Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines

Manjunath Kudlur Kevin Fan Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{kvman, fank, mahlke}@umich.edu

ABSTRACT

In this paper, we present a methodology for designing a pipeline of accelerators for an application. The application is modeled using sequential C language with simple stylizations. The synthesis of the accelerator pipeline involves designing loop accelerators for individual kernels, instantiating buffers for arrays used in the application, and hooking up these building blocks to form a pipeline. A compiler-based system automatically synthesizes loop accelerators for individual kernels at varying performance levels. An integer linear program formulation which simultaneously optimizes the cost of loop accelerators and the cost of memory buffers is proposed to compose the loop accelerators to form an accelerator pipeline for the whole application. Cases studies for some applications, including `FMRadio` and `Beamformer`, are presented to illustrate our design methodology. Experiments show significant cost savings are achieved through hardware sharing, while achieving the prescribed throughput requirements.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Experimentation

Keywords

system-level synthesis, application-specific hardware, loop accelerator

1. INTRODUCTION

As communication bandwidths are scaled or more features are added to portable devices, such as high-definition video, embedded computing systems are required to perform increasingly demanding computation tasks. Programmable processors are unable to meet increasing performance requirements and decreasing cost and energy budgets. Application specific hardware in the form of loop accelerators are often used to address these issues. A loop accelerator implements a critical loop from an application with far greater performance and efficiency than would be possible with a programmable implementation. However, a single accelerator designed and operated in isolation is insufficient. These tasks are commonly streaming applications that consist of multiple compute-intensive functions (e.g., filters) that operate in turn on streaming data. The natural realization of these tasks is a hardware pipeline of accelerators, each implementing one or more functions which process the data. These accelerators must be designed to meet overall throughput requirements while minimizing the hardware cost.

Designing a highly customized system of accelerators presents several difficult challenges. The design space is enormous because of the large number of variables, which include the number, type, and specific design of each accelerator, mapping of application loops to accelerators, arrangement of accelerators in the overall pipeline, and method of inter-accelerator communication. In the face of such challenges, an automated accelerator pipeline design system enables the systematic exploration of a much larger portion of the design space than is possible with manual designs, leading to higher-quality results. In addition, an automated system designs pipelines that are correct by construction by using a parameterized template, greatly reducing the verification portion of the product cycle. All of these factors enable automated design systems to deliver high-performance, low-cost solutions with shorter time-to-market, which is critical particularly in the embedded domain.

In this paper, we present an automated system for designing stylized accelerator pipelines from streaming applications, referred to as *Streamroller*. Streamroller synthesizes a highly customized pipeline that minimizes hardware cost while meeting a user-prescribed performance level. The input to the system is a behavioral description of the application specified in C that is comprised of a system specification and a set of kernels. The system specification describes the organization and communication in the pipeline, while the kernels describe the functionality of each stage on a single packet of data. Streamroller designs the complete accelerator pipeline by determining the throughput of each stage as well as the inter-stage buffer organization. A unique aspect of the system is the utilization of multifunction loop accelerators to enable multiple pipeline stages to time multiplex the hardware for a single pipeline stage. This approach sacrifices performance as kernel execution is sequentialized, but greatly increases the ability to share hardware in the design and thus drive down the overall cost.

The contributions of this work are threefold:

- A systematic design methodology for creating rate-matched accelerator pipelines with minumum cost at a user-specified throughput.
- A system that can exploit high degrees of hardware reuse by mapping multiple loops to multifunction accelerators.
- A stylized accelerator pipeline template optimized for streaming applications.

Related Work. High level synthesis converts a high-level specification into a structural design. Most high level synthesis systems

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

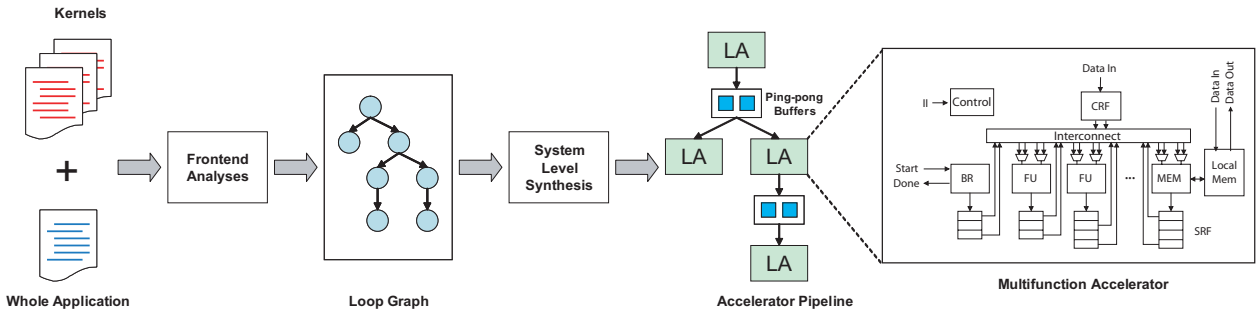


Figure 1: Overview of the Streamroller synthesis system.

build an operation-level data flow representation of the specification, and derive the datapath for the hardware from the schedule of the DFG. However, the granularity of synthesis has varied over the years. One of the earliest systems, Chippe [1], used a PASCAL-like language as the input specification. It synthesized hardware for acyclic basic blocks without memory accesses. The granularity of synthesis was a basic block for most of the earlier systems [12, 3, 2, 4]. MIMOLA [8] and Cathedral III [10] represent comprehensive approaches to high level synthesis. They perform memory and data path synthesis for a restricted domain of applications. [19, 9, 17] are also some of the earlier systems that varied widely in the input specification language and granularity of synthesis. While the above systems took a hardware-centric view of the high level synthesis problem, our system takes a programmer centric view, similar to [14]. The input specification is sequential C which makes it easy for the designer to develop and debug applications quickly, and the system automatically derives the architecture at a desired performance level.

Our approach has lot of similarities to system level design methodologies based on process networks. [5] and [15] map applications expressed as Kahn process networks to multiprocessor architectures. [5] produces standard cell implementations, whereas [15] maps to FPGAs. Their solution reduces hardware cost, while meeting performance constraints. While they use fixed estimates for the processing times of individual nodes, our system actively selects the processing rate for each node to control the overall cost. [16] and [21] are compilation systems that map process networks to a fixed multiprocessor system. Modeling of performance constraints in our system is similar to these works. Also, [21] achieves hardware sharing by mapping mutually exclusive tasks to the same hardware. However, we map arbitrary loops to the same hardware when there is enough slack. The loop graph used in our system is similar to a process network. However, the main difference between these works and ours is that process networks have parallel semantics, which makes it harder to develop applications. Our system derives the loop graph automatically from the sequential C program.

2. SYSTEM OVERVIEW

Figure 1 shows a broad overview of our accelerator pipeline synthesis system. The system takes as input the application written in C, expressed as a set of communicating kernels. Performance and design constraints, such as overall throughput of the pipeline, clock period and memory bandwidth, are also specified. The frontend of the system performs data dependency analysis on the application to derive the loop graph, which is a representation of the communication structure between the kernels. The system synthesizes an accelerator pipeline with minimum cost to meet the performance constraints. The pipeline consists of a number of loop accelerators (LAs) to execute the kernels in the application and ping-pong memory buffers for communicating values. The following subsections describe the input specification and the accelerator pipeline schema in more detail.

2.1 Input Specification

The input to the synthesis system is the whole application written in C. Simple stylizations are imposed on the structure of the C program. The stylizations make the analysis of the program simpler, but still enable a wide variety of media and network applications to be expressed. Sequential C semantics make it easy for applications to be developed and debugged quickly, even with the stylization restrictions. The input program consists of two logical parts, viz., a set of *kernel specifications* and the *system specification*.

Kernel specification. Conceptually, kernels form one stage of processing in the application. For example, in wireless applications, a low pass filter can be a single kernel. In our system, a kernel is expressed as a single C function. All inputs and outputs to the kernel have to be provided as arguments to the function. Arguments can be C arrays or scalars. The body of the kernel functions have to be perfectly nested `for` loops. Separating kernels into independent functions enables reuse and modularity. For example, many image procecessing applications perform the same transform on an input image in multiple stages. The same kernel function can be called with appropriate arguments to accomplish this.

System specification. The system specification describes one "packet's" forward flow through the pipeline. The system specification is expressed as a C function whose body contains a sequence of calls to the kernel functions. The system function will be invoked continuously on consecutive packets of data. What constitutes a packet depends on the application. In image processing applications, a packet can be a sub-block of a bigger image. In wireless applications, a packet can be a chunk of data received over the wireless channel. Typically, the applications in these domains process continuous streams of such packets. However, the processing that happens on a single packet is sequential in nature. Thus, our input specification fits well for expressing applications in these domains.

Figure 2 shows an example input specification. The system specification function is `fmradio`, which is shown in Figure 2(b). It takes the array `inp` as the input and outputs `out`. The body is made of calls to different kernels shown in Figure 2(a). `fmradio` uses local arrays to pass data between the kernels. A simple dataflow analysis of the system specification yields the *loop graph* shown in Figure 2(c). The nodes in the loop graph correspond to kernels whereas the edges indicates dataflow through an array between kernels. Note that arbitrarily complex loop graphs can be expressed by using just straight line C code.

Performance specification. The applications targeted by our synthesis system have real time requirements usually expressed at the highest level in terms of, say, frames/second or Kbps. Since the input specification corresponds to end-to-end processing of one packet, the real time constraint can be easily translated to the number of times the system specification function has to be called per second. Given the clock period, the performance specification reduces to the number of cycles between consecutive invocations of the system specification function. For example, consider the application in Figure 2. The `fmradio` function completely processes

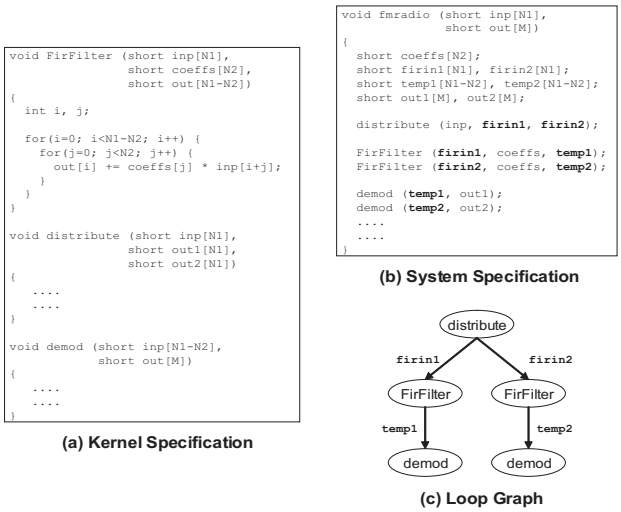


Figure 2: Input specification

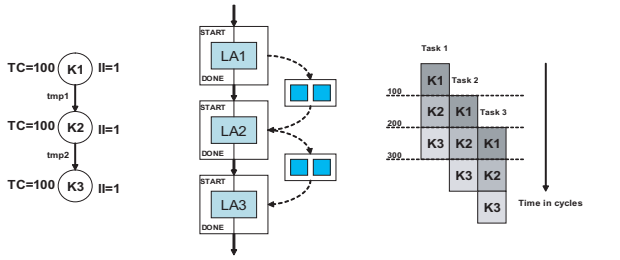


Figure 3: Example accelerator pipeline (High performance)

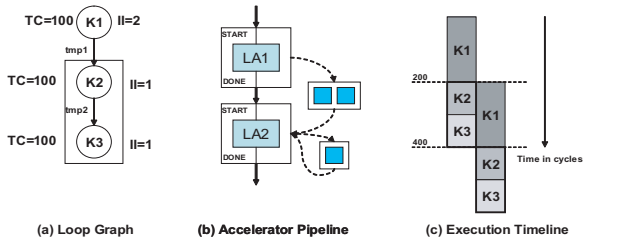


Figure 4: Example accelerator pipeline (Low performance)

one input packet `inp`. Suppose `N1` is 512, `fmradio` processes $512 \times 16 = 8192$ bits per call. To achieve a real time requirement of 128 Mbps, `fmradio` has to be called $\frac{(128\times1024\times1024)}{8192} = 16384$ times every second. If the clock frequency under consideration is 200 MHz, the `fmradio` function has to be invoked approximately once every 12208 cycles to meet the real time requirement.

2.2 Accelerator Pipeline Hardware Schema

Each kernel is mapped to a *loop accelerator* (LA). Depending on the performance requirements, multiple kernels can be mapped to the same LA, which performs the functions of both the kernels. These multifunction LAs form the building blocks for the accelerator pipeline. The intermediate arrays used in the input specification are mapped to SRAMs with ports connected to producer and consumer LAs. This section presents the hardware schema of the individual LAs and the accelerator pipeline.

Multifunction Loop Accelerator. The hardware schema used in this paper is shown in the inset on the right side of Figure 1. The innermost loops of the kernel specification function are modulo scheduled, and the architecture for the LAs is derived directly from the schedule. Modulo schedules [13] are characterized by initiation interval (II), which is the number of cycles between invocations of successive iterations of a loop. Therefore, high performance schedules of a loop have lower IIs. The accelerator is designed to exploit the high degree of parallelism available in modulo scheduled loops with a large number of function units (FUs). Each FU writes to a dedicated shift register file (SRF); in each cycle, the register contents shift downwards by one register. Wires from the registers back to the FU inputs allow data transfer from producers to consumers.

Multifunction LAs are designed to execute more than one loop nest. The general hardware schema for multifunction LAs is similar to a single function LA. The set of FUs in the multifunction LA is the union of FUs required by the individual loops. The widths and depths of SRFs are set such that they can support values of operations from all loops assigned to the corresponding FUs. A detailed description of how an optimal datapath for a multifunction LA is derived is beyond the scope of this paper; readers are referred to [7] for more information. Instead, this paper focuses on how multifunction LAs can be used as building blocks to build an accelerator for the whole application.

Accelerator Pipeline Schema. The accelerator pipeline is designed such that all processing on a single packet of data (henceforth referred to as a *task*) is done sequentially to respect the program order in the system specification function. However, multiple tasks can be in progress in the pipeline at the same time. The sequential execution is achieved by using the two control signals `START` and `DONE`. The LA begins execution of a loop when the `START` signal goes high, and raises the `DONE` signal at the end. By connecting the `DONE` signal of a producer LA to the `START` signal of a consumer LA, sequential execution of a task is ensured. Arrays used for communication are mapped to SRAMs in the accelerator. Since multiple tasks can be in flight in the pipeline, more than one SRAM buffer could be allocated to a program array.

Figure 3(b) shows the accelerator pipeline corresponding to the loop graph in Figure 3(a). There are three loops `K1`, `K2`, and `K3` in the application, each with a trip count (**TC**) of 100. The accelerator shown in Figure 3(b) is capable of executing the application with an overall throughput of 100 cycles. Each loop is modulo scheduled with II=1. Therefore, the approximate latency of each loop (one stage in the accelerator pipeline) is 100 cycles. Figure 3(c) shows the execution timeline for 3 tasks executing in the pipeline. Note that execution of `K1` in task 2 is overlapped with execution of `K2` in task 1. This means that `K1` will be producing new values for the array `tmp1` while `K2` is still using the old values. To avoid this and still provide overlapped execution of tasks, two SRAM buffers are allocated to `tmp1`. Alternate tasks ping-pong between these 2 buffers. Similarly, two buffers are allocated for `tmp2`.

Figure 4(b) shows a different accelerator pipeline for the same application. This pipeline has a lower throughput of 200 cycles, as opposed to 100 cycles capable by Figure 3(b). In this pipeline, `K1` is modulo scheduled with II=2, which is a lower performance implementation. Also, LA2 is a multifunction accelerator capable of executing `K2` and `K3`, each with an II of 1. Figure 4(c) shows the execution timeline of 2 tasks through the pipeline. Note that execution of `K2` and `K3` never overlap across tasks. Therefore, only one buffer is allocated for `tmp2`.

3. DESIGN METHODOLOGY

This section presents our methodology for designing an accelerator pipeline for an application. Section 3.1 describes the cost tradeoffs for various components of the accelerator pipeline. Section 3.2 describes an integer linear programming (ILP) formulation for finding an optimal cost accelerator pipeline at a prescribed throughput. Section 3.3 presents a practical end-to-end system that generates Verilog RTL from the sequential C application.

3.1 Cost Components

The cost of individual loop accelerators forms a major component of the cost of the accelerator pipeline. As described in Section 2.2,

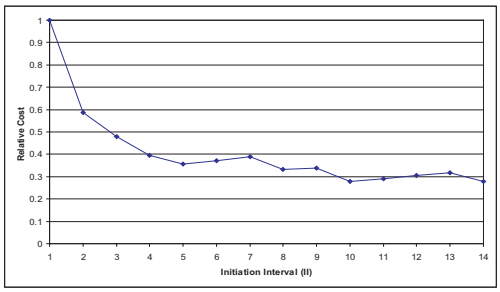


Figure 5: II vs. Cost of a loop accelerator

the datapath for an LA is derived from the modulo schedule of the innermost loop of a kernel function. An LA consists of a number of FUs connected to SRFs. Since all operations in the loop body are packed within II cycles in a modulo schedule, the number of FUs is determined solely by the II. The cost of FUs is also determined by the bitwidths of operations assigned to it. Note that the bitwidth of an FU has to be as large as the widest operation assigned to it. The bitwidth of a SRF is same as the FU connected to it. The depth, however, depends on the schedule. The SRF has to hold all live values produced by an FU. If the consumer of an operation is scheduled far away from its producer, the SRF connected to the producer FU will be deep. Also, when the loops have recurrences, the variables that carry the dependences across the recurrence must be live for at least II cycles. Thus, the cost of SRFs tends to increase for very large IIs.

Figure 5 shows a plot of the cost of a single LA with increasing II. The loop for which this LA was built is a part of the *Beamformer* [11] application. The costs are obtained by scheduling the loop at different IIs using a cost sensitive modulo scheduler [6] and synthesizing the resultant Verilog using the Synopsys design compiler. The highest cost corresponds to the lowest II of 1, which is the highest performance implementation of the loop. As II is increased, the number of FUs in the datapath and the corresponding SRFs decrease. Therefore, we see the general trend of decreasing overall cost with increasing IIs. However, beyond II of 10, there is no decrease in cost. At high IIs, the depth of SRFs corresponding to the recurrence variables grow disproportionately compared to other SRFs. Thus, there is no further cost improvement beyond II of 10.

Apart from II, the other dimension that affects multifunction accelerator cost is the different loops that the LA implements. A multifunction LA saves cost over two single function LAs. The amount of cost saved depends on the mix of operations in the two loop bodies. The more similar they are, the more cost saved. However, a multifunction LA has an adverse effect on performance. Since there is only one physical hardware to execute two loops, instances of these two loops in different tasks cannot be overlapped. To achieve the same overall throughput of the pipeline, the multifunction LA might have to implement a higher performance version of the individual loops. Thus, the tradeoff of independent LAs with low performance versus one multifunction LA implementing high performance versions of the loops, has to be considered.

The other major component of cost of the accelerator pipeline is that of the memory buffers. Note that there has to be at least one buffer for every array in the application. To allow for task overlap, more than one buffer might have to allocated to an array. The size and bitwidth of an array is application dependent and the modulo schedule for a loop has no control over the cost of a memory buffer. However, depending on the amount of task overlap required, which is dictated by the overall throughput requirement, the number of buffers for an array vary.

3.2 ILP Formulation

The accelerator pipeline design system has to judiciously choose the IIs for each kernel in the loop, and the number of buffers al-

Minimize: $\sum_{i=1}^{p} C_i + BUFCOST$
Subject to:

$$IIMIN_i \leq II_i \leq IIMAX_i \quad \forall i \quad \textbf{(1)}$$
$$L_i = TC_i \times II_i \quad \textbf{(2)}$$
$$L_i \leq \tau \quad \forall i \quad \textbf{(3)}$$
$$\sum_{k \in path\ i \rightarrow j} L_k \leq d_{a,i,j} \times \tau \quad \forall paths\ i \rightarrow j \quad \textbf{(4)}$$
$$\sum_{j=1}^{p} b_{i,j} = 1 \quad \forall i \quad \textbf{(5)}$$
$$\sum_{i=1}^{p} b_{i,j} \times L_i \leq \tau \quad 1 \leq j \leq p \quad \textbf{(6)}$$
$$\left.\begin{array}{lll} TL_{i,j} & \geq & 0 \\ TL_{i,j} & \leq & P \times b_{i,j} \\ TL_{i,j} & \leq & L_i \\ TL_{i,j} & \geq & L_i - (1 - b_{i,j}) \times P \\ \sum_{i=1}^{p} TL_{i,j} & \leq & \tau \end{array}\right\} \quad \textbf{(7)}$$
$$II_i = \sum_{k=IIMIN_i}^{IIMAX_i} k \times ii_{i,k} \quad ii_{i,k} \in \{0,1\} \quad \textbf{(8)}$$
$$\sum_{k=IIMIN_i}^{IIMAX_i} ii_{i,k} = 1 \quad \textbf{(9)}$$
$$CL_i = \sum_{k=IIMIN_i}^{IIMAX_i} Cost(i,k) \times ii_{i,k} \quad \textbf{(10)}$$
$$SUMC_j = \sum_{i=1}^{p} b_{i,j} \times CL_i \quad \textbf{(11)}$$
$$MAXC_j \geq b_{i,j} \times CL_i \quad 1 \leq i \leq p \quad \textbf{(12)}$$
$$C_j = MAXC_j + 0.5 \times (SUMC_j - MAXC_j) \quad \textbf{(13)}$$
$$BUFCOST = \sum_a d_{a,i,j} \times Memcost(a) \quad \textbf{(14)}$$

Figure 6: ILP formulation for system level synthesis

located for each program array such that the overall throughput is achieved, and at the same time, cost is minimized. The system also has to consider combining many loops into one multifunction LA whenever cost savings are possible. In this section, we develop an integer linear programming (ILP) formulation that optimizes the overall cost of the accelerator pipeline by choosing IIs and the number of buffers for the arrays. Figure 6 shows the integer linear program for the optimization problem.

Consider the *loop graph* constructed from the system specification function. For every kernel instance i in the system specification function, the loop graph has a vertex v_i. If a kernel i writes to an array a, and kernel j reads from the array, and edge $e_{a,i,j}$ is added between vertices v_i and v_j. Note that the array name a is also a part of the edge label. This is because more than one array could be used for communication between a pair of loops. An integer variable II_i is introduced for every kernel i, and equation 1 bounds it between $IIMIN_i$ and $IIMAX_i$. Section 3.3 describes how $IIMIN_i$ and $IIMAX_i$ are determined. If the trip count of loop i is TC_i, then the latency L_i of the LA implementing loop i can be approximated by equation 2.

For every edge $e_{a,i,j}$, an integer variable $d_{a,i,j}$ is introduced to denote the number of buffers that are synthesized for the array a. Let the overall throughput for the entire pipeline be denoted by τ. The latency of any loop i should be no more than the throughput τ, as shown in equation 3. Note that, if there is only one buffer for an array a, then the producer i cannot begin execution in the next task before the consumer j in the previous task is done executing. Effectively, the buffer is occupied for $L_i + L_j$ cycles. However, i and j can be overlapped across consecutive tasks if more than one buffer is allocated for array a. Given that $d_{a,i,j}$ denotes the number of buffers allocated for array a, equation 4 formalizes the above constraint. Equation 4 simplifies to $L_i + L_j \leq d_{a,i,j} \times \tau$ when there is a direct edge from i to j. In this case, $d_{a,i,j}$ can have a maximum value of 2, i.e., two buffers for the array a is sufficient to support maximal task overlap. However, i and j may not be directly connected, and there could be many paths (possibly of length longer than 2) from i

to j in the loop graph. In the general case, more than 2 buffers may be required for array a, and consecutive tasks use the buffers in a round-robin fashion.

Multiple loops can be combined into one multifunction LA. The assignment of IIs to the loops is independent of whether or not it becomes part of a multifunction LA. Suppose the number of kernel instances in the system specification function is p. There can be a maximum of p accelerators in the pipeline. This extreme case corresponds to the design where there are no multifunction LAs in the system. Binary variables $b_{i,j}$ are introduced to denote the assignment of loop i to the accelerator j. Equation 5 ensures that every loop is assigned to at most one LA. The latency of a multifunction accelerator now becomes the sum of the latencies of individual loops assigned to it, which should be no more than the overall throughput τ, as shown in equation 6. Equation 6 involves the product of a binary variable and an integer variable, and is non-linear. However, it can be linearized using auxiliary variables $TL_{i,j}$ as shown in equation 7. P is a suitable large constant in equation 7.

Equations 1–7 can provide a valid selection of IIs for the loops and number of buffers for the arrays, and a valid combination of loops into multifunction LAs. An objective function is designed such that the cost of the overall pipeline is minimized. First, variables CL_i are introduced to denote the cost of a single function LA that just implements loop i. Note that this cost depends solely on II_i. However, CL_i is not a linear function of II_i as shown in Figure 5. To overcome this, a one-hot encoding of II_i is used to express CL_i in terms of II_i as shown in equations 8–10. Note that $Cost(i,k)$ denotes the cost of a single function LA implementing loop i with II=k, and is a constant. The cost C_j of a multifunction LA j is a function of the loops assigned to it. It cannot be expressed as a simple linear function, and can be obtained only by actually synthesizing the multifunction LA. To approximate C_j, we introduce two variables, $SUMC_j$ and $MAXC_j$ in equations 11 and 12 which represent the sum of costs of single function LAs that implement loops assigned to j, and the maximum of costs of those LAs, respectively. Equation 11 is not linear. However it can be linearized using the same technique shown in equation 7. The cost C_j of a multifunction LA j is bounded by $MAXC_j$ and $SUMC_j$. As an approximation, we use equation 13 to represent C_j. The actual cost of a multifunction might vary widely between $MAXC_j$ and $SUMC_j$. If the loops that are combined are exactly identical, the cost will be same as $MAXC_j$. If they have less overlap in terms of kinds of operations in the loop body, then the cost of combined LA will be close to $SUMC_j$. Empirically, we found setting C_j at the midpoint between $MAXC_j$ and $SUMC_j$ worked satisfactorily for a wide range of applications. Getting a better estimate for C_j without actually synthesizing the multifunction LAs will strengthen the objective function, and is a subject of future research.

The other component of cost, the array buffers, does not depend on assignment of loops to a multifunction LA, and can be easily calculated from $d_{a,i,j}$'s. Since $d_{a,i,j}$ denotes the number of buffers allocated for the array a, the overall cost of memory buffers in the system is given by equation 14. $Memcost(a)$ is a constant depending on the size and bitwidth of the array a. The objective of the ILP solver is to minimize the sum, $sum_{i=1}^{p} C_j + BUFCOST$, subject to the constraints given by equations 1–14.

3.3 Implementation

We use the SUIF compiler infrastructure [20] to process the input specification and build the loop graph. High level information like trip counts of the kernels, the sizes and bitwidths of arrays used for communication, and the communication structure between kernels is gathered in a SUIF pass. The application is converted to assembly format, which is the input to the Trimaran [18] compiler tool chain. Operation level data flow analysis is performed to determine $IIMIN_i$'s for each kernel. Note that the minimum achievable II is a function of recurrence cycles in the loop body. A cost sensitive modulo scheduler is used to schedule the loop bodies at increasing IIs, beginning with $IIMIN_i$. The datapath for the LA is derived from the schedule and synthesized using Synopsys design compiler. Thus, $Cost(i,k)$'s, the gate count estimates for the LA implementing loop i at II=k, are obtained. As II is increased, $Cost(i,k)$ decreases up to a point. As described in Section 3.1, the cost of LAs begins to increase at higher IIs. The value of $IIMAX_i$ is set to the point where this happens. $Memcost(a)$'s are computed using the Artisan memory compiler which synthesizes SRAMs for the communication arrays. Using the constants obtained as above and the throughput specification, the ILP program is formed and solved using the CPLEX solver. Thus, the II_i's for all loops and $d_{a,i,j}$'s, the number of buffers allocated for the arrays, are obtained.

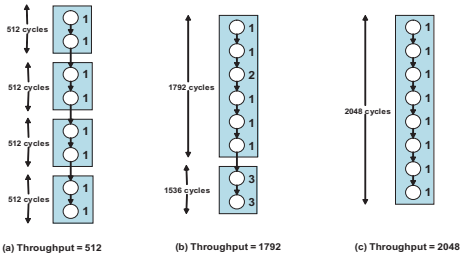


Figure 8: Pipeline configurations for Simple

4. CASE STUDIES

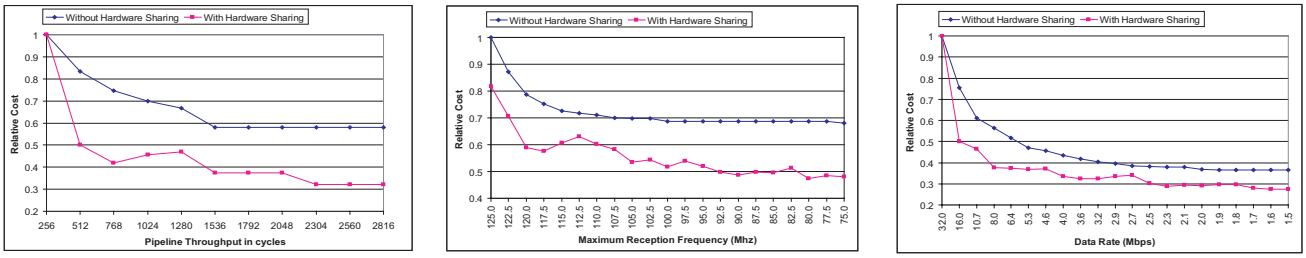
This section presents accelerator pipelines for different applications designed using our system. `Simple` is a synthetic application included to illustrate how different components of cost are optimized. `Beamformer` and `FMRadio` are streaming applications from the VersaBench [11] suite. For each application, accelerator pipelines were designed with varying throughputs, and the system area results are presented.

Simple. This applications consists of a sequence of eight loops, each with a trip count of 256 and containing a mix of add and multiply operations. The loop iterations are completely parallel, allowing modulo schedules with IIs of 1. The highest performance pipeline for this application can have a throughput of 256 cycles. Figure 7(a) shows the cost of accelerator pipelines designed for `Simple` for throughputs varying from 256 cycles to 2816 cycles.

The costs shown are gate counts, and are relative to the cost of the pipeline with a throughput of 256 cycles. The set of points labeled "Without Hardware Sharing" correspond to the designs with no multifunction LAs. The ILP formulation was modified to get the lowest cost design without combining any loops. We see that multifunction LAs are able to achieve significant cost savings through sharing hardware across multiple loops. On an average, 40% cost savings are achieved by hardware sharing.

Figure 8 illustrates the use of multifunction LAs in the accelerator pipeline. The loop graph is shown with the IIs next to the nodes. Boxes indicate which loops were combined into multifunction LAs. When the required throughput is 512 cycles (Figure 8(a)), adjacent loops are combined into multifunction LAs, resulting in only 4 stages in the pipeline. Each multifunction LA now has a latency of 512 cycles, as it implements two loops at II of 1. Figure 8(b) illustrates the complexity of designing a pipeline for a large application. When the required throughput is 1792, the number of possibilities are too many for a designer to manually explore. Our automated method systematically derives the pipeline configuration with minimum cost as shown.

FMRadio. FMRadio is a software implementation of an FM Radio receiver. Figure 7 shows the costs of the accelerator pipeline for FMRadio capable of receiving varying maximum frequencies. The frequencies were derived assuming a clock rate of 200 MHz.



(a) Simple (b) FMRadio (c) Beamformer

Figure 7: Cost vs. throughput

Our implementation has 27 loops in the loop graph. On an average, multifunction LAs cause 20% cost savings by sharing resources across multiple loops. Even though the general trend is lower cost for lower throughput configurations, the "With hardware sharing" curve is not smooth. For example, the cost for 112.5 MHz configuration is about 10% higher than the 117.5 MHz configuration. The approximation adopted using Equation 13 causes this sub-optimality in some cases. However, the cost of the pipeline with multifunction LAs is still much lower than without hardware sharing. The memory component of the cost was 45% of the overall cost for low II designs, indicating that more FUs are required in the data path in high performance implementations. The memory component of the cost was up to 70% of the overall cost for high II, low performance designs.

Beamformer. Beamformer is a spatial filter operating on data from an array of sensors. Again, the data rates shown in Figure 7(c) are derived assuming a 200 MHz clock. Our implementation has 10 loops in the loop graph. 15% cost savings are achieved due to hardware sharing on an average. The memory component of the overall cost ranged from 60% for low II designs to 70% for high II designs.

Discussion. Some of the salient points about the designs generated in our case studies are as follows:

- Multifunction LAs reduce cost over having two independent low performance accelerators. As Figure 5 shows, the reduction in cost of a single accelerator is not linear with respect to decrease in performance. However, combining two similar loops can halve the cost. Thus, the system combines as many loops as possible to save cost.
- Pipelines with a low prescribed throughput often contain overdesigned LAs. Increasing the IIs beyond a certain point only increases the cost. Therefore, the system just picks a higher performance LA to save cost.
- Memory buffers are a significant portion of overall cost. In some cases, reducing the number of buffers to 1 while retaining the producer and consumer loops at high performance saved more cost than synthesizing low performance LAs for the loops with two buffers for the array.

5. CONCLUSION

This paper presents *Streamroller*, an automated system for designing accelerator pipelines for compute-intensive streaming applications at a user-prescribed performance level. Synthesis consists of designing a set of communicating loop accelerators and buffers for storing intermediate results, and orchestrating the pipeline execution. Multifunction accelerators are used to reduce cost through hardware sharing between pipeline stages.

Three case studies are presented to highlight the capabilities and effectiveness of the design system. The studies reveal three important findings about accelerator pipelines: multifunction LAs are found to be more cost efficient than having multiple independent, lower performance accelerators; reducing the performance of a single LA below a certain point only serves to increase cost; and the memory buffers are significant and the configuration must be optimized to minimize overall system cost.

6. ACKNOWLEDGMENTS

Thanks to the anonymous referees who provided excellent feedback. This research was supported in part by ARM Limited, the National Science Foundation grants CCR-0325898 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

7. REFERENCES

[1] F. Brewer and D. Gajski. Chippe: A system for constraint driven behavioral synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(7):681–695, July 1990.

[2] C. M. Chu et al. Hyper: An interactive synthesis environment for high performance real time applications. In *Proc. of the 1989 International Conference on Computer Design*, pages 432–435, Oct. 1989.

[3] R. Composano. Design process model in the yorktown silicon compiler. In *Proc. of the 25st Design Automation Conference*, pages 489–494, Dec. 1988.

[4] S. Devadas and R. Newton. Data path synthesis from behavioral descriptions: An algorithmic approach. In *Proc. of the 1987 International Symposium on on Circuits and Systems*, pages 398–401, May 1987.

[5] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Synthesis of application specific multiprocessor architectures for process networks. In *Proc. of the 2004 International Conference on VLSI Design*, pages 780–783, Dec. 2004.

[6] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 219–230, Nov. 2005.

[7] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Increasing hardware efficiency with multifunction loop accelerators. In *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, Oct. 2006.

[8] P. Marwedel. The MIMOLA system: Detailed description of the system software. In *Proc. of the 30th Design Automation Conference*, pages 59–63, 1993.

[9] G. D. Micheli and D. C. Ku. HERCULES: System for high-level synthesis. In *Proc. of the 25st Design Automation Conference*, pages 483–488, May 1988.

[10] S. Note, W. Geurts, F. Catthoor, and H. D. Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proc. of the 28th Design Automation Conference*, pages 597–602, June 1991.

[11] R. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Technical Report MIT-LCS-TM-646, Massachusetts Institute of Technology, June 2004.

[12] L. Ramachandran, V. Chaiyakul, and D. D. Gajski. Vhdl synthesis system (vss) user's manual version 5.0. Technical Report ICS-TR-92-52, University of California at Irvine, June 1992.

[13] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[14] R. Schreiber et al. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[15] T. Stefanov et al. System design using Kahn Process Networks: The Compaan/Laura approach. In *Proc. of the 2004 Design, Automation and Test in Europe*, pages 340–345, Feb. 2004.

[16] R. Szymanek and K. Kuchcinski. Task assignment and scheduling under memory constraints. In *Proc. of the 26th Euromicro Conference*, pages 84–90, Sept. 2000.

[17] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn. The system architect's workbench. In *Proc. of the 25st Design Automation Conference*, pages 337–343, Dec. 1988.

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

[19] H. L. Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, 1991.

[20] R. P. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[21] Y. Xie and W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Proc. of the 2001 Design, Automation and Test in Europe*, pages 620–625, Dec. 2001.