Template-based generation of streaming accelerators from a high level representation

Nikolaos Bellas, Sek M. Chai, Malcolm Dwyer, Dan Linzmeier
Embedded Systems Research, Motorola Labs,
(bellas@labs.motorola.com)

Abstract— **Hardware accelerators, used as application-specific extensions to the computational capabilities of a system, are efficient mechanisms to enhance the performance and reduce the power dissipation in a System On Chip (SoC). These accelerators execute on the computationally critical part of the application, and offload computations from the scalar processors. In this paper, we present a design automation tool that generates accelerators based on a given application kernel. The accelerators are processing streaming data, and support a programming model which can naturally express a large number of embedded applications, and which results in efficient and fast hardware implementations. We demonstrate the applicability of the tool for architectural space exploration for a number of media applications, with results on area, throughput, and clock speeds.**

I. INTRODUCTION

The levels of integration of modern FPGAs have advanced to a point where the performance and flexibility are sufficient to map all functions of a complex SoCs into a single die. FPGA manufacturers have embedded fixed functionality cores such as general purpose processors, multipliers, multi-ported SRAM memories, and DSP slices in order to speed-up commonly used applications. At the same time, tool vendors have offered a plethora of pre-defined peripherals, fixed IP functions, and even synthesizable processor cores for the designer to customize the chip. The availability of a tool flow that abstracts out the particular hardware structures and presents a software-only front end interface to the application developer is a necessary step to precipitate the acceptance of FPGAs as SoC platforms. Such a tool can be used by a larger pool of engineers, and not necessarily experts in system architecture and hardware design. Furthermore, an architectural automation tool should combine interactive architectural exploration, automatic hardware-software partition and an efficient mapping of one or multiple kernels to the reconfigurable fabric.

The programming model of this FPGA platform have the familiarity of high level programming language and the capability to efficiently map compute-intensive kernels onto the reconfigurable fabric. Typically, scalar processors are reasonably efficient in handling normal conditional code with a low degree of instruction and data level parallelism. They have been shown to be more efficient than the hardware generated by direct code mapping into gates [31]. However, scalar processors are very inefficient for high throughput, parallelizable code due to limited support of all kinds of parallelism (instruction, data, and task). They are further limited by the low memory bandwidth due to the narrow pipes to the main core.

We have developed an automation process which maps streaming data flow graphs (sDFG) to accelerators of the main scalar core. The streaming programming model assumes that the kernels process streams of data with a relatively limited lifetime, and deterministic memory access pattern. The streaming model decouples the description of memory access sequences from the computation within a kernel, thus making the customization of each of these two components (computation and memory access) easier and more re-usable.

The design space exploration involves an iterative design cycle in which Pareto-optimal implementations of a given sDFG are produced under user and system constraints. For each iteration, a search space iterator instantiates a set of parameters that meet the given constraints, then a scheduler produces a schedule of operations optimized for throughput, and finally an RTL generation back-end tool produces the hardware description of the accelerator. Separate hardware generation flows produce the computational units based on the scheduled sDFG and the stream units based on the stream descriptors. Each generated accelerator is synthesized and implemented on a Xilinx Virtex-4 FPGA to be evaluated in terms of stream throughput, area, and clock speed, and later classified as Pareto-optimal or eliminated from consideration.

The contributions of the papers are the following:

- first, we propose the usage of the streaming paradigm (sDFGs) for application acceleration in a SoC-based reconfigurable fabric.
- second, we propose a template-based, automated method to perform architectural exploration on the accelerated application by evaluating the design space separately for the stream unit and the stream computational unit.
- and third, we explain how these concepts are placed in the context of a bus-based SoC design and how the accelerators are connected to the rest of the system.

The rest of the paper is organized as follows: Section II gives background information on the streaming programming paradigm and explains how it exploits technology trends that favor computation over communication. Section III details our template-based methodology, and section IV presents a set of embedded applications and the results of the method on a Xilinx

Virtex-4 FPGA. Section V gives a summary of previous work on the relative areas, and Section VI presents the conclusion and future work.

II. Stream Programming Model

A. Architecture

The hardware accelerators that are generated by our method follow the streaming architectural paradigm. They act as filters on input streaming data to generate the output streaming data specified by the streaming data flow graphs. Stream kernels exhibit a large degree of data and task level parallelism, with regular or even statically defined communication patterns [1].

The regularity of data access and the short lifetime of the stream data allow for efficient optimization of both the communication and the computational portion of the algorithm. Even more importantly, they make possible the decoupling of the stream access from the computation and their separate optimization.

Under this model, memory load/store operations no longer need to be scheduled amongst compute operations and optimal scheduling of operations now does not depend upon memory latencies. With this independence, the underlying memory system may be changed or may exhibit variable latencies, as with caches, with no effect on the computation schedule.

The decoupled memory access allows data pre-fetching to occur during computation. It can be achieved with the programmer describing the shape and location of data in memory using stream descriptors, described in the next section. This decoupling allows the stream units to take advantage of available bandwidth to prefetch data before it is needed. The architecture becomes dependent on average bandwidth of the memory subsystem with less sensitivity to the peak latency to access a data element. In addition, the architecture benefits from having fewer stalls due to slow memory access.

Deep pipelining allows multiple functional units to be chained, reducing the access to large register files to store temporary data. This process is achieved with the programmer describing the data flow graph of the operations to be performed. Each operation is mapped to a set of functional units connected with a network. The number of functional units is dependent on the number of available logic gates, the number of potential parallel operations per cycle, and the user performance requirements.

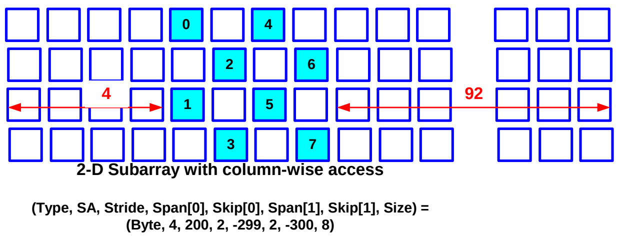

0
4
2
6
4
1
5
92
3
7
2-D Subarray with column-wise access
(Type, SA, Stride, Span[0], Skip[0], Span[1], Skip[1], Size) =
(Byte, 4, 200, 2, -299, 2, -300, 8)

Figure 1. Stream descriptors for a memory access pattern

B. Stream Descriptors

The architecture includes several independent stream units to prefetch data from memory and turn streams into FIFO queues of stream elements. Additional stream units are created to write stream elements into memory. Each unit handles all issues regarding loading/storing of data including: address calculation, byte alignment, data ordering, and memory bus interface.

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, Span0, Skip0, Span1, Skip1, Size), where:

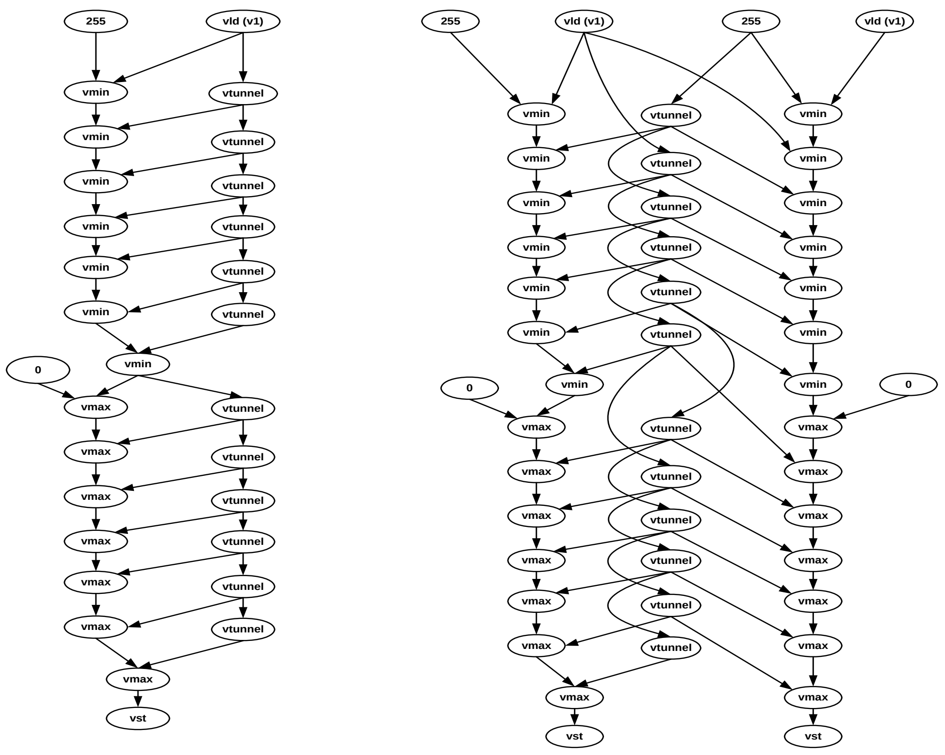
- Type indicates the element size in bytes (Type is 0 for bytes, 1 for 16-bit half-words, etc.).
- Start_Address represents the memory address of the first stream element.
- Stride is the spacing, in number of elements, between two consecutive stream element.
- Span0 is the number of elements that are gathered before applying the skip0 offset.
- Skip0 is the offset applied between groups of span0 elements, after the stride has been applied
- Span1 is the number of elements that are gathered before applying the skip1 offset.
- Skip1 is the offset applied between groups of span1 elements, after the stride and the Skip0 have been applied.

The Stride, Span, Skip, and Type fields define the shape of the data stream in memory, while Start_Address define the location of the first data element. The grouping and order in which data is accessed defines a Stream Record and corresponds to the desired alignment for the computation kernel. Multidimensional or even non-regular shapes can be created by extending the defined semantics of each stream descriptor. Figure 1 shows an example of a static memory access pattern described by a stream descriptor.

The stream descriptors and compiler manipulations are active research areas. Readers are referred to [1][8] for more details.

C. Stream Computation

The streaming paradigm allows the application to exploit the large number of functional units that are readily available in modern VLSI technologies without taxing the communication resources [12]. We are using a "Data-flow Graph" (DFG) language to express operations in a machine-independent manner. A DFG consists of nodes, representing basic arithmetic, and logical operations composing the vector operation, and directed edges, representing the dependency of one operation on the output of a previous



```
for (i=6; i < N; i++) {
     min = + ∞ ;
     for (j=0; j < 6; j++) {
        if (a[i-j] < min)
             min = a[i-j];
     }
     b[i] = min;
     max = - ∞ ;
     for (j=0; j < 6; j++) {
         if (b[i-j] > max)
              max = b[i-j];
     }

     c[i] = max;
}
```

Figure 2 The Open sDFG unrolled once and twice, and the C kernel of the Open filter

operation [8].

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 of the underlying design.

Each node in the DFG is denoted by a descriptor, which specifies:

- Input operands. The input operands are specified as relative references to previous nodes rather than named registers. This feature helps eliminate the unnecessary contention for named registers as well as the overhead associated with register re-naming.
- The operation to be performed by the node.
- The minimum precision of its output value. This can be derived from the precision of the input operands and from the operation performed by the node. However, implementations are allowed to use more precision if that is easier.
- The signedness of the output result.

Unlike the stream unit that accesses data from external sources like memory or peripherals, the production and consumption of intermediate results is done locally, within the accelerator.

We will use the sDFG of Figure 2 in the rest of the paper to illustrate the concepts and trade-offs involved in our methodology. This sDFG implements the open morphological filter which is the basis of a many image processing applications such as edge detection [23]. The open filter is a non-linear operation on each pixel located in *(r,c)* and is defined as an erosion *e* followed by a dilation *d* on a grayscale image I:

$$e(r,c) = MIN_{i \in D1,\, j \in D2}[I(r+i, c+j)]$$
$$d(r,c) = MAX_{i \in D1,\, j \in D2}[I(r+i, c+j)]$$
$$Open(r,c) = d(e(r,c))$$

where D1xD2 is the window that defines the filter applied on the image pixels. The equivalent C code of the unrolled kernel is also shown in Figure 2.

III. Template-Based Hardware Generation

The problem we are addressing in this paper is the automatic generation of synthesizable accelerators from the streaming representation of Section II. Our approach is to select designs from a well-engineered framework, instead of generating the given hardware from a generic representation of a high level language. We generate highly optimized designs at various points at the cost-performance space based on the given application, the user requirements, and the capabilities of the rest of the system.

Figure 3 shows the iterative design flow. The main points of the tool flow are the following:

- a common template based on a regular architecture that accesses and processes streaming data,
- an iteration engine that instantiates system parameters that meet system and user constraints to initiate the next iteration of space search,
- a scheduler that performs sDFG scheduling and hardware allocation based on the parameters set by the iterator,

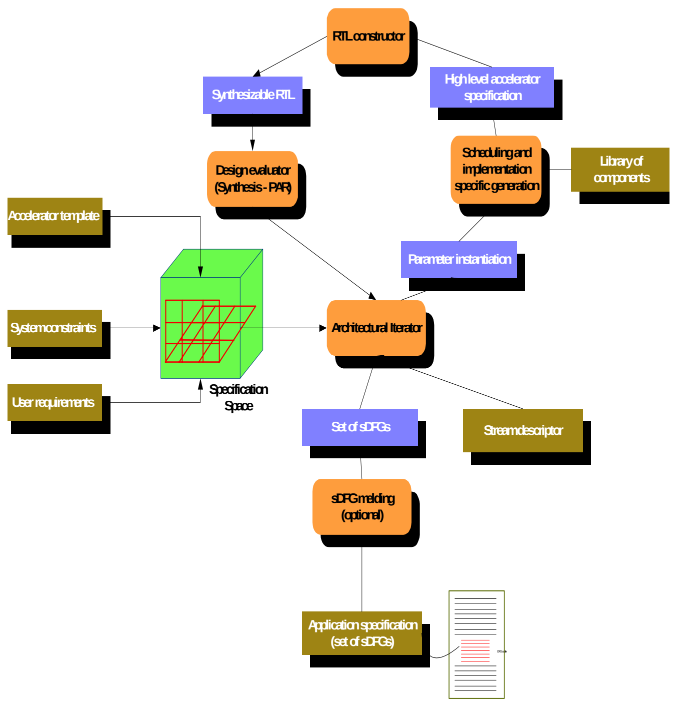


Figure 3 Template-based accelerator generation

- an RTL constructor engine that produces optimized Verilog code for the data path and the stream units,
- and an evaluation phase that synthesizes and maps the designs in FPGA and produces quality metrics such as area, and clock speed

Each of the data path and the stream unit have their own acceleration generation process. The rest of the section details each one of these engines and their interfaces

A. *Architectural template*

The architectural template consists of two parts: the streaming data path and the stream unit (Figure 4). The stream unit expands into one or more input and output stream modules, and is generated to match the characteristics of the stream descriptors, and the characteristics of the bus-based system and the streaming data path. The data path is generated to execute a given sDFG to match user and system constraints in the specification space.

Stream Unit

The stream unit transfers streams from a system memory or peripheral, through a system bus and present them in-order to the accelerator. It also transfers processed output streams back to the memory.

The stream queue and the alignment unit store the incoming stream data and present them to the data path in-order. The number of storage elements, their size, and their interconnect depend on the stream descriptors and the requested bandwidth of the data.

The peak bandwidth for the accelerator depends on the schedule of the sDFG as we will discuss later. The size of the storage elements matches the size of the stream elements, for example it can be one byte for 8-bit pixel data. Finally, the interconnect between the storage elements and the flow of streaming data between them depends on the span and skip of the stream description.

As an example, suppose that the sDFG of Figure 2 is unrolled twice and is scheduled so that the peak bandwidth of the input stream module is two bytes per cycle. Such a case arises when, for example, an Open filter is applied to a tiled image for edge detection. Under these assumptions, the stream queue should have at least two 8-bit pipelined registers to meet the bandwidth requirements. There is no need for any feedback connections from the head to the tail of the queue because the stream elements are not reused. The stream unit hardware generator detects this lack of reuse by examining the value of the skip0 and skip1 parameters, which are non-negative.

As we will examine later, the space iterator may also decide to allocate extra registers to the stream queue to match the system bus bandwidth capabilities. For example, in the case of an 8-byte PLB bus, the stream queue can have 8 or more storage elements to exploit the spatial locality of the memory accesses.

The bus line buffer is used to temporarily hold the data accessed from the system bus, and filter them to the stream queue when there is enough space. Two pointers, head and tail, follow the production and consumption of streaming elements in the stream queue and produce information on the emptiness and the fullness of the queue. By detecting cases where the stride is greater than 1, the bus line buffer eliminates unnecessary elements before sending the stream to the stream queue.

The address generation unit (AGU) is hardwired to generate the memory access pattern of the stream descriptors. The number of registers that store internal variables (e.g. span_left[i]), their width, the value and size of the stream description parameters are some of the configuration mechanisms of this unit.

The AGU aggressively generates addresses for data prefetching and sends them to the Address Line Buffer module. This module stores the addresses, merges addresses that fall within the same bus word (or burst size word in case bus burst is enabled), and competes for bus accesses with the other stream units. The generated number of buffers in the Address Line Buffer matches the average latency of the memory and bus systems and the capability of the bus to pipeline data accesses. For example, the PLB bus used in the Virtex architecture can pipeline up to two read accesses to a memory location, and, therefore, an efficient Address Line Buffer will have at least two address buffers.

Finally, the arbiter regulates the access of the stream units to the system bus. It uses a round-robin algorithm, and its complexity depends on the number of input and output streams of the sDFG.

Data path

The data path template of Figure 4 is an interconnect of reconfigurable functional units that produce and consume

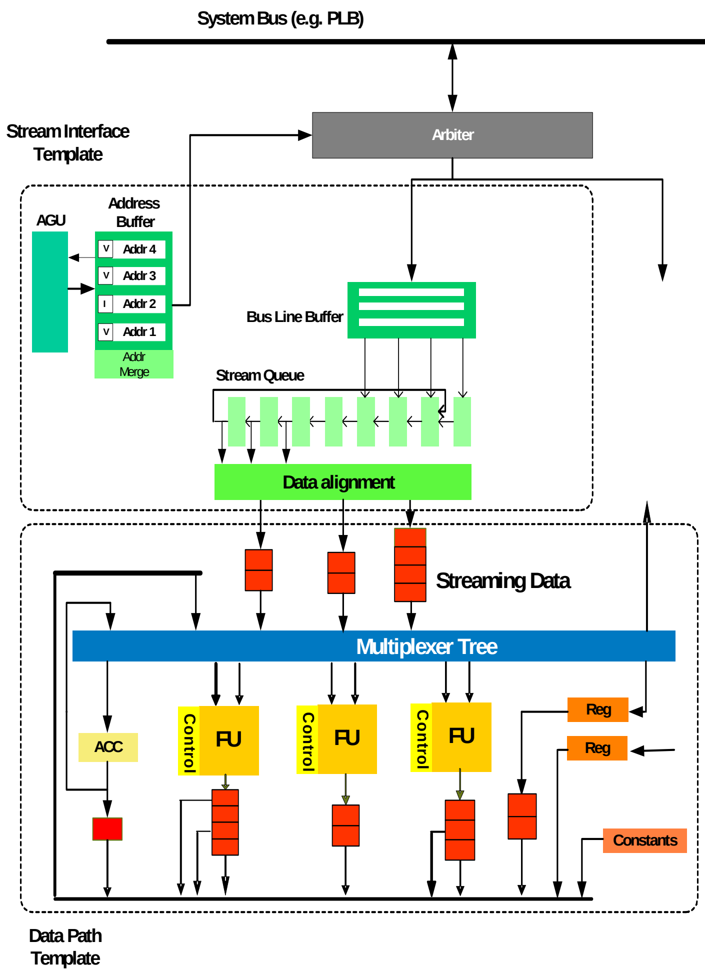


Figure 4. The accelerator template consists of the Data Path and the Stream Unit templates. Different optimizations are deployed in each case.

streaming data, and communicate via reconfigurable links. The links are chained at the output of a slice of a functional unit, and have a single input and potentially multiple outputs. They implement variable delay lines without the need of an explicitly addressable register file. The template also allow for the usage of a set of named registers that can be used by the sDFG to pass values from one sDFG iteration to the next and implement cross-iteration dependencies, and also to pass parameters to the program. Furthermore, the programming model allows for the use of accumulators for reduction operations [8].

The control logic of the data path is distributed and spatially close to the corresponding functional unit, multiplexer or line queue. This was an explicit design decision to avoid creating critical paths due to long wires in modern VLSI technologies.

The type of the functional units (ALUs, multipliers, shifters, etc.), the specific operation performed within a type (e.g. only addition and subtraction for an ALU), the width of the functional unit, the size and number of storage elements of a FIFO, the interconnects between functional units (via FIFOs), the bandwidth from and towards the stream units are some of the reconfigurable parameters of the data path.

The data path requests data-sourcing from the input stream module and data-sinking from the output stream module. A simple, demand-driven protocol between the two modules is used to implement the communication. Stall signals from the stream units to the data path allow for a less than perfect memory system. A stall signal from any stream unit will cause the stall of the accelerator engine.

The accelerator will compute for as long as valid streaming data are coming and not all of the outgoing streaming data have been produced. Coupled with each input from a stream unit to the data path is a *Valid* bit that notifies when a stream element of data is valid or not. A *Done* signal is asserted when a stream has transferred all of its data to the accelerator.

B. Architectural Iterator

The iterator selects a set of parameters in the space specified by the user and the system. For each one of this set of parameters, the tool flow builds an implementation by

Table 1a. User constraints and example values

Number of Functional Units per Type	Available bits for each FU instance	Minimum allocation slice	Execution latency for each FU	Pipeline Initiation Cycle for each FU
4 input stream ports (for all streams) 1 ALU unit 1 Multiplier unit	192 bits for each ALU 64 bits for each multiplier	16 bits for ALUs 32 bits for multipliers	One cycle for ALUs One cycle for multipliers	One cycle for all FUs

Table 1b. System constraints and example values

Area (in gates or slices)	Peak bus bandwidth	Average memory latency	Bus pipeline factor
3000 slices	64 bits/cycle	2-30 cycles	2 pending requests

breaking the task into the implementation of the data path and the implementation of the stream unit.

Scheduling and High Level Implementation

The scheduler receives as input the sDFG along with the user and system constraints (Table 1a and Table 1b) 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. The outline of the scheduling algorithm is given in Figure 5. The output of this stage is a hardware representation of the accelerator at a higher specification level than an RTL specification. We will omit the description of this High level Model (HLM) of the data path of the accelerator for brevity. Figure 5 is the flow graph of the scheduling process of the data path.

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 [20]. If R_k is the number of resources of type k available in the system, and N_k is the number of operations that use a functional unit of type k in a loop iteration, then the lower bound for the MII is $\max_{1 \le k \le m}\left(\left\lceil \frac{N_k}{R_k} \right\rceil\right)$, where m is the number of available types. As an example, the sDFG of Figure 2 requires 14, 16-bit min/max nodes per iteration. If there is an available ALU with 192 bits, then the minimum MII is $\left(\left\lceil \frac{192/16}{14} \right\rceil\right) = 2$ cycles.

Data dependencies that cross iteration boundaries also constraint the MII as follows: if there is a cross-iteration data dependency between nodes n_c of iteration 1 and n_1 of iteration 2 such that: $n_1(1) \rightarrow n_2(1) \rightarrow \ldots \rightarrow n_c(1) \rightarrow n_1(2)$, the nodes n_i of successive iterations should be scheduled at least C cycles apart assuming that the latency of each node is one cycle. This situation arises when there is a cycle in an sDFG, due to a back-edge $n_c(1) \rightarrow n_1(2)$.

In general, the MII is bounded by $\max_{circlei}\left(\left\lceil \frac{Ci}{Di} \right\rceil\right)$ for each circle in the sDFG. The C_i is the sum of the execution latencies of all nodes in the circle, and D_i is the sum of all iteration distances of edges in the circle. The Open filter of Figure 2 has no cycles, and, thus, no cross-iteration constraints.

During scheduling, the interconnects are not counted as resources, but rather they are "filled-in" during the generation of the HLM. By setting the schedule period equal to the MII, the scheduler maximizes the throughput of the accelerator, which is the main optimization target of the tool flow. Next, 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.

Then, a backward scheduling heuristic is used to re-schedule some of the nodes by scheduling from the step MII-1 towards the step 0. This, in effect, "spreads out" the nodes within the steady-state period of the schedule and distributes the schedule more evenly within the MII steps. The net effect of this approach is to reduce the latency between successive nodes in the schedule, thereby reducing the storage requirements of the line delay queues [20].

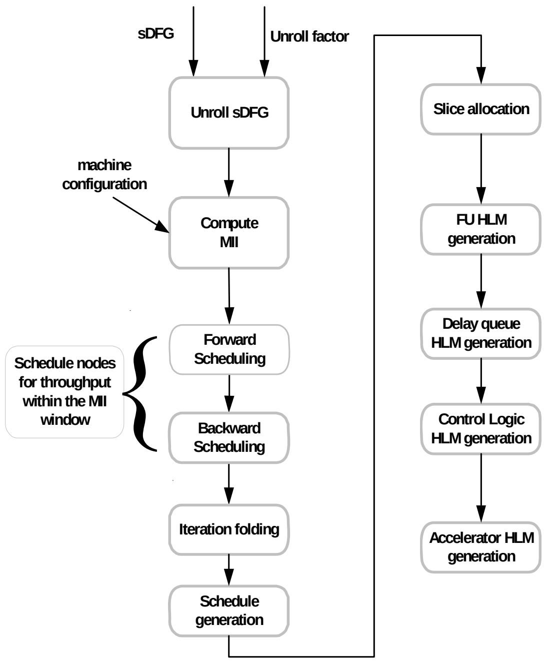


Figure 5. Scheduling and high level model (HLM)

Table 2. Application Benchmark Details

Benchmark Name	Number of nodes	Description
Open filter	30	Morphological filter used for edge detection
Edge detection Filters	67	A graph that combines all the filters used in an edge detection system (open, close and binarize)
Row DCT	95	Row-wise Discrete Cosine Transform filter
Color processing	232	Three filters that perform a 3x3 low pass filtering, followed by gamma correction, followed by RGB to YUV color space transformation. Used as part of a digital camera image finishing chain

The scheduler needs to only generate code for the steady state body of the schedule and not for the prologue and epilogue as is often the case in modulo scheduling [26]. Each data token that populates the FU 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 vld) produces data with valid bits when the data are available, and a sink operation (like a vst) accepts data only when they are valid.

For example, the schedule of the vector-add operation of two N-element vectors of Figure 6 can be expressed by a single word that combines all four operations. The colored operations process invalid data and are don't cares for a particular cycle. The valid bits ensure the correct execution of the code in the first two and the last two cycles of the schedule.

Next, the tool flow binds the operation nodes to the functional unit slices, and generates the delay links at the output of each slice to store the streaming outputs as they are produced by the FUs. Finally, the control logic for each one of the data path elements is produced based on the schedule. The back end portion of the scheduler applies bitwidth optimizations to the operations to reduce the area and increase performance.

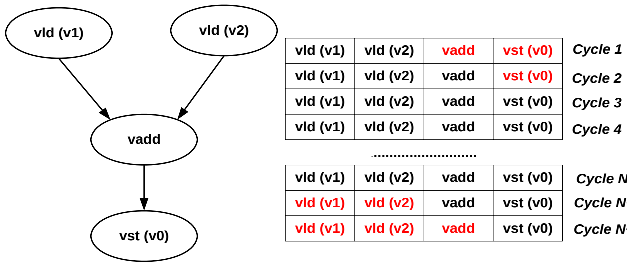

vld (v1)
vld (v2)
vadd
vst (v0)
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle 1
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle 2
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle 3
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle 4
....
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle N
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle N
vld (v1) | vld (v2) | vadd | vst (v0) | Cycle N

Figure 6. Valid bits allow the scheduler to produce only the steady-state code

The stream unit design is generated based on user and system constraints, as shown earlier in Table 1a and Table 1b. 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 bandwidth to enable burst transfers. In addition, the number of stream data queue (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. These settings are chosen so that data continues to stream into and out of the stream unit without stalling the bus or data path.

C. *RTL constructor*

The RTL constructor reads the HLM representation and emits structural Verilog for the data path and the stream unit.

D. *Evaluator*

At that point, the evaluation process is done by passing the resulting Verilog code through the Xilinx ISE tool-flow. We synthesize, and map the design targeting a Xilinx Virtex-4 architecture. We evaluate the design in terms of clock speed, and area overhead. As we will examine in the experimental evaluation, we are able to produce high-quality accelerators both in terms of area, and clock speed.

IV. Experimental Evaluation

A. *Methodology*

This section describes the evaluation of the design methodology presented in previous sections. An application set, shown in Table 2, is selected from a wide range of media applications related to video compression, color processing, and image processing. Key compute intensive kernels from this application set is chosen for implementation. In all cases, these benchmarks are first coded as sDFG, as shown as an example in the Figure 2.

A design automation tool, using the design flow shown previously in Figure 3, is implemented in C++. The tool accepts each sDFG in the application set to generate candidate hardware accelerators according to the template shown in Figure 4. Different architectural configurations and loop unrolling factor are chosen such that Pareto-optimal designs for each benchmark can be chosen.

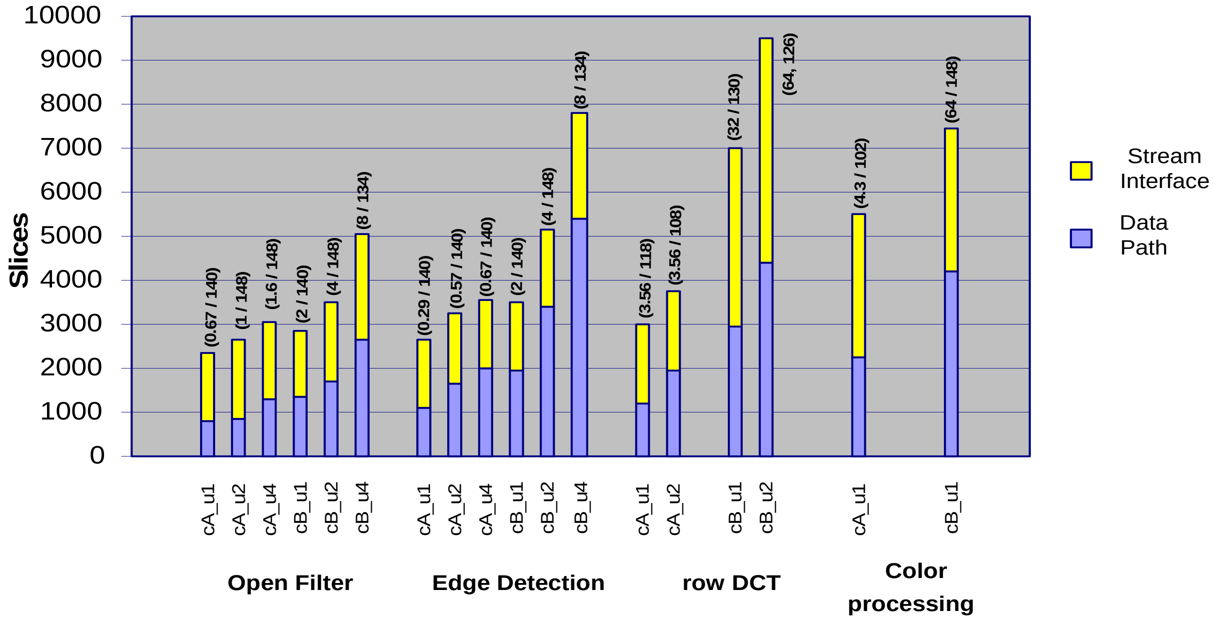


Figure 7. Synthesis results for hardware accelerator

Figure 7 shows the synthesis results for the benchmarks under several configurations. Each configuration is described by a pair of parameter (c_i,u_i). The c_i parameter refers to user constraints in terms of maximum number of computational resources that the tool can utilize to schedule the sDFG. For this experiment, c_B corresponds to a very wide configuration with an unlimited number of functional units while c_A corresponds to the intermediate configuration with fewer functional units, similar to the RSVP-2™ accelerator [8].

The u_i parameter shows the degree of unrolling for the sDFG to achieve higher throughput. In wider configurations, sDFG unrolling can be an effective means to use resources that could otherwise remain unused. However, a higher degree of unrolling can strain the bus and memory subsystem, which may result in lower or negligible speed up.

The generated hardware is synthesized and mapped onto a Xilinx Virtex-4 FPGA, and the quality metrics of the produced bitstream (area, clock frequency) are recorded to assess the Pareto-optimality of the design

B. *Discussion*

The results of Figure 7 show the total number of FPGA slices for each configuration of each benchmark, and how the slices are distributed among the data path and the stream interface. The average I/O bandwidth in bytes per cycle between the data path and the stream interfaces, and the clock frequency in MHz after synthesis are also indicated at the top of each bar. The I/O bandwidth shown is an upper limit on the achievable bandwidth between the accelerator and the external bus. Some configurations are not shown because they require very large amount of I/O bandwidth because of the large unrolling factor.

These results enforce our initial premise that template-based approach can produce fast and area efficient designs. Using a high level representation such as the sDFG allowed for quick architectural exploration of different configurations (c_i,u_i). The streaming programming model facilitates the selection of sDFG selection and coding. Furthermore, it allows for design optimizations of both stream and data path, without recoding the benchmark.

In general, wider designs require more resources because the template design requires a larger number of queuing elements at the output of each functional unit to store live variables at each cycle. On the other hand, wider configurations are faster due to the lack of large multiplexers at the inputs of the functional units. In all cases, the maximum clock frequency is determined by the stream interface unit which is slower than the data path.

V. Related Work

In this section, we discuss previous work in the areas of streaming programming model and streaming architectures, architectural automation for ASIC and FPGA design flows, and special reconfigurable architectures.

A. *Streaming Programming Model and Architectures*

Using streaming representations to expose concurrency and to express data communication explicitly has been recognized as an efficient way to both program off-the-shelf parallel processors, like graphics chips, and to architect new processors. A thorough analysis of the streaming programming model is given in [1], and an example of a language (Brook) that describes explicit streaming model concepts is detailed in [4]. Example streaming processors include Merrimac [12], Raw [30], Cell [19], and RSVP™ [8].

B. Architectural automation for ASICs and FPGAs

There has been an intense interest in the research community in the last decade to automate the architectural process for ASIC of FPGA tool flows starting from a high level representation like C, Java, Matlab, DFGs and so on [10][13]. The following list is a non-exhaustive selection of related projects in academia and industry.

The PICO project incorporated a lot of concepts from earlier work on VLIW machines, and described a methodology to generate a VLIW engine along with an accelerator optimized for a particular application [21][28]. Similar projects include the Cyber tool [32], the OCAPI [29], the DEFACTO compiler [35], the ASC streaming compiler effort [24] in which new explicit constructs have to be added to the C language to express streaming parallelism, the CASH compiler that maps the complete the C application onto asynchronous circuits [31], the Streams-C compiler 15], and work on Single Assignment C mapping to gates [27]. The Impulse-C [25] and Handel-C [37] are efforts to utilize C with extensions as a high level RTL language for FPGA design. At an even higher level of abstraction, the Matlab to gates compiler from AccelChip [3] targets mainly DSP kernels on FPGA platforms. In some cases, a domain specific language is used to map high level abstractions to gates for a particular application domain, such as networking [22].

Most of the above mentioned approaches use C as a more "user-friendly" hardware description language, and they add constructs to enhance concurrency, variable bitwidth, and so on in order to make C more amenable to hardware design. We believe that a template-based architectural automation that evaluates a large number of potential designs and focus on the most "profitable" parts of the code is able to offer both design efficiency in terms of speed and cost, as well as programmability for developers that are not well-versed in hardware design.

A related problem is to automatically detect clusters of heavily executed assembly-level instructions that can be merged and extend the ISA of the processor. This research extends to both an ASIC environment [9] for the ARM processor, and an FPGA environment for the Altera Nios processor [11]. Tensilica [39] and ARC [36] are offering the opportunity to extend their base architecture with tightly-couple processors and to automatically generate compiler extensions so that calls to these coprocessors can be embedded in the application.

C. Reconfigurable processors

A number of academic projects and commercial products are tackling the hardware synthesis problem by designing efficient compile-time configurable or run-time reconfigurable architectures. This effort also stems from the fact that off-the-shelf, commercial FPGA architectures have little if any support for run-time reconfiguration. The GARP [5] and SCORE projects [6][7] propose the addition of reconfigurable planes that act as coprocessors of a scalar processor (MIPS in the case of Garp). Multiple planes offer a very fast context switch mechanism for run-time reconfiguration, and allows for virtual compute pages that can be mapped on the fabric both spatially and temporally.

Other projects include the RaPid architecture [14], the Chimaera architecture [34], the Piperench architecture [16], the RAW/Virtual Wires research [2], the Amalgam project [18] and so on.

Embedding reconfigurable fabric into an otherwise ASIC design, also called embedded FPGAs, is a recent trend in many commercial offerings. Stretch, Inc. [17] uses the Tensilica toolset to generate tightly couple co-processors and map them on the Instruction Set Extension Fabric (ISEF). Similar ideas come from several start-ups such as IPFLex, PACT/XPP [38], SiliconHive, etc.

Our work is not related to any such proprietary architecture, rather it produces hardware code that can target any fabric that understands the RTL representation.

VI. Conclusion and Future Work

A design methodology and prototype tool to automate the design and architectural exploration of hardware accelerators are described in this paper. These accelerators are programmed as streaming kernels to map to the streaming accelerators. In comparison to other approaches, we utilize a well-engineered template to enable fast convergence to an area and speed efficient design. We show how this methodology is used for an application set with various architectural configurations. New streaming accelerators are generated without recoding the application or re-design of the platform.

References

[1] Amarasinghe S., Thies B. Architectures, Languages and Compilers for the Streaming Domain. Tutorial at the 12th Annual International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA

[2] Babb J., et. al. Parallelizing Applications into Silicon. Proceedings of the 7th IEEE Symposium on Field Custom Computing Machines (FCCM), April 1999, Napa Valley, CA

[3] 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

[4] Buck I. Current Brook specification (0.2). http://merrimac.stanford.edu/brook, October 2003

[5] Callahan T., Hauser J., Wawrzynek J. The Garp Architecture and C Compiler. IEEE Computer Magazine, vol. 33, no. 4, April 2000, pp. 62-69

[6] Caspi E., Huang R., Yeh J., Markovskiy Y., DeHon A., Wawrzynek J. Stream Computations organized for Reconfigurable Execution (SCORE): Introduction and Tutorial. BRASS research group technical report, University of California, Berkeley, August 2000

[7] Caspi E., DeHon A., Wawrzynek J. A Streaming Multithreaded Model. Proceedings of the 3rd Workshop on Media and Stream

Processors (MSP), in conjunction with the 34th International Symposium on Microarchitecture, December 2001, Austin, TX

[8] Chirisescu S., et. al. The Reconfigurable Streaming Vector Processor, RSVP™. Proceedings of the 36th International Conference on Microarchitecture, December 2003, pp. 141-150, San Diego, CA

[9] Clark N., Zhong H., and Mahlke S. Processor Acceleration Through Automated Instruction Set Customization. Proceedings of the 36th International Symposium on Microarchitecture, December 3-5, 2003, pp. 129-140,San Diego, CA

[10] Compton K., Hauck S. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, vol. 34, No. 2, June 2002, pp. 171-210

[11] Cong J., Fan Y., Han G. and Zhang Z. Application-Specific Instruction Generation for Configurable Processor Architectures. Proceedings of the 12th International Symposium on FPGAs, February 2004, pp. 183-189, Monterrey, CA

[12] Dally W. J., Hanrahan P., Erez M., Knight T. J., Labonté F., Ahn J.H., Jayasena N., Kapasi U. J., Das A., Gummaraju J., Buck, I. Merrimac: Supercomputing with Streams. Proceedings of the 2003 Supercomputing Conference, November 2003, pp-35-42, Phoenix, AZ

[13] De Micheli G. Hardware Synthesis from C/C++ models. Proceedings of the conference on Design, Automation and Test in Europe (DATE), March 1999, pp. 382-383, Munich, Germany

[14] Ebeling C., Cronquist D., Franklin P., Secosky J., Berg S. Mapping Applications to the RaPiD configurable architecture. Proceedings of the 5th IEEE Symposium on Field Custom Computing Machines (FCCM), April 16-18, 1997, pp. 106-115, Napa Valley, CA

[15] Gokhale M., Stone J., Arnold J., Kalinowski M. Stream-Oriented FPGA computing in the Streams-C High Level Language. Proceedings of the 8th IEEE Symposium on Field Custom Computing Machines (FCCM), April 17-19, 2000, pp. 39-48, Napa Valley, CA

[16] Goldstein S. C. et. al. PipeRench: A Reconfigurable Architecture and Compiler. IEEE Computer Magazine, vol. 33, no. 4 April 2000, , pp. 70-77

[17] Gonzalez R. Software Configurable Processors Change System Design. Hot Chips XVII, August 15-16, 2005, Palo Alto, CA

[18] Gottlieb D. B., Cook J. J., Walstrom J. D., Ferrera S, Wang C. W., Carter N. P. Clustered Programmable-Reconfigurable Processors. Proceedings of the 1st IEEE International Conference on Field Programmable Technology (FPT), December 2002.

[19] Gschwind M., Hofstee P., Flachs B., Hopkins M., Watanabe Y., Yamazaki T. A novel SIMD architecture for the Cell heterogeneous chip-multiprocessors. Hot Chips XVII, August 15-16, 2005, Palo Alto, CA

[20] Hwang C. T., Hsu Y. S., Lin Y. L. PLS: A Scheduler for Pipeline Synthesis. IEEE Transactions of Integrated Circuits and Systems, vol. 12, no. 9, September 1993, pp. 1279-1286

[21] Kathail V., Aditya S., Schreiber R., Rau B.R., Cronquist D., Sivaraman M. PICO: Automatically Designing Custom Computers. IEEE Computer Magazine, vol. 35, no. 9, September 2002, pp. 39-47

[22] Kulkarni ., Brebner G., Schelle G. Mapping a Domain Specific Language to a Platform FPGA. Proceedings of the 41st Design Automation Conference (DAC), pp.924-927, San Diego, CA

[23] Lee J., Haralick R., Shapiro L. Morphological Edge Detection. IEEE Journal of Robotics and Automation, vol. 3, issue 2, April 1987

[24] Mencer O., Pierce D. J., Howes L.W., Luk W. Design Space Exploration with a Stream Compiler. Proceedings of the IEEE International Conference on Field Programmable Technology (FPT), December 2003, Tokyo, Japan

[25] Pellerin D., Thibault S. Practical FPGA Programming in C. Prentice Hall, 2005

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

[27] Rinker R., Carter M., Patel A., Chawathe M., Ross C., Hammes J., Najjar W., Bohm W. An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 9, no. 1, February 2001, pp. 130-139

[28] Schreiber R., Aditya S., Mahlke S., Kathail V., Rau B.R., Cronquist D., Sivaraman M. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. HP Labs Technical Report HPL-2001-249, October 5th 2001

[29] Schaumont P., Vernalde S., Rijnders L., Engels M., Bolsen I. A programming environment for the design of complex high speed ASICs. Proceedings of the 35th Design Automation Conference (DAC), June 1998, pp. 315-320, San Francisco, CA

[30] Taylor M. B., et. al. The RAW Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro Magazine, 22(2), March 2002, pp.25-35

[31] Vidiu M., Venkataramani , Chelcea T., Goldstein S.C. Spatial Computation. Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 9-13, 2004, pp. 14- 26, Boston, MA

[32] Wakabayashi K. and Okamoto T. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. IEEE Transactions on Computer-Aided Design, 19(12):1507-1522, December 2000

[33] Wulf W. A., McKee S. A. Hitting the memory wall: implications of the obvious. ACM SIGARCH Computer Architecture News, Vol. 23, no. 1, March 1995, pp. 20-24.

[34] Ye A. Z., Moshovos A., Hauck S., Banerjee P. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit. Proceedings of the 27th International Symposium on Computer Architecture (ISCA), June 2000, pp. 225-235, Vancouver, BC.

[35] H. Ziegler H., Hall M. Evaluating Heuristics in Automatically Mapping Multi-Loop Applications to FPGAs Proceedings of the 13th International Symposium on FPGAs, February 2005, pp. 184-195, Monterey, CA

[36] Architect white Paper, www.arc.com

[37] Celoxica Corporation, Handel-C language reference manual, www.celoxica.com

[38] PACT Debuts Extreme Processor. Microprocessor Report, October 9th, 2000

[39] Automated Configurable Processor Design Flow, White Paper, www.tensilica.com

[40] Virtex-4 FPGA handbook, www.xilinx.com, August 2004