fig. 1(b)), while the stream unit is derived from stream access patterns.

A data path consists of a network of functional units that produce and consume streaming data elements. A reconfigurable link is formed by a tree of multiplexers and buffers to direct proper data elements from the output of a producing functional unit to the input of the next consuming functional units. The control logic is distributed and spatially near the corresponding functional unit, multiplexer, and buffers. Unlike a centralized VLIW codeword which tends to increase the signal critical path, distributed control logic avoids long interconnects in critical paths. The reconfigurable parameters of the data path include the following: type of functional units (ALUs, multipliers, shifters, etc), the custom operation performed within a type (e.g. only addition or subtraction for an ALU), the width of the functional unit, the size and number of storage elements, the interconnect between functional units, and the bandwidth to and from the stream unit [8].

The stream programming model allows a programmer to explicitly define the characteristics of the data streams between computation kernels and the memory. Dedicated hardware, called stream unit, is used for data movement such that communication is decoupled from computation. The stream units assemble the stream records and present

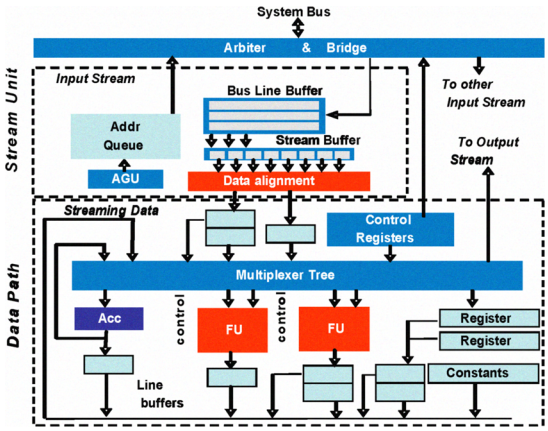

Fig. 2. Streaming accelerator template including the Stream Unit and the Data Path.

them to the computation kernels as ordered packets. Much like vector processing, stream programs hide latency, amortize instruction overhead and expose data parallelism by operating on large sets of data.

The programmer describes the memory access patterns using stream descriptors, which define the shape and location of data in memory. This decoupling allows the stream interface units to take advantage of available bandwidth to prefetch data before it is needed. Data is transferred though the stream units, which are programmed using stream descriptors.

A stream descriptor is represented by the tuple (*Type*, *Start_Address*, *Stride*, *Span*, *Skip*, *Size*) where:

- *Type* indicates how many bytes are in each element (Type is 0 for bytes, 1 for 16-bit half-words, etc.)
- *Start_Address* represents the memory address of the first stream element.
- *Stride* is the spacing in number of elements between two consecutive stream elements.
- *Span* is the number of elements that are gathered before applying the skip offset
- *Skip* is the offset applied between groups of span elements, after the stride has been applied
- *Size* is the number of elements in the stream

The stream unit handles all issues regarding loading/storing of data including: address calculation, alignment, data ordering, and bus interfacing. The stream unit consists of one or more input and output stream modules, and is generated to match the characteristics of the programmer's description of the stream data through stream descriptors, the characteristics of the bus-based system, and the streaming datapath (Fig. 2).

3. PROTEUS TOOLSET

The sDFG kernels and their stream descriptors, and other resource constraints, such as maximum gate count and maximum bandwidth in and out the kernel, are used by our compiler to allocate a set of functional units (step 1). Then using modulo scheduling compilation a sequence of events are arranged so that the functional units can operate properly (step 2). The sequence of events is similar to a series of VLIW (very long instruction word) operations. The streaming hardware accelerator, consisting of a data path and stream unit, is then selected (step 3). An interim hardware description file is used to list the components within the accelerator. The sDFG nodes and arcs have been associated with hardware resources such as functional units and queues. A set of state machines is also listed to generate the proper control signals. The hardware is then generated (step 4) and synthesized (step 5) into the FPGA. This process is repeated for each streaming kernel. The following paragraphs focus on the VLIW scheduling process.

The scheduler receives as input the sDFG along with the user and system constraints and schedules the operation of the sDFG to optimize throughput. The scheduler uses modulo scheduling to overlap multiple iterations in each cycle and exploits all the available parallelism under the resource constraints and data dependencies.

A strict lower bound of the initiation interval, called Minimum Initiation Interval (MII), is obtained by the number of available resources and the loop cross-iteration data dependencies. The schedule is generated within the MII window by first scheduling the nodes from top to bottom (forward scheduling) using a greedy approach. In this step, the nodes are scheduled immediately when all their parents have been scheduled and there exists an available resource to execute them.

The scheduler only generates the code for the steady state body of the schedule and not for the prologue and epilogue, as is often the case in modulo scheduling. To be able to perform correct execution of the prologue and epilogue parts of the scheduled code, the generated hardware utilizes *valid bits*. Each data token that populates the functional unit inputs, outputs and line queues in every clock cycle is tagged with a valid bit. An operation produces valid output data only if both input data are valid. A source operation (like a stream load) produces data with valid bits when the data are available, and a sink operation (like a stream store) accepts data only when they are valid. This hardware enhancement ensures correct execution of the code, since a functional unit produces valid data in a given clock cycle, only when it performs an operation in that cycle and its inputs are all valid.

Next, the tool flow binds the operation nodes to the functional unit slices, and generates the register queues at the output of each slice to store the streaming outputs as they are produced by the FUs.

The stream unit design is generated based on user and system constraints. The size and number of buffer elements

are chosen to meet the performance of the bus as well as the target performance of the generated data path. For example, the number of bus address queue elements, used to store pending addresses, is set to at least the bus pipeline factor so that bus transfers are sustained without stalling the data path. The number of line buffer elements, used to store data, should be at least the bus size to enable bus transfers.

In addition, the number of stream line buffer (used to store pending stream elements in a FIFO) is set to match the maximum bandwidth of the data path so that the stream unit can buffer the proper number of stream elements that can be consumed by the data path in a single cycle.

Finally, the HDL constructor reads the HLM representation and emits structural Verilog for the data path and the stream unit.

4. EXPERIMENTAL EVALUATION

The selected benchmarks are part of applications such as video codecs, image processing and computer vision. Some benchmarks are "kernels", representing portions of applications selected for implementation and manually mapped to sDFGs and stream descriptors.

Benchmarks such as *lpr, hpf, lpf*, and l*ens* are complete applications and consist of smaller component kernels. The *lpr* application concerns the automatic recognition of license plates and is implemented into three stages.

The *lens* benchmark, which consists of a single sDFG, is a complex image warping application that performs correction of images from the wide angle lens space to the 2D rectilinear space. The two benchmarks, *lpf* and *hpf,* perform a sequence of image processing steps used in a digital camera.

Fig. 3 shows the number of FPGA slices for the data path part of these benchmarks. Higher slice count represents sDFGs that require more complex computation. For example, the *lens* occupies almost 30% of the Virtex 4LX80 device. The intent of Fig. 3 is to show the capability of Proteus to handle both simple and complex sDFGs producing very high quality accelerators in a fraction of the manual design time.

Using Proteus has resulted into large productivity gains. For example, an input representation of 800 lines of the sDFG *lens* code generated more than 100,000 lines of Verilog, significantly cutting the engineering development time. To further illustrate the efficiency of the tool, we implemented the software version of the *lens* code in C and we compared its performance when executed in the Core 2 Quad processor against the hardware accelerators generated by Proteus. In order to make a fair comparison, we optimized the code to exploit the quad-threaded, SIMD architecture of Core 2 Quad by using the Intel pthread library and by manually rewriting the inner loops of the benchmark using the x86 SSE ISA extensions. The FPGA version provided a speed up of around 1.4x, or 56x per Hz,

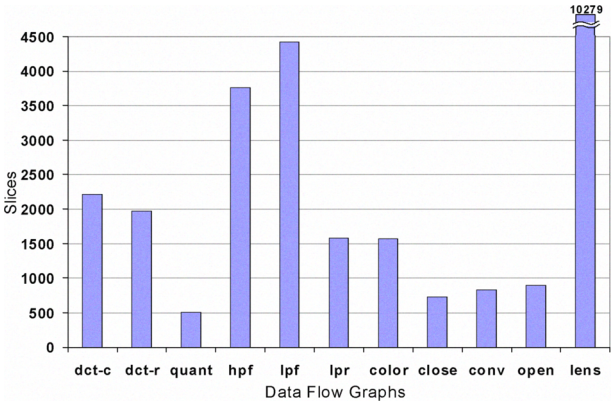

Fig. 3. Area comparison between different application benchmarks

enabling real time lens distortion correction of VGA output frames at 22 frames/sec. Moreover, the clock frequency, throughput, and area overhead of the generated hardware were very close to the theoretical bounds of a manual HDL implementation.

5. REFERENCES

[1] Nikolaos Bellas, Sek Chai, Malcolm Dwyer, Dan Linzmeier An Architectural Framework for Automated Streaming Kernel Selection. 14th *Reconfigurable Architectures Workshop (RAW),* March 2007, Long Beach, CA

[2] M. Gokhale. et. al. Stream-Oriented FPGA Computing in the Streams-C High Level Language, *Proceedings of International Conference on Field Programmable Custom Computing Machines* (FCCM), 2000, 49-56.

[3] D. Lau, O. Pritchard, P. Molson. Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. *International Symposium on Field-Programmable Custom Computing Machines*, April 2006, Napa Valley, CA, 45-56

[4] F. Plavec, Z. Vranesic, S. Brown. Towards Compilation of Streaming Programs into FPGA hardware. *Forum on Specification, Verification and Design Languages*, September 2008, Sophia Antipolis, France

[5] O. Mencer. et. al. Design Space Exploration with a Stream Compiler. *Proceedings of International Conference on Field Programmable Technology* (FPT), Tokyo, December 2003, 270-27

[6] Banerjee P. et. al. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. Proceedings of the IEEE Symposium on Field Custom Computing Machines (FCCM), April 17-19, 2000, pp. 39-48, Napa Valley, CA

[7] B.R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Processing,* 1996, 24:3-64

[8] N. Bellas, S.M. Chai, M. Dwyer, D. Linzmeier. Template-based generation of streaming accelerators from a high level representation. *International Symposium on Field-Programmable Custom Computing Machines (FCCM),* April 24-26, 2006, Napa Valley, CA