Optimus: Efficient Realization of Streaming Applications on FPGAs

Amir Hormati, Manjunath Kudlur, Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{hormati, kvman, mahlke}@umich.edu

David Bacon, Rodric Rabbah
IBM T.J. Watson Research Center
Hawthorne, NY
{bacon, rabbah}@us.ibm.com

ABSTRACT

In this paper, we introduce Optimus: an optimizing synthesis compiler for streaming applications. Optimus compiles programs written in a high level streaming language to either software or hardware implementations. The compiler uses a hierarchical compilation strategy that separates concerns between macro- and micro-functional requirements. Macro-functional concerns address how components (modules) are assembled to implement larger more complex applications. Micro-functional issues deal with synthesis issues of the module internals. Optimus thus allows software developers who lack deep hardware design expertise to transparently leverage the advantages of hardware customization without crossing the semantic gap between high level languages and hardware description languages. Optimus generates streaming hardware that achieves on average 40x speedup over our baseline embedded processor for a fraction of the energy. Additionally, our results show that streaming-specific optimizations can further improve performance by 255% and reduce the area requirements by 16% in average. These designs are competitive with Handel-C implementations for some of the same benchmarks.

Categories and Subject Descriptors

B.5.1 [**Register-transfer-level Implementation**]: Design—*Data Path Design*; B.5.2 [**Register-transfer-level Implementation**]: Design Aids—*Automatic synthesis*

General Terms

Design, Performance

Keywords

Streaming, Compiler, FPGA, Optimization, Heterogeneous, Embedded System

1. INTRODUCTION

In the world of embedded systems, there are many devices that offer increasingly powerful computing capabilities. It is predicted that mobile computing devices with embedded processors will ultimately change the industry much as laptops supplanted desktops as the primary commodity processing platform. However, the power and frequency concerns that plague the microprocessor industry effectively mean architects have to find new ways to provide increasing performance since conventional frequency scaling methodologies no longer apply. As a result, there is a significant opportunity to explore alternate architectures that can enable the next evolutionary step in computing.

One significantly promising approach is to provide automatic customization of hardware according to the applications they run. An application-customized architecture can offer extremely high performance with very low power compared to a more general-purpose design. Furthermore, the increasing availability of reconfigurable field-programmable gate arrays (FPGAs) as co-processors and processing ingredients in heterogeneous systems-on-a-chip [1, 2] means emerging architectures can offer enormous flexibility and adaptability in the face of rapidly changing software standards and customer needs.

This paper describes a methodology and a set of complementary optimizations to efficiently realize stream graphs directly in hardware. Our ultimate goal is to automatically refine a high level stream program into either software or hardware. In the case of the former, a program can run on a conventional processor or a multicore architecture. In the case of the latter, the application is realized as an efficient customized circuit design mapped onto FPGAs.

The emphasis on stream programs is self-evident as recent years have witnessed the proliferation of embedded streaming applications in many areas including digital signal processing, graphics, multimedia, network processing, and encryption. There are several new streaming languages and the area currently commands considerable attention from academia and industry. The stream programming paradigm offers a promising approach for programming multicore architectures. Examples of relatively new streaming languages include StreamIt [24], Brook [4], CUDA [20], SPUR [27], Cg [16], Baker [6], and Spidle [7].

We adopt a stream programming model where applications can be naturally described as dataflow graphs where nodes embody computation and edges imply communication. Such a streaming model is attractive from a multicore perspective because it makes the abundant parallelism inherent to streaming applications quite explicit. As a result, compilers can more readily derive concurrent implementations from high level applications, with relatively less effort compared to automatic parallelization starting from imperative sequential languages such as C [23, 9, 14]. In the same way, mapping a high-level stream program to hardware (e.g., FPGAs) becomes more practical and productive—compared to using a hardware description language such as Verilog or VHDL, or HDL

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-469-0/08/10 ...$5.00.

derivatives of C such as SystemC or Handel-C—if a compiler can readily generate efficient hardware implementations from the programs described in a streaming language.

The idea of mapping high level programs directly into hardware is not a new one. Indeed, there is a lot of work on automatic synthesis of hardware starting from C and its many HDL-oriented derivatives. This work differs from most existing work on the topic of high level synthesis (Section 2) by shifting the focus from micro-functional details to macro-functional ones. Specifically, our work does not focus so much on how individual modules are synthesized (i.e., micro-functional), but rather on how modules are composed to assemble an overall design (i.e., macro-functional). As a result, we can synthesize entire applications into a hardware substrate, and not just individual loops and kernels as is the case with a lot of existing work. Thus, our work is complementary to existing work on high level synthesis while offering new opportunities for efficient assembly of streaming applications in hardware.

This paper describes Optimus, our optimizing synthesis framework for streaming applications. Optimus uses a canonical intermediate representation to describe streaming programs. A program is comprised of interconnected *filters*, derived from the dataflow graph representation of the program. Each filter is comprised of *blocks* that contain statements. The blocks are themselves interconnected based on control and dataflow dependences. Our set of optimizations that deal with inter-filter details address macro-functional concerns. Similarly, our micro-functional optimizations address synthesis issues that arise from dataflow dependences between blocks. The Optimus model allows us to leverage decades of classic compiler research studied by others in their work to generate high-quality circuits, while also offering the ability to apply macro-functional optimizations that are specifically targeted for streaming applications. Macro-functional optimizations, which address how filters (modules) are assembled to implement an application tend to be tedious and time-consuming to perform manually, and require expertise in hardware design. An important example of a macro-functional optimization is deciding on how much buffering to allow between a pair of communicating modules: if too little buffering is provided, then throughput decreases as modules stall to send or receive data; whereas too much buffering incurs substantial space overheads. Macro-functional optimizations require careful consideration of area and performance tradeoffs to judiciously maximize application throughput at the lowest costs.

Our results (Section 5) using eight streaming benchmarks, including FFT, DCT, DES, sorting, and matrix multiplication, show that we can achieve significant performance advantages compared to an embedded processor for a fraction of the energy. It is not surprising that a custom hardware design is better than a general-purpose processor. We also found that Optimus-generated designs are performance-competitive and incur small area overhead in comparison to some of the benchmarks that we also implemented in Handel-C.

The primary emphasis of this paper is on the salient macro- and micro-functional optimizations for streaming programs. We use the StreamIt programming language as our input language although other languages that embody the same streaming model are equally applicable. Optimus compiles StreamIt programs to Verilog. We then use standard synthesis tools to generate FPGA designs. Optimus uses its own hardware models to characterize space-time tradeoffs, and performs many optimizations including critical path balancing and memory allocation. It is built on top of the Trimaran compiler [25], and hence it inherits a rich suite of ILP optimizations (for micro-functional efficiency). The compiler also admits profile-guided optimizations to simplify circuit models for streaming applications. Profiling data provides a cheap and practical alternative to otherwise difficult and intractable optimization problems. The core optimizations are described in Section 4, and Section 3 describes our overall stream-oriented synthesis framework with both macro- and micro-functional emphasis.

2. RELATED WORK

C is closely linked to the Von Neumann processor model, in which variables correspond to memory locations and function invocations reside on stacks. C lets users manipulate pointers to memory and to functions, which does not make sense in an FPGA circuit model. Thus, any attempt to compile C to FPGA configurations would encounter problems that derive purely from the C language, not from the the application itself. Several projects have tried to address the inadequacies of C with different techniques.

As a result of an extensive amount of research in the area of high-level synthesis, researchers have introduced several compiler systems and abstraction languages [13, 18, 21, 17, 12, 8, 3, 11, 10] each of which has some unique capabilities. ROCCC [10] is a C to hardware compilation project whose objective is the FPGA-based acceleration of frequently executed loop nests. This compiler performs extensive compile-time transformations to maximize various forms of parallelism and minimize the number of off-FPGA memory accesses. Circuits generated by ROCCC can be used by Optimus as IP blocks to accelerate the execution of loop nests. Another C to hardware compiler is SPARK [11], which takes a subset of C as input and outputs synthesizable VHDL. Its optimizations include code motion, variable renaming, FSM state minimization, etc. Streams-C [8] relies on a CSP model for communication between processes and can meet relatively high-density control requirements. Researchers in academia and industry have also designed various high-level abstraction languages such as SA-C [19], Handel-C [5], SystemC [22], etc., to make designing hardware systems easier for average software developers. SA-C helps compilers exploit data reuse because of its special constructs (e.g., windows) and it functional nature. Handel-C is a low level hardware/software construction language with C syntax that supports behavioral descriptions and uses a CSP-style communication model.

Although all these systems and abstraction languages have proved useful in various domains, they have different shortcomings. GARP, Streams-C, and SPARK do not support accesses to two-dimensional arrays, so image processing applications must be mapped manually. ROCCC accepts only perfectly nested and constant bound loops operating on arrays with affine index expressions. Moreover, all arrays are assumed to be located in the memory and no local data is allowed. The previous systems and languages do not support the stream-oriented optimizations that we discuss in this paper. They also do not provide some of the constructs that are essential for stream programming such as peeking.

3. FROM STREAMIT TO HARDWARE

Optimus is a compiler and synthesizer that takes as input a streaming application and generates an efficient FPGA (hardware) implementation. We designed a hardware template capable of representing fairly optimized circuits for streaming applications. The template captures the salient properties of streaming codes, and is malleable enough that it can be used in many different circuit designs we generate. This section details our approach using a simple example illustrated in Figure 1.

3.1 Input Language

We use the StreamIt language as the input language to the com-

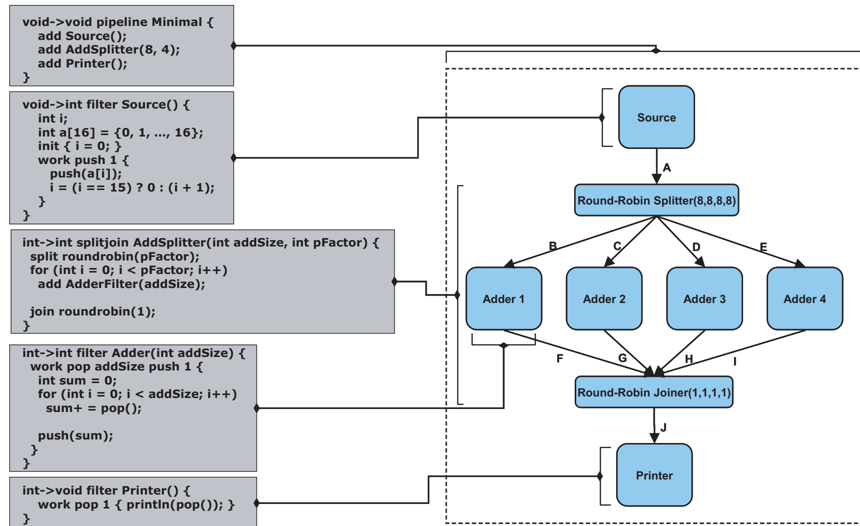


Figure 1: A sample StreamIt program is shown on the left. The corresponding stream graph with all the filters instantiated is shown on the right.

piler. StreamIt is an architecture-independent programming language for high-performance streaming applications [24]. Programs in StreamIt are represented as graphs where nodes, called *filters* encapsulate computation, and edges represent FIFO communication. StreamIt is based on the synchronous dataflow (SDF) [15] model of computation. Each filter consists of a *work* function that repeatedly executes when sufficient data is available on its input FIFO (queue). The work function reads data from its input queue using *pop* operations, and writes data to its output queue using *push* operations. The work function can also inspect input without removing them from the FIFO using a *peek* operation. Peek operations are critical for exposing data parallelism in sliding-window filters (e.g., FIR filters), as they elide the need for internal filter state. StreamIt provides three hierarchical stream primitives for composing filters into larger stream graphs: *pipeline*, *splitjoin*, and *feedback loop*. A pipeline connects streams sequentially. A splitjoin specifies task or data parallel streams that diverge from a common splitter and merge into a common joiner. A feedback loop creates a cycle in the dataflow graph.

A simple StreamIt program and its corresponding stream graph are illustrated in Figure 1. This example consists of five streams: `Minimal`, `Source`, `AddSplitter`, `Adder`, and `Printer`. `Minimal` is a top level pipeline with three-stages. The middle stage, `AddSplitter`, consists of a splitter, 4 parallel `Adder` filters, and a joiner. The splitter distributes data to each of its connected filters in a roundrobin fashion. Each `Adder` receives eight data elements at a time. StreamIt allows stream graphs to be described programmatically, and affords the compiler the ability to fully elaborate the graph at compile time by instantiating and connecting instances of the filters.

Filters in StreamIt are self-contained, and can only access their locally declared variables and fields. Hence, data exchange between filters is accomplished using explicit transfers across inter-filter FIFOs (queues) using the push and pop operations. StreamIt filters may be either stateful or stateless. In Figure 1, the `Source` filter is stateful; all the other filters are stateless. `Source` is stateful because the `i` field carries a dependence from one execution of the work function to the next. In addition to the work function, filters may also define an *init* function to initialize local fields.

The structured nature of StreamIt programs make them a good match for realizing streaming code in hardware. We leverage many of the language features during compilation for FPGAs.

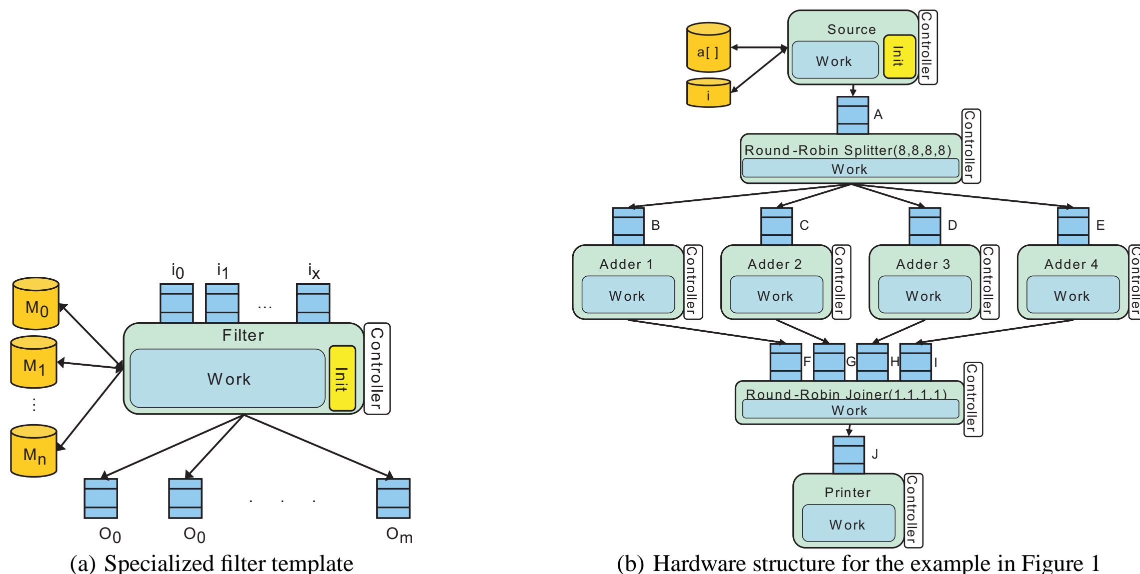
3.2 Synthesizing a Stream Graph

Optimus uses a specialized filter template to synthesize the filters that appear in the input stream graph. The template is shown in Figure 2(a). The template consists of five main components: input queues, output queues, memories, the filter itself, and the controller. Input and output queues are used to send and receive data. The template supports an arbitrary number of input and output queues to implement splitters and joiners. Memory modules are used to store the state for stateful filters. Each filter can be connected to several memory components. All the memory modules are local to each filter. For each memory module, there are dedicated read and write buses between the module and the corresponding filter. The buses are shared between the accessors of the memory in the filter. The hardware block implementing the filter consists of the work module and an optional init module. Both init and work modules will be connected to a memory module in case that module needs initialization. The controller makes sure that the init function gets executed only once before the first invocation of the work function. Depending on the way that the circuit is scheduled, the controller may have other responsibilities to orchestrate the execution.

After instantiating the template for all filters in a stream graph, the next step is to connect them. This step is straightforward based on the stream graph and the way data flows through the graph. Whenever Optimus connects the template for two filters together, it merges their input and output queues together. In other words, those two filters will share one FIFO queue for transferring data between them. Figure 2(b) shows the top-level hardware for the stream graph in Figure 1. As it is illustrated, the only stateful filter with memory components is the `Source` filter. The `Source` filter also is the only filter with an init component.

3.3 Synthesizing Filters

Each filter is organized as a control flow graph (CFG) with an overlayed data flow graph (DFG). Basic blocks (BBs) of instructions are used as the core building units for each filter. The template for the BBs is shown in Figure 3(a). Each BB module has four sets of input/output signals. The first set includes the control



(a) Specialized filter template

(b) Hardware structure for the example in Figure 1

Figure 2: (a) The specialized template used for synthesizing filters. (b) The complete hardware for the stream graph shown in Figure 1.

signals. All BBs have one *control input* signal and one or more *control output* signals. A control input signal will activate a BB as long as the signal is active. A control output signal will be connected to the inputs of the other BBs in order to activate them in the right order. Connecting these control signals is done based on the edges in the corresponding CFG. The second set of input/outputs consist of data signals which carry operand values. Optimus uses a DFG for connecting these signals. The third set of input/output signals help each module to communicate with external resources such as queues, memories, and other types of IP (intellectual property) cores. These signals provide a unified interface in which any IP core can be connected to the hardware. The last set of signals, marked as *Ack* in Figure 3(a), is meant for flow control. The Ack signals are useful when a BB cannot perform its operations in the associated clock cycle and needs to wait one or more cycles. This mainly happens when a BB accesses an external resource (e.g., memory) and the resource is not ready to respond within the same cycle.

Generally, each BB ends with one or more registers to store live-out data and control signals. In the baseline design, it is assumed that all the live-out values are registered to control the wire latency in the final design. Since all the live values are latched at the basic block outputs, one clock cycle is needed to transfer data from one BB to its successors. In other words, the execution of each BB takes at least one cycle.

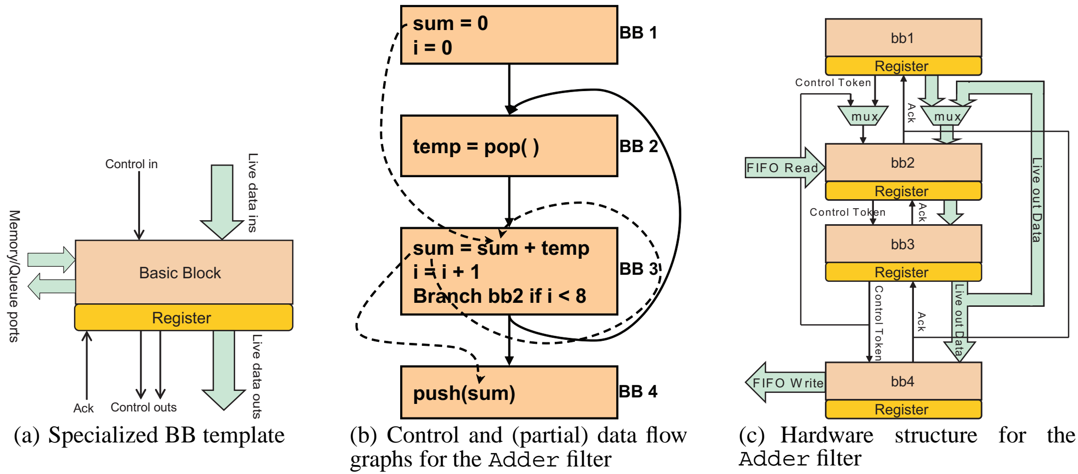
After the hardware module for each BB is generated, Optimus will connect the modules based on the CFG and DFG for each work or init function. Connecting the control signals is based on the CFG. The control outputs of all BBs are connected to the control inputs of the immediate successor BBs. In case a BB has more than one control input signal, MUXes are used to select the right control input signal. The DFG is used for connecting the data signals, such that the live-out signals of each BB are connected to the live-ins of the immediate successor. MUXes are again used in case a value can reach a BB from two different paths.

We will use the `Adder` filter as an example to clarify the main points. Figure 3(b) shows the CFG and DFG for `Adder`. The solid lines show the control flow and the dashed lines show the data flow for *sum*. This graph has four BBs and there is a backedge from BB 3 to BB 2. Based on the DFG, a data signal is needed for transferring the value for the variable *sum* from BB 1 to BB 3 through BB 2. All the control flow signals in the figure are connected based on the CFG for the `Adder` filter. Since BB 2 is the target of two branches (the fall through from BB 1 and the loop target from BB 3), a MUX is added to its inputs for selecting the appropriate control signal. The execution of the `Adder` filter will take 18 cycles (2 cycles for each of the 8 iterations, and 2 cycles for the rest of BBs).

The only remaining task is to generate hardware to fill each BB module based on the operations of that BB. Optimus generates a function unit (FU), similar to Figure 4(a), for each operation. Each FU can have multiple inputs and outputs and one predicate input. If a BB has a conditional branch operation, Optimus will generate a comparator FU to compute the *control output* signals. The data flow graph in each BB determines how the FUs should be connected to each other. At the end of each BB, its *control input* signal is used to enable the register module. Figure 4(b) illustrates the FU and the necessary connectivities for BB 3 of the `Adder` filter. This figure does not show all the details of computing control signals and setting the Ack signal.

3.4 Hardware Orchestration

The final issue is the orchestration of execution for the entire streaming circuit. We focus on two ways of scheduling the filter executions: *Static* and *Greedy*. In a static schedule, the compiler dictates the number of executions of each filter, such that it consumes all of its input data and produces sufficient data for its consumers. In this model, the compiler guarantees that a filter will have a sufficient number of input data available. Hence the execution of the filter work function will not block on reads (i.e., pops). Similarly, the compiler also asserts that the output queue from a filter is sufficiently empty so that all the writes (i.e., pushes) also succeed without blocking the filter. In this type of scheduling, double buffering is used between pairs of filters to provide communication-computation concurrency. This allows the producer and consumer to run independent of each other. The size of each individual queue is typically set to the least common multiple of the pop rate and push rate of the consumer and procedure filters. We refer to this form of scheduling and FIFO sizing as “rate-matched”.



(a) Specialized BB template

(b) Control and (partial) data flow graphs for the `Adder` filter

(c) Hardware structure for the `Adder` filter

Figure 3: (a) The template used for synthesizing basic blocks. (b) Control flow graph and partial data flow graph for the `Adder` filter. (c) The complete hardware generated for the `Adder` filter.

A greedy schedule takes a different approach and does not try to statically rate-match filters. In this approach, filters execute eagerly, and block when they attempt to read from an empty queue, or write to a full queue. Since all queue accesses are blocking in this approach, the size of the queue throttles the execution of the stream graph. This allows for a tradeoff between the size of the queue and the overall circuit throughput. Smaller queues take up less area, but may not be optimal. In our benchmarks, we observed that it is common that a queue size of one element is sufficient for correct execution that is also as efficient as a rate-matched static scheduled. The queue sizing is further discussed in the following section.

Optimus is capable of generating the necessary hardware for both schedulers. This choice has implications on the rest of the circuit in terms of queue sizes, power consumption, execution time, and allowed hardware sharing. In this paper, the greedy scheduler is used for all designs. The comparison between the two schedulers is left for future work.

4. STREAM OPTIMIZATIONS

Streaming languages allow programmers to focus on designing their applications. Specifically, programmers describe their computation programmatically and algorithmically, and do not need to commit to specific implementation details related to scheduling, buffering, synchronization, or the underlying data transport mechanisms in their target platforms. This programming practice leads to code that is easy to maintain and port, but places a burden on the compiler to derive high-performing implementations.

Optimus applies many of the classical optimizations used in previous works, and introduces a set of new macro-functional optimizations that specifically target streaming programs. Our compiler focus is on improving communication latency and reducing memory storage requirements (i.e., area). Communication latency can be optimized by sending larger chunks of data between filters. Storage can be optimized by intelligently sizing queues between filters, and allocating output registers to increase spatial reuse. It is not uncommon in today's synthesis frameworks to apply many of these optimizations manually, either directly in the source code or after the circuit is generated. This process can be time-consuming, error-prone, and complex for large benchmarks. It also defeats the purpose of using elegant and practical streaming languages that are attractive because they promote productive and portable programming.

4.1 Queue Allocation

The queues that connect hardware filters are implemented using the SRAM structures on the FPGA. FPGAs have limited SRAM capacity, ranging from 4 KB on the low-end FPGAs to 128 KB on the high-end ones. The SRAM is also used to implement the local arrays and other data structures used by the filters. Thus, for large stream graphs, the SRAM quickly becomes the bottleneck resource. The scheduling strategy used to orchestrate the execution of the filters can significantly impact the storage requirements. Optimus judiciously calculates the size of each queue to allocate between filters in order to better utilize the SRAM and maintain the high throughput achieved by a rate-matched static schedule.

The idea behind our approach is to recognize that a slot in the queue may be reused if the value that previously occupied the slot is already consumed. Thus, we can reduce the total storage requirement for the inter-filter FIFOs if we can determine the maximum number of overlapping lifetimes for the values exchanged between filters.

Figure 5 shows the cycle-by-cycle schedule of a pair of communicating filters. Only the push and pop operations are shown. The schedule shows all the cycles in the steady state executions of the producer-consumer pair. Suppose the producer pushed N items per execution of its work function, and the consumer popped M items from the queue every time its work function executes. For the filters to be rate-matched, the producer must run its work function $\frac{LCM(M,N)}{M}$ times, and the consumer $\frac{LCM(M,N)}{N}$ times. We determine the maximum number of overlapping lifetimes by simulating the rate-matched schedule. We use double-buffering during simulation to provide communication-computation concurrency. The simulation needs to only cover one steady-state execution of the filters. In the case a filter peeks at more data than it pops, an initialization schedule is run to prime all the FIFOs.

A causally correct schedule is obtained by shifting the producer schedule to occur at time 0, and shifting the consumer schedule down such that all pops appear at least one cycle after their corresponding pushes. Figure 5 shows an example schedule. Such a schedule reveals the lifetime of every entry in the queue between the producer and the consumer. The lifetime extends from the cycle at which an entry is pushed and the cycle at which the entry is

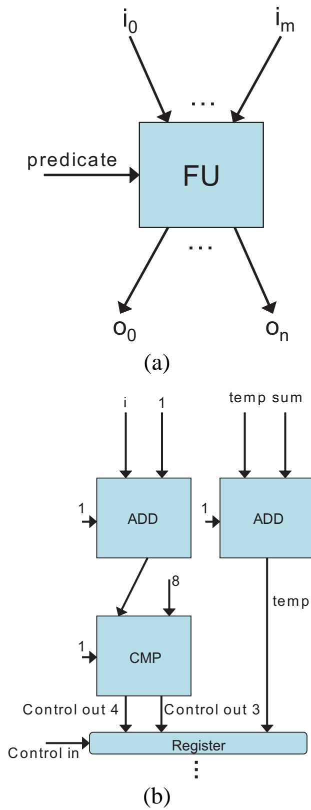


Figure 4: (a) Template for synthesizing operations. (b) Simplified hardware structure for BB 3 in Figure 3(b).

popped. The maximum number of queue entries whose lifetimes overlap can be easily calculated from the schedule. In Figure 5, the maximum number of overlapping lifetimes is 3. Setting the queue size to a value less than this maximum will stall the filters because one of the pushes at the producer cannot succeed as it would appear before the pop of the previous queue entry. Conversely, setting the queue size to a value more than this maximum would not improve the schedule. Thus, the minimum queue size for a producer-consumer pair that retains the throughput of the static schedule is obtained by calculating the maximum number of overlapping lifetimes of a rate-matched schedule.

4.2 Queue Access Fusion

A critical factor in streaming applications is sustained throughput. One of the key issues that can have negative effect on the throughput of a streaming circuit is communication latency between different filters. This issue arises from the fact that each queue or memory access, regardless of data width, takes at least one cycle. The one cycle access time would have a direct affect on the latency of the longest path in filters. It can also limit the filter-level parallelism in splitters and joiners. To overcome these bottlenecks, we consider bundling similar queue accesses together to create a single wide access using *queue access fusion*. This is conceptually similar to creating SIMD loads and stores. Of course, to support fused queue accesses, the basic queue structure requires modifications.

Code motion and loop unrolling are applied to find opportunities for fusing queue accesses and shortening the longest paths in a filter. Automatic SIMDization techniques use a similar approach with one difference: the vector length is known a priori, whereas

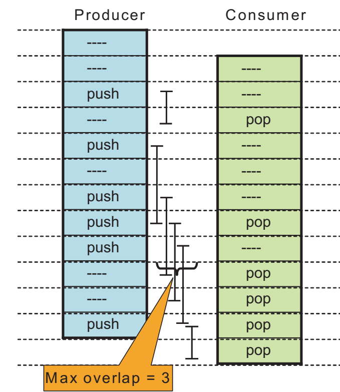


Figure 5: Overlapped producer-consumer schedules showing maximum number of overlapping lifetimes.

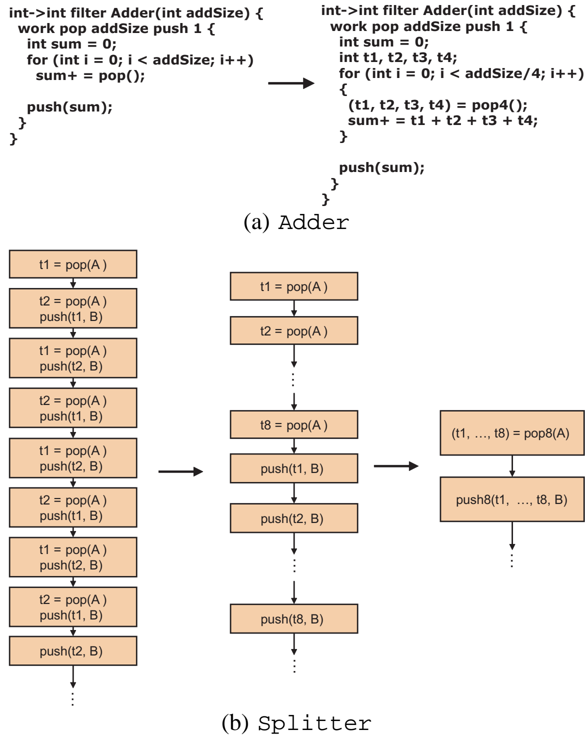
we can realize variable vector lengths between producer-consumer filter pairs. Loop unrolling is applied to loops with queue or memory operations to expose the operations to the code motion phase. Optimus needs to consider area constraints while it is performing the unrolling because unrolling may result in area expansion and cause the design to overflow the target FPGA. The next step, is to cluster memory and queue operations via aggressive code motion. The end result is a several clusters of memory and queue operations with no other intervening operations. Each cluster of operations is assigned a vector length according to the number of operations in the cluster. Subsequently, the compiler determines a single vector length for the filter by calculating the greatest common divisor of each cluster's vector length. For example, if the vector lengths of the clusters are 8, 12, and 16, then the filter's vector length is 4.

Figure 6 shows this optimization applied to a filter and a splitter from Figure 1. For the `Adder` filter in Figure 6(a), the loop is unrolled 4 times and a vector length of 4 is chosen for fusion. The loop is not fully unrolled because of area constraints. The unoptimized `Adder` filter will take 18 cycles to finish (assuming the value of `AddSize` is 8), but the optimized one will take only 6 cycles. Figure 6(b) illustrates the effectiveness of the fusion optimization for the splitter filter. The unoptimized splitter needs 9 cycles to read 8 data values from its input queue and push them to the input queue of `Adder1`. During these 9 cycles, the next filter in the splitjoin (`Adder2`) would be idle while it awaits its input data to arrive. In this case, the access fusion optimization will reduce the filter's idle time to 2 cycles. The optimization in general reduces the critical path of computation and can reduce execution time. If the optimization is successful in finding large clusters of accesses and fusing them, it will also significantly reduce the total area of the design. If the optimization is not successful, the loop unrolling would result in area expansion. However, an intelligent compiler would reverse the unrolling when it is not profitable.

One of the restrictions imposed by the our generated hardware is that the vector length for all accesses from a filter to a specific queue has to be the same, although vector lengths to the same FIFO from different filters may differ. This is realized by incrementing and decrementing read and write pointers using different constant offsets. For example, if the read vector length is 1 and the write vector length is 8, the queue can be viewed as an 8x8 matrix with the write pointer pointing to the rows and the read pointer pointing to individual elements of the matrix. Figure 7 illustrates the possible configurations.

4.3 Flip-flop Elimination

As it was discussed in Section 3, all live-out data signals, includ-



(a) `Adder`

(b) `Splitter`

Figure 6: An example of access fusion using the stream program in Figure 1.

ing pass-through live signals, are registered at the end of each basic block to bound wire delays. The output of memory and queue operations cannot be registered in the block that issues those operations because memory (and queues) needs one cycle to respond. Therefore, the results of those operations are registered in the immediate successors of the issuing basic block, as well as along all blocks that transmit the values along to their destinations. The CFG in Figure 8(a) illustrates the registers added for various operands as rounded-edge rectangles attached to the basic blocks. Note that live operands are saved at the end of each basic block regardless of whether they are passing through or generated in that block. This register assignment ensures that the critical path in a CFG is not greater than the maximum of delays through the basic blocks.

Many of these flip-flops are unnecessary and can be removed without affecting the clock speed. In order to keep the circuit functional, a subset of registers must be maintained. There are two main situations where flip-flops cannot be removed. First, if an operand is both live-in and live-out along a backedge, it has to be registered before or after the backedge to prevent formation of a combinational loop. The second case is more complex. If an operand is the result of a queue or memory read, it does not have to be registered because the hardware for the queue and memory hold its output as long as no other operation has changed its read status. For a read operation from a queue, a status change occurs when another pop is issued. In a memory structure, status changes when a store writes to the same address as a read. When the compiler can determine that no intervening pops or read/write conflicts occurs, then it can elide the corresponding registers.

Figure 8 shows a sample CFG and all the data registers before and after the flip-flop optimization. Based on the rules for flip-flop elimination and ignoring clock cycle constraints, all the registers can be removed except X-register in BB 2 and the T-register in BB 5. The X-Register cannot be removed because there is an intervening pop operation in BB 2 that can change the status of the input queue. Note that if the control flows to the right instead of left after BB 1, then no register is needed because there are no pop operation along that path. The register for T also cannot be removed because T is both live-in and live-out along the backedge going from BB 5 to itself.

An issue with flip-flop elimination is the possibility of increasing the critical path length. In general, Optimus tries to balance the length of the combinational paths by splitting the large basic blocks and adding registers to the end of each BB. Optimus has an internal model of the target FPGAs to assess the latency of different combinational operations. If removing any of the registers in the flip-flop elimination optimization lengthens the critical path, then that register is left in place.

5. EXPERIMENTS

We compiled and simulated various applications from different domains. Our target platform is a Xilinx Virtex-4 (XC4VLX200) FPGA [26]. ISE Foundation was used for synthesizing the HDL generated by Optimus. Xilinx Xpower is used to measure the energy and power consumption of our circuits. For comparison, we used a 300 mW 300 MHZ embedded PowerPC 405 processor. We compare our FPGA results to the benchmarks compiled and executed on the PowerPC. We use the StreamIt compiler, and the same StreamIt source code for the benchmarks, to generate binaries that run on the PowerPC processor. Our benchmarks are FFT (fast Fourier transform), parallel adder (the example shown in the paper), bubble sort and merge sort, integer inverse DCT (discrete cosine transform), DES (data encryption standard), matrix multiply and its blocked variant. In the case of DES, we used a reference C implementation of the benchmark instead of the StreamIt version for the PowerPC measurements. This is because DES performs a lot of bit-level operations, and tuned implementation can cleverly carry out the operations in parallel using word-wide masks. In the case of the FPGA, we compile the StreamIt version of DES down to HDL.

Performance and Energy Consumption: Figure 9(a) shows the performance of streaming hardware compared with PowerPC for various benchmarks. In this experiment, none of the streaming-specific optimizations are used. Speedup varies from 1.1x to 58x for different benchmarks. Bubble sort achieves the highest speedup because it heavily exploits pipeline-level and instruction-level parallelism. Parallel adder has the lowest speedup over the baseline because the communication to computation ratio is high in this benchmark. Figure 9(b) illustrates the energy consumption of the circuits generated by Optimus as a fraction of the PowerPC energy usage. On average, these benchmarks consume 0.7x of the PowerPC energy. The only benchmarks which use more energy on the FPGA are parallel adder and DES. This again happens due to large communication to computation ratio in case of the parallel adder. In DES, the higher energy consumption is due to the inability of Optimus to efficiently take advantage of the bit-level parallelism in the stream graph. Considering the fact that the baseline processor is a 300mW core, these results show that the hardware generated by the Optimus system is suitable for low-power embedded systems in terms of both performance and energy consumption.

Queue Allocation: In the designs generated by Optimus, one of the main components that uses the on-chip memory is the queue structure. The queue allocation optimization tries to efficiently reduce the sizes of the queues without affecting the performance. The StreamIt compiler generally uses rate matching between the filters to calculate queue sizes. We used the rate matched queue sizes as the baseline and show the savings due to the queue allocation al-

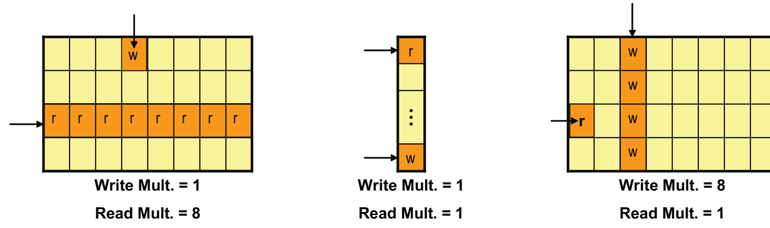
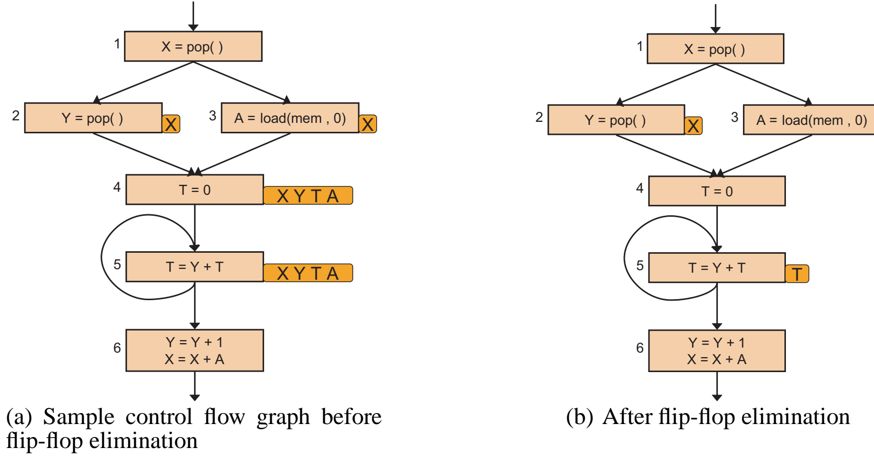


Figure 7: Various configuration of queues used by queue access fusion optimization.



(a) Sample control flow graph before flip-flop elimination

(b) After flip-flop elimination

Figure 8: An example of flip-flop elimination.

gorithm in Figure 10(a). As shown in the figure, this optimization reduces the queue sizes by an average of nearly 50%. Additionally, after reducing the queue sizes to the new values, no performance loss was observed in any of these benchmarks. These results demonstrate that the queue allocation optimization used by Optimus is quiet effective in saving the on-chip memory resources.

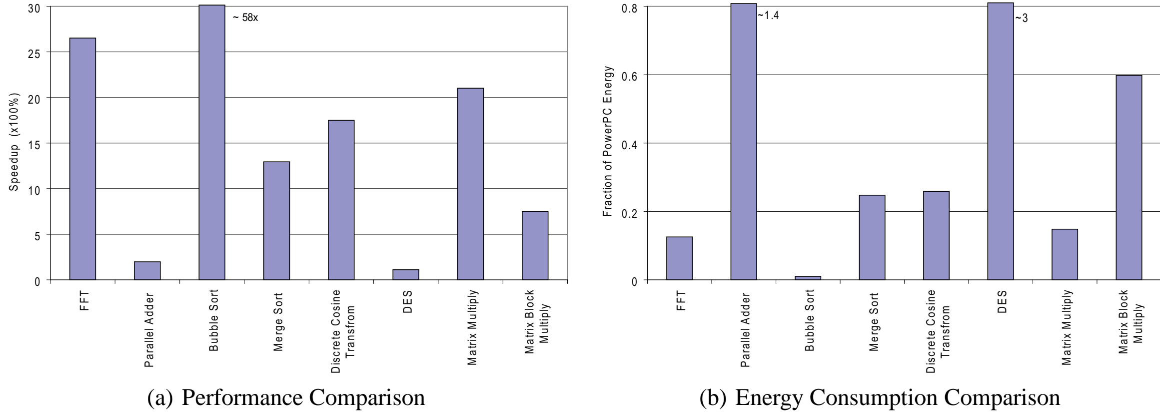
Queue Access Fusion: As discussed in Section 4.2, the goal of queue access fusion is to increase the throughput of streaming circuits by fusing multiple queue operations into a single (wider) operation. Figure 10(b) illustrates the effect of this optimization on various benchmarks in terms of performance. We limit the maximum vector length to 8. This means that the maximum speedup achievable is 8x. As shown in the figure, the average speedup is 3.2x, and 7.2x in the best case. In some benchmarks, no speedup is achieved because there was not any opportunity to fuse accesses in the slowest filters. The slowdowns are typically due to the fact that the wider queues are marginally slower than normal queues. In order to understand the area and performance tradeoff between different queue configurations, we synthesized three queues with the same size but different read/write widths. As the results in Table 1 show, the wider queues are slightly larger than their narrower counterparts.

Flip-flop Elimination: The goal of this optimization is to identify and eliminate redundant registers such that the circuit still functions properly and the critical path length does not change. The results of this optimization are shown in Figure 10(c). Flip-flop elimination reduces flip-flop utilization by 30% and slice utilization by 16%. As shown in the figure, the improvement in flip-flop use is always greater than slice utilization. This means that there are many slices used only for latching purposes and not for logic computation. The area savings due to this optimization vary based on the number of pops and loads and their arrangement in each benchmark.

Comparison to Handel-C: We compared our generated circuits to those generated using Handel-C and its compilation toolchain. Handel-C is a variant of the C programming language. It is aimed toward synthesizing hardware from C code. We implemented DES and DCT in Handel-C and generated their hardware designs. The Handel-C implementations preserved the overall streaming structure of the benchmarks. Our area and performance comparisons show that the Optimus-generated circuits are an average of 5% faster and 66% larger. Using our stream-specific optimization, we can further improve the performance of the Optimus-generated circuits so that they are 12x faster, although the designs are also 90% larger than the Handel-C designs.

There are several important factors that make the Handel-C designs inefficient in terms of performance. First, Handel-C is not able to automatically perform the same kind of macro-level optimizations that Optimus carries out. Second, Handel-C does not try to balance the critical paths between flip-flops to achieve higher frequency designs. The lack of these optimizations and transformations is the main reason the Handel-C designs lag in performance compared to the Optimus-generated ones. The optimizations can be done manually in the Handel-C code, but that requires more work for the programmer, and it would obfuscate the streaming nature of the code.

In terms of area comparisons, the designs in Handel-C are more area-efficient for two main reasons. First, Handel-C tries to utilize resources (IPs) that are unique to various families of FPGAs. Designs generated in this way are usually more area-efficient. Second, Handel-C performs some low-level netlist optimizations that improve the area by a large factor. We believe netlist-level optimiza-



(a) Performance Comparison

(b) Energy Consumption Comparison

Figure 9: Figure 9(a) illustrates the speedup comparison between the hardware designs and a 300 mW PowerPC 405 running at 300 MHZ. Figure 9(b) shows the energy consumption of the FPGA as a fraction of PowerPC energy use for various benchmarks.

Queue Configuration	Total number of bits	Number of Slices	Clock (MHZ)
(read width = 128, write width = 16)	4096	70	>300
(read width = 16, write width = 16)	4096	56	>300
(read width = 16, write width = 128)	4096	95	>300

Table 1: Area and delay for different queue configurations

tions should be implemented in the low-level hardware synthesis tool and not in a high-level compiler. Therefore, Optimus does not implement any of the low-level optimizations that Handel-C performs to improve the area efficiency.

6. CONCLUSION

Streaming applications are important to embedded systems developers. Improving the performance of these applications in an embedded setting is typically accomplished via special purpose processors and ASICs that are inflexible and invariably expensive to design. An alternate approach is to use configurable hardware fabrics such as FPGAs that provide a performance- and power-competitive platform for their cost. In addition, FPGAs are increasing available as components in heterogeneous systems, and their versatility makes them attractive platforms in a domain where software and consumer requirements change rapidly. Unfortunately, the complexity of programming FPGAs has limited their benefits as only system engineers with hardware design expertise are able to effectively map software down to hardware circuits.

The goal of our work is enable the efficient realization of streaming programs directly in hardware, when appropriate. Our Optimus compilation methodology allows for streaming programs expressed in a high-level streaming language such as StreamIt to be automatically refined to hardware and realized as circuits in FPGAs. The Optimus compiler uses a hierarchical compilation strategy that separates concerns between macro- and micro-functional requirements. Macro-functional optimization are geared to efficiently assembly filter module into larger applications. These optimizations affect space (area) and time (throughput) characteristics of the application circuits. Our goal in this regard is to provide the highest performance for the lowest area cost. Comparing our generated designs to an industry-strength compiler shows that we are performance and area competitive although we believe there is much more to be gained in our framework. Our results are largely enabled by stream-specific considerations and optimizations. Micro-functional optimizations are designed to improve the efficiency of the filter modules themselves. Our stream-aware optimization improve performance an average of 255% and reduce the area requirements by 16% compared to our baseline results.

7. ACKNOWLEDGMENTS

We thank Andrei Hagiescu for providing us with the Handel-C implementations of DES and DCT. Stephen Neuendorffer from Xilinx also helped us in using and understanding the Xilinx Synthesis tools. We also extend our thanks to the anonymous reviewers who provided excellent comments.

8. REFERENCES

[1] AMD torrenza architecture, 2008. http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.aspx.

[2] Intel quickassist technology, 2008. http://www.intel.com/technology/platforms /quickassist/index.htm.

[3] K. Bondalapati et al. DEFACTO: A design environment for adaptive computing technology. In *Proc. RAW*, pages 570–578, Apr. 1999.

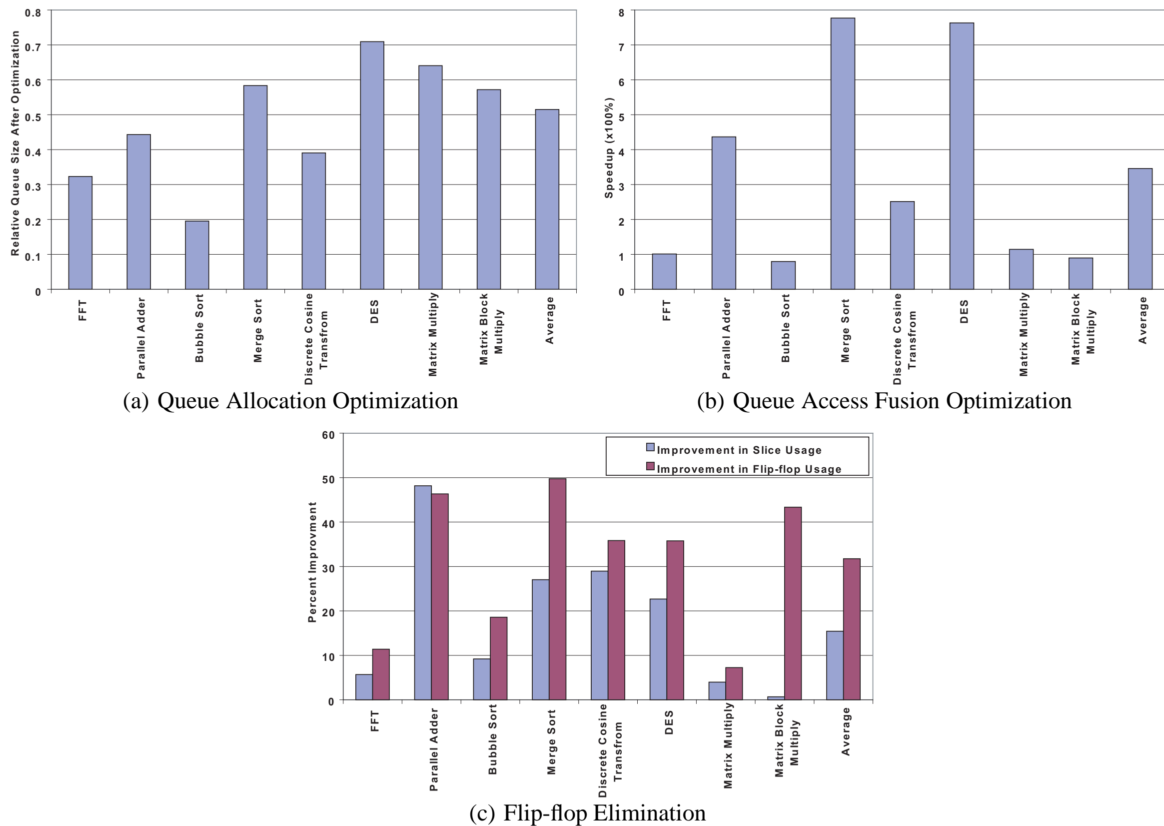
[4] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Gr.*, 23(3):777–786, Aug. 2004.

[5] Celoxica. Handel-C language overview, 1996. http://www.celoxica.com.

[6] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. '05 PLDI*, pages 224–236, June 2005.

[7] C. Consel et al. Spidle: A DSL approach to specifying streaming applications. In *Proc. 2nd GPCE*, pages 1–17, 2003.

[8] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski.



(a) Queue Allocation Optimization

(b) Queue Access Fusion Optimization

(c) Flip-flop Elimination

Figure 10: Performance improvements and area savings due to different optimizations performed by Optimus.

Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. 8th FCCM*, pages 49–56, Apr. 2000.

[9] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th ASPLOS*, pages 151–162, 2006.

[10] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized generation of data-path from c codes for fpgas. In *Proc. 2005 DATE*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. 16th Intl. Conf. on VLSI Design*, pages 461–466, Jan. 2003.

[12] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. 5th FCCM*, pages 12–21, Apr. 1997.

[13] Impulse-CoDeveloper. http://www.impulsec.com/.

[14] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. '08 PLDI*, pages 114–124, June 2008.

[15] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[16] W. Mark, R. Glanville, K. Akeley, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proc. 30th SIGGRAPH*, pages 893–907, July 2003.

[17] O. Mencer, H. Hubert, M. Morf, and M. J. Flynn. Stream: Object-oriented programming of stream architectures using pam-blox. In *Proc. 10th FPL*, pages 595–604, London, UK, 2000. Springer-Verlag.

[18] Mentor. Catapult C. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/.

[19] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8):63–69, 2003.

[20] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. May 2007.

[21] S. Sirowy, G. Stitt, and F. Vahid. C is for circuits: capturing fpga circuits as sequential code for portability. In *Proc. 16th FPGA*, pages 117–126, New York, NY, USA, 2008. ACM.

[22] SystemC-Consortuim. SystemC language overview, 2000. http://www.systemc.org.

[23] M. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. 31st ISCA*, pages 2–13, June 2004.

[24] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 02 CC*, pages 179–196, 2002.

[25] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[26] Xilinx. Virtex-4 data sheets, 2004. http://www.xilinx.com/support/documentation/virtex-4.htm.

[27] D. Zhang, Z. Li, H. Song, and L. Liu. A programming model for an embedded media processing architecture. volume 3553 of *Lecture Notes in Computer Science*, pages 251–261, July 2005.