Diastolic Arrays: Throughput-Driven Reconfigurable Computing

Myong Hyon Cho*, Chih-Chi Cheng†, Michel Kinsy*, G. Edward Suh‡ and Srinivas Devadas*
*Massachusetts Institute of Technology, †National Taiwan University, ‡Cornell University
*{mhcho, mkinsy, devadas}@mit.edu, †ccc@video.ee.ntu.edu.tw, ‡suh@csl.cornell.edu

Abstract— **Diastolic arrays are arrays of processing elements that communicate exclusively through First-In First-Out (FIFO) queues. FIFO virtualization units enable relaxed timing of data transfers, and include hardware support to guarantee bandwidth and buffer space for all data transfers, which may follow composite paths through the network. We show that the architecture of diastolic arrays enables efficient synthesis from high-level specifications of communicating finite state machines so average throughput is maximized. Preliminary results are presented on an H.264 decoding benchmark.**

I. Introduction

A diastolic array is a reconfigurable substrate that is meant to serve as a coprocessing platform to speed up applications or parts of applications that are throughput-sensitive and latency-tolerant. Diastolic arrays are coarser-grained than FPGAs but finer-grained than multicores. The adjective diastolic is used to refer to the relaxation of the heart between muscle contractions. Data transfers in diastolic arrays have more relaxed latency requirements than in systolic arrays, hence the name.

A diastolic array has a programmable processing element (PE) with a simple ISA, running on a fast substrate clock. Diastolic array processors communicate exclusively through networked First-In First-Out (FIFO) queue virtualization units that provide hardware support to guarantee bandwidth and buffer space for all data transfers. The architecture of diastolic arrays enables efficient synthesis from high-level specifications of communicating finite state machines so *average* throughput is maximized.

A design is represented as multiple processing modules (finite state machines) connected through point-to-point virtual FIFOs. Each virtual FIFO connects a fixed source-destination pair for one input-output pair; if two modules need multiple connections, each connection gets its own virtual FIFO. As we will show, virtual FIFOs provide means for efficient communication and synchronization among processing modules with focus on average case performance. FIFO-based communication naturally supports simple synchronization through waiting on inputs and backpressure; a processing module stalls if either an input FIFO is empty or an output FIFO is full.

Data transfers in a diastolic array are all statically routed and are allowed a varying number of clocks depending on the length of, and congestion in, the transfer path comprising a sequence of FIFO virtualization units (FVUs). FIFOs (that have room for more than one value) average out data-dependent variances in each module's execution and communication, and the performance of the design is determined by the module with the maximum average latency, not the worst-case input that causes the longest latency in a module.

During synthesis to diastolic arrays, FSM modules are assigned to processing elements (placement), "virtual" FIFOs used for communication between modules are realized as a sequence of FVUs (routing), modules are compiled into instructions for processors (compilation), and the routing logic within each PE is statically configured to implement the correct virtual FIFO routing, while guaranteeing bandwidth and buffer space (configuration).

A case study in Section II shows that averaging data-dependent variances is critical to achieving high throughput for applications such as H.264 decoding. A candidate architecture for a diastolic array is presented in Section III. The synthesis flow is presented in Section IV, and applied to an H.264 decoder in Section V. Related work is summarized in Section VI. Section VII concludes the paper.

II. Motivating Application

A. Example: H.264 Decoder

H.264 is widely being used for video compression. Figure 1 shows a specification of H.264 decoder; each module has data-dependent latencies. We will examine the entropy decoder module and inter-prediction modules and show that an architecture which targets average case latency has a performance benefit over a conventional pipelined design that assumes the worst case.

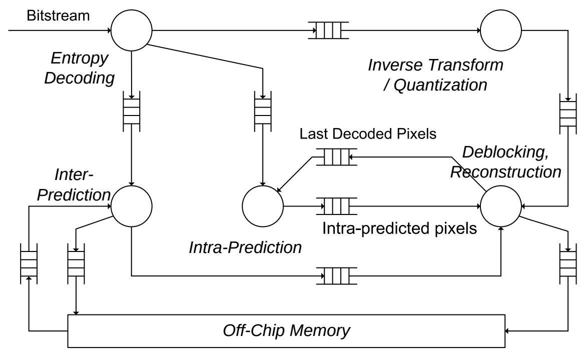


Fig. 1. High-level module description of H.264 decoder.

The entropy decoder module in H.264 decoder performs context-adaptive variable length decoding (CAVLD) that uses 20 different code tables. Each image block from the input stream requires access to different code tables and the number of table lookups varies significantly across inputs. Because the table lookup and following computations take up the majority of time in entropy decoding, we can assume that the latency of the entropy decoder module is proportional to the number of table lookups for each input (image block). In the inter-prediction module, the latency is dominated by the number of pixels it reads from reference frames, which depends on the input block's offset from the reference block (motion vector). Therefore, the latency of inter-prediction module is again highly dependent on the input block and can be different for each input. Table I shows the profiling results of both modules for the input stream 'toys and calendar', illustrating the large difference between the worst-case latency and the average-case latency.

TABLE I
H.264 PROFILING RESULTS FOR A STANDARD INPUT STREAM.

Entropy Decoder		Inter-prediction	
#lookups	Occurrence %	Data read (bytes)	Occurrence %
0~5	43.5%	0~239	0.01%
6~11	38.6%	240~399	9.3%
12~17	14.4%	400~559	19.6%
18~23	3.0%	560~719	67.5%
24~	0.4%	720~	0.4%
Average	7.56 lookups	Average	589.3 bytes
Maximum	32 lookups	Maximum	954 bytes

If each module is completely decoupled through infinite size FIFOs, the average-case design on a diastolic array will have 40% to 80% lower latency (or higher throughput) compared to the pipelined design that always performs the maximum number of operations, i.e., performs 32 lookups in the entropy decoder and reads the entire reference frame in the inter-predicton module. In practice, however, the throughput could be lower than the average case if the FIFO is not large enough because individual module latencies vary from input to input.

III. DIASTOLIC ARCHITECTURE

This section describes a candidate diastolic array architecture. The architecture provides guarantees of bandwidth and buffer space for all data transfers through:
(1) Non-blocking, weighted round-robin transfers of packets corresponding to different virtual FIFOs (VFIFOs) from one FIFO virtualization unit (FVU) to a neighbor,
(2) Ratioed transfer of packets corresponding to the same VFIFO from an FVU to its neighbors and in-order reception of said packets at FVUs to enable composite-path data transfers where sub-paths split and reconverge (cf. Figure 2(b)), and
(3) Allocation of FVU space to packets from particular VFIFOs to avoid deadlock and to maximize throughput.

A. Microarchitecture Overview

A diastolic array realizes the high-level computation model with a grid of processing elements (PEs) each with an attached FVU as shown in Figure 2 (a). In this architecture, all PEs operate synchronously using a single global substrate clock. FVUs are connected to neighboring FVUs and support many VFIFOs with synchronization mechanisms; from the PE's perspective, FVUs appear as many VFIFOs. In our candidate architecture, PEs are simple MIPS-like processors and the FVU network consists of 4 nearest-neighbor connections. In this architecture, each FVU can take up to 5 inputs (4 neighbors and the PE) and produce up to 5 outputs in each clock cycle. Peripheral FVUs interface with I/O pads.

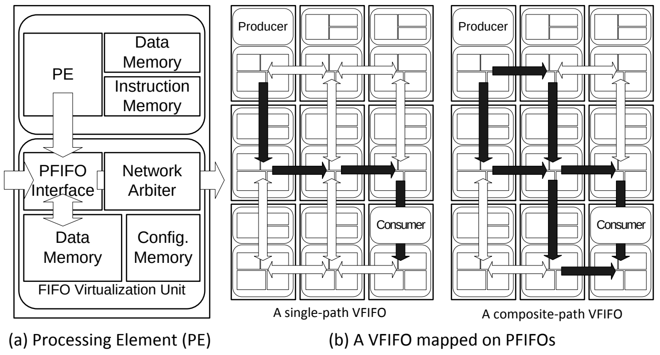


Fig. 2. A diastolic array architecture and data propagation through the 4-nearest-neighbor interconnect. (a) The processing elements consist of a computation unit and a FIFO. (b) A virtual FIFO can be routed using a single path or a composite path with sub-paths that split and reconverge.

Figure 2 (b) illustrates how a VFIFO is implemented with multiple FVUs – a single-path and a composite-path VFIFO marked by black arrows. In both examples, the VFIFO connects the top-left PE (producer) and the bottom-right PE (consumer). FVUs may route VFIFO packets along single- as well as composite-path routes (cf. Section III-C). The synthesis tool statically determines the routing and maps each VFIFO to corresponding FVUs along possibly multiple paths where each hop has a pre-determined flow rate (cf. Section IV-F).

B. PE and PE-to-FVU Interface

Our initial PE design is based on a MIPS-like 32-bit 5-stage in-order processing core. The ISA for computation is almost identical to the MIPS ISA with a branch delay slot so that a standard MIPS compiler backend can be used to generate efficient PE code. We use the gcc backend for MIPS in our synthesis framework. The main difference between the PE and a traditional MIPS core is in its support for VFIFO mechanisms; our PE supports additional instructions for VFIFO communication. The PE-to-FVU interface supports either an enqueue or a dequeue in each cycle.

C. FIFO Virtualization Units (FVUs)

FVUs implement the VFIFOs and synchronization based on backpressure. An FVU has to perform two main functions, namely, allocate buffer space for VFIFO packets that cannot be used by other VFIFOs, and route VFIFO packets with appropriate rates. The routes may be single- or composite-path routes, with the latter requiring increased FVU complexity. When transferring data for a VFIFO, an FVU must ensure that the receiving FVU has space available for the corresponding VFIFO.

Each FVU has one data memory that is shared among all VFIFOs mapped to the FVU. The synthesis tool statically partitions a large part of this data memory among VFIFOs. Each VFIFO assigned to an FVU has a partition size of at least one packet and these partitions are exclusively used for each VFIFO while the remaining data memory can be shared by all of the VFIFOs. In this way, the synthesis tool can guarantee that each virtual FIFO has the necessary number of FIFO entries to avoid deadlock no matter what the traffic pattern is. Further, the synthesis tool tries to allocate buffer space to achieve the maximum transfer rate for each VFIFO across the corresponding FVUs (cf. Section IV-F.2).

In each substrate cycle, an FVU can receive data from up to 5 sources, the attached PE and 4 nearest-neighbor FVUs, and send data to up to the same 5 destinations. The FVU-to-FVU interface is used to forward data from the source PE to the destination PE; in each substrate cycle, the FVU selects one VFIFO for each subsequent hop in a weighted round-robin fashion and forwards its data. This is done in a *non-blocking* fashion; if there is no data available for a VFIFO, or if the receiving FVU does not have an entry available for the particular VFIFO, the next VFIFO is selected. The weights are determined after the routing step (cf. Section IV-F) and applied in the configuration step (cf. Section IV-G).

Each FVU has four possible neighboring FVUs. For each of these links, a list of VFIFOs that share this link is generated by the synthesis tool after the routing step. For each link, a weighted round-robin send algorithm is used to schedule packets, where the weights are given by the flow rates. A VFIFO that does not have packets to send out or one that does not have space available in the subsequent FVU is passed over for the next VFIFO. If an FVU corresponds to a split point for a composite-path route for a VFIFO, the VFIFO is added to the list for multiple links, and the weight is the flow rate for that link as given by the routing step. Each of the packets of the VFIFO that come into this split FVU are marked by a marking algorithm to go in a particular direction in a deterministic way, in the ratios of the flow rates for the different directions. A packet marked to go in one direction is not sent in another. This is done so packets can be received in order at the reconvergent or destination FVU.

At an FVU that is a reconvergent point for a VFIFO, an acknowledgement algorithm allows an incoming packet from multiple neighbors to come in at appropriate ratios so as to guarantee in-order communication through this FVU, and to ensure that deadlock due to out-of-order packets will not occur. A later packet should not use up space in a reconvergent FVU on a composite path and block an earlier packet. The ratios in the acknowledgement algorithms depend on the throughput ratios of the split and reconvergent flows and are determined after the routing step as described in Section IV-G. FVUs are then configured with appropriate weights for the round-robin send algorithm and ratios for the marking and acknowledgement algorithms.

IV. Synthesis Flow

The synthesis flow is illustrated in Figure 3. While we described a candidate diastolic architecture in Section III, various PE microarchitectures and FVU network topologies can be supported with the synthesis flow described here.

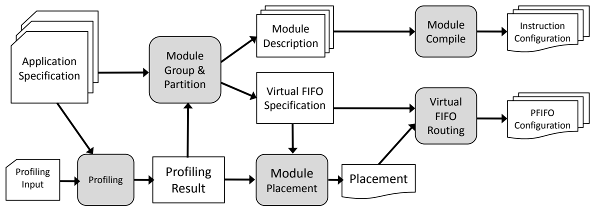


Fig. 3. Synthesis Flow for Diastolic Arrays.

A. Specification

Synthesis begins from a specification of the hardware design as finite state machine (FSM) modules described in C that communicate via VFIFOs. The specification will also provide minimum VFIFO sizes that ensure that the design does not deadlock. We will assume that for VFIFO i, z_i packets are required, with p_i bits in each packet.

Our specification is simpler than synchronous data flow [1], and similar to an *intermediate output* of a parallelizing compiler such as StreamIt [2] after parallelism extraction, but could also be directly written by a designer. Minimum requirements for FIFO sizes can be determined by compilers such as StreamIt [3].

A high-level view of the specification of an H.264 decoder was shown in Figure 1. The goal of synthesis is to maximize average throughput, which requires that bandwidth *and* buffer space be properly allocated to all VFIFOs.

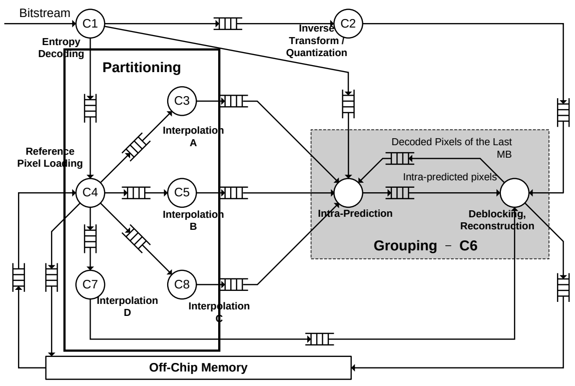


Fig. 4. Some modules in the H.264 specification of Figure 1 are grouped together, others are partitioned across PEs.

B. Profiling Modules in the Specification

Profiling the specification provides the synthesis tool with information required for various steps. First, each module is simulated separately on an array PE and a histogram of module latency over different module inputs is produced, which gives us a range of latency as well as an average. These latencies are used in the module grouping and partitioning step (cf. Section IV-C). Profiling is also used to compute a rate distribution and average transfer rate d_i in bits per second for each VFIFO i, which is a key measure used in the routing step. In this step, the entire system is simulated, assuming a target system throughput is specified, and assuming that each VFIFO has large buffer space and does not share hardware with other VFIFOs. Achieving the d_i rates becomes the goal for the routing and the buffer allocation steps. For each VFIFO i, a distribution of buffer size and average buffer size m_i in bits that is required for sustaining the average transfer rate d_i is determined. This is derived from the variation in buffer sizes during system simulation.

C. Module Grouping and Partitioning

Based on the profiling results, modules are grouped or partitioned. Grouping involves assigning two or more modules on the same PE, while partitioning involves splitting a module across multiple PEs in order to exploit parallelism and reduce the effective average latency.

Grouping is done if there are more modules than PEs, or if there is tight feedback between modules. When two modules are run on the same PE, their latencies will increase to their sum. If the inverse of this combined latency is equal to or more than the VFIFO rates corresponding to each of these modules, the modules can be grouped together. Grouped modules are executed in interleaved fashion on a PE. If a module does not have inputs available, it cedes to the next module. After a module execution, if there is no space available in the PE's FIFO for the result, the module will stall until space is available.

If we are unable to obtain the target system throughput, the modules whose latencies are too high are targets for partitioning. We will not address automatic partitioning here.

An example of module grouping and partitioning for our H.264 example is shown in Figure 4. The partitioning was done manually, and the grouping automatically using a simple bin-packing heuristic. After grouping and partitioning, the profiling step is run again to determine the new module and VFIFO rates.

D. Compilation

Modules are compiled to a generic PE in a decoupled way to produce scheduling information and executables. Our PEs have an ISA that is a subset of the MIPS ISA and we used a MIPS compiler in our experiments of Section V. All communications to other PEs are through FVUs with FIFO control, which the compilation step is aware of only at the level of writing and reading data values to and from the VFIFO. A PE may wait on a given read (dequeue) instruction or a write (enqueue) instruction because of FIFO control.

E. Placement

The primary goal of placement is to find a placement of the modules such that a feasible route can be determined for each of the VFIFOs. A *feasible route* is a route for each of the VFIFOs that has enough bandwidth for each FIFO's average transfer rate d_i obtained through profiling, and therefore allows the system to achieve maximum average throughput.

We need to define a notion of routability in order to generate a good placement. A recursive graph partitioning approach can then be used to find a placement with high routability. Cutsize-based approaches such as the Kernighan-Lin algorithm [4] are not directly applicable because cutsize does not represent the total traffic of a set of modules placed in an array section. Even if the design is partitioned into two sections with a minimum cutsize, one of partitioned sections might have significant internal costs that result in poor routability. By using the sum of internal costs and external costs as a cost function rather than cutsize, as shown below, heuristic algorithms can be used to maximize routability of placements.

Given a set of PEs A with cardinality $|A|$, and a set of modules V_A placed on PEs in A, we define *routability* of this (partial) placement as

$$R(A) = \frac{|A|}{\sum\limits_{v_i, v_j \in V_A} w(i,j) + \sum\limits_{v_i \in V_A, v_j \notin V_A} (w(i,j) + w(j,i))}$$

where $\{v_i\}$ is the set of modules in the design, $w(i,j)$ is the total demand for bandwidth from module i to module j, and $v_i \in V_A$ iff v_i is placed on a PE included in A.

An effective heuristic to generate placements that are significantly more routable than random placements gives priorities to PEs based on residual capacities of connecting links. Modules with the highest total demand are iteratively placed on PEs with the highest priorities. Since placement is done after module grouping, at most one module is placed on each PE. When a module is placed on a PE the capacities of links connecting to the PE are scaled down using a scaling parameter, and the priorities of neighboring PEs are recomputed. This heuristic spreads modules across the array to a degree determined by the scaling parameter. A number of placements are generated using different scaling parameters, and run through the rest of the synthesis process, and the best solution is selected.

For acyclic specifications, such as stream computations without feedback [2], [3], there are no hard requirements on the communication latency of VFIFO packets. In the H.264 application, modules both write and read off-chip memory, however, these two operations are so far apart in time that this feedback can be ignored during synthesis.

If feedback across modules occurs within a few substrate clocks, then the latency of communication paths can affect system throughput. An example of such feedback is bypass paths in a pipelined processor. The latency of communication can be included in the module latency, but we wish for other communication and the synthesis flow to not adversely affect this latency. We will need communication paths with minimum latency in the implementation (cf. Section IV-H), in addition to guaranteeing bandwidth and buffer space for all communications. In the next two sections, we assume that we have acyclic specifications, and then generalize our methodology in Section IV-H.

F. Routing of Virtual FIFOs

The route for each VFIFO is determined after module placement. The routing step chooses paths for each virtual

FIFO that require multiple hops using the transfer rates for each VFIFO. A VFIFO route can correspond to *multiple* paths through the mesh network, each with the same source and same destination. A route with multiple paths is referred to as a split flow. The source processor sends data at pre-determined ratios through multiple paths, and the data elements are received and processed in order at the destination processor. In addition, intermediate FVUs may need to collect packets for a given VFIFO and send them out at pre-determined ratios – the reconvergent points of Section III-C.

1) Multicommodity Flow Linear Program: We can formulate the search for a feasible route as a maximum concurrent multicommodity flow problem, where the commodities correspond to the data packets in each VFIFO. This problem is solvable in polynomial time using linear programming (LP) [5]. This formulation is slightly different from the one in Chapter 3 of [6] which minimizes the maximum link capacity required to support given bandwidth requirements.

Definition 1: **Maximum concurrent multicommodity flow:** Given a flow network $G(V, E)$, where edge $(u, v) \in E$ has capacity $c(u, v)$. There are k commodities K_1, $K_2, \ldots,$ K_k, defined by $K_i = (s_i, t_i, d_i)$, where s_i and t_i are the source and sink, respectively for commodity i, and d_i is the demand. The flow of commodity i along edge (u, v) is $f_i(u, v)$. Find an assignment of flow, i.e., $\forall (u, v) \in E \;\; f_i(u, v) \geq 0$, which satisfies the constraints:

$$\textbf{Capacity constraints}: \qquad \sum_{i=1}^{k} f_i(u, v) \;\leq\; c(u, v)$$

$$\textbf{Flow conservation}:$$

$$\forall i,\; \forall u \neq s_i,\; t_i \quad \sum_{(w,u)\in E} f_i(w, u) = \sum_{(u,w)\in E} f_i(u, w)$$

$$\forall i \quad \sum_{(s_i,w)\in E} f_i(s_i, w) \;=\; \sum_{(w,t_i)\in E} f_i(w, t_i) \;\leq\; d_i$$

and maximizes the minimal fraction of the flow of each commodity to its demand:

$$T \;=\; \min_{1\leq i\leq k} \frac{\sum_{(s_i,w)\in E} f_i(s_i, w)}{d_i} \qquad (1)$$

We assume $\forall i,\; \forall w \;\; f_i(w, s_i) = f_i(t_i, w) = 0$ in the formulation. k is the number of VFIFOs. The capacities for the edges in the network are equal to the bandwidth of the link between adjacent FVUs in the diastolic array architecture. The link bandwidths $c(u, v)$ are all equal to L by default, but may be set to lower values (cf. Section IV-H). The source for commodity i is the source processor in the given placement for VFIFO i, and similarly the destination. The demand d_i for VFIFO i is the average transfer rate for that FIFO obtained through profiling. To accommodate bursts in VFIFO traffic, we also maximize the minimum residual capacity over all links, namely,

$$S \;=\; \min_{(u,v)\in E} c(u, v) - \sum_{i=1}^{k} f_i(u, v).$$

We would like T to be 1 and S to be large. One strategy is to run the LP maximizing $x \cdot T + S$, where x is a large constant.

2) Buffer Allocation Linear Program: We still need to incorporate the requirements on buffer sizes for deadlock avoidance and to achieve the d_i rates. After we run LP and obtain the $f_i(u, v)$'s, we have a flow for each VFIFO i, i.e., a set of paths with particular rates on each link in the diastolic array. We can run another linear program to perform buffer allocation along each chosen VFIFO route.

We first determine the FVUs in the PE's that correspond to each VFIFO's commodity flow.

$$\forall u,\; \forall i \;\; iff\; \exists v\; s.t.\; (f_i(u, v) > 0 \;or\; f_i(v, u) > 0)\; g_i(u) = 1$$

Note that for a given flow the g_i's are constants that are either 1 or 0, corresponding to whether or not packets from the VFIFO will reside in the FVU corresponding to PE u.

The buffer size in PE u that we wish to allocate to VFIFO i in terms of the number of packets is denoted $l_i(u)$, and these are the variables in the LP. The available buffering in a PE u is $b(u)$ bits. In our candidate architecture, these are all equal to M bits, however, critical FIFOs (cf. Section IV-H) may be assigned some of the buffer space prior to running the LP. Recall that m_i is the average buffer size required for VFIFO i to sustain its transfer rate d_i, as obtained by the profiling step (cf. Section IV-B), z_i is the number of packets in VFIFO i that ensures that deadlock will not occur and p_i is the packet size for the virtual FIFO packet in bits (cf. Section IV-A).

Definition 2: **Optimal Buffer Allocation:** For each VFIFO i, we are given a buffer size requirement m_i and a set of FVUs $g_i(u) = 1$ that are on the VFIFO's route. We are given available buffer sizes $b(u)$ for each PE u. Find an assignment of buffers $l_i(u)$ for each VFIFO i that satisfies:

$$\textbf{FVU Buffer Limit}: \quad \forall u \;\; \sum_{i=1}^{k} p_i \cdot l_i(u) \;\;\leq\;\; b(u)$$

$$\textbf{Deadlock Avoidance}: \quad \forall i,\; \forall u \;\; l_i(u) \;\geq\; g_i(u)$$

$$\forall i \quad \sum_{w:g_i(w)=1} l_i(w) \;\geq\; z_i$$

$$\textbf{Allocation}: \qquad \forall i \;\; p_i \cdot \sum_{w:g_i(w)=1} l_i(w) \;\leq\; m_i$$

and maximizes the minimal fraction of the allocated buffers of each commodity to its demand for buffering:

$$U \;=\; \min_{1\leq i\leq k} \frac{p_i \cdot \sum_{w:g_i(w)=1} l_i(w)}{m_i} \qquad (2)$$

The deadlock avoidance requirement comes from the *specification*; there should be at least p_i bits worth of dedicated space available in *each* FVU that is used by a VFIFO to route its packets, and further the set of FVUs implementing the VFIFOs should provide z_i packets worth of space. Since the FVUs in the diastolic array will accept packets from VFIFOs whose limit has not been exceeded, while possibly rejecting packets from other VFIFOs, deadlock will not occur in the array implementation. There is a limit on the number of

VFIFOs that can be mapped to an FVU due to the deadlock requirement.

On top of the deadlock avoidance requirement, we would also like to allocate enough buffer space for each VFIFO so the transfer rates can be met, while ensuring fair allocation across VFIFOs (cf. Eqn. 2). Note that the $l_i(u)$'s corresponding to $g_i(u) = 0$ can be set to zero. Of course, we need the $l_i(u)$'s to be integers, so we will truncate or round up the values after we obtain the solution to the LP.

Given a user-specified amount of CPU time, we choose many solutions with corresponding maximum T for the first LP, to maximize U. We also repeat this process for many placements, and pick the solution with the maximum T, breaking ties by choosing the one with maximum U.

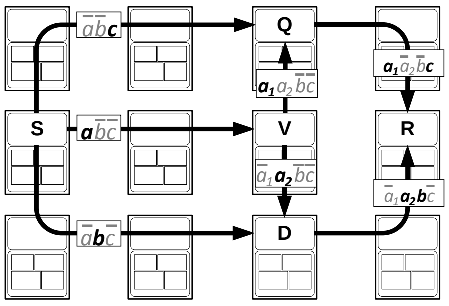
G. *Configuration*

Once we have found a feasible route, or settled on a throughput less than the maximum, the final step is to generate configurations for each processor and FVU. The PEs are configured with the compiled code of the modules that will execute on the PE.

The flow rates determined by the LP are made integral with appropriate multiplications to keep the ratios as close as possible in the case where an integral flow is not provided by LP. (There is no need to round up or truncate these rates.) Each link in each FVU is first configured with the set of VFIFOs that share this link. Each link is configured to send out packets in a weighted round-robin fashion over all the VFIFOs that share this link. If an FVU is a split point for a VFIFO, the marking algorithm for incoming packets is configured with appropriate ratios for the different links that this VFIFO's packets will depart the FVU on. At reconvergent FVUs including the destination, packets corresponding to each VFIFO are received in order, by choosing the packets from different links using an acknowledgement algorithm.

Consider the example of Figure 5. A FVU may split a flow of packets two to four ways. For a three-way split in ratio $a^R : b^B : c^T$, the marking algorithm at FVU S will mark the first a packets to the right, the next b packets to the bottom, and the next c packets to the top, repeatedly. Note that these a packets will contend for bandwidth in the link to the right with other VFIFO packets, and a weighted round-robin send algorithm will periodically send these packets out. The right sub-flow is represented as $a\overline{b}\overline{c}$ indicating that the first a packets from the source were picked, the next b packets were sent somewhere else, etc; this pattern repeats indefinitely. The right sub-flow is split again in FVU V in the ratio ${a_1}^T : {a_2}^B$; the patterns generated will be $a_1\overline{a_2}\overline{b}\overline{c}$, and $\overline{a_1}a_2\overline{b}\overline{c}$. At FVU Q, the sub-flows $a_1\overline{a_2}\overline{b}\overline{c}$ and $\overline{a}\overline{b}c$ converge. The acknowledgement algorithm at Q will pick a_1 packets from the bottom and c packets from the left, repeatedly. This produces a sub-flow represented as $a_1\overline{a_2}\overline{b}c$. Finally, the FVU at destination R will pick a_1 packets from the top, $a_2 + b$ packets from the bottom, and c packets from the top, repeatedly.

As a final step of configuration, buffer space constraints for each VFIFO that is assigned to an FVU are specified.



$\overline{a}\overline{b}c$
Q
$a_1\overline{a_2}\overline{b}c$
$a_1 a_2 \overline{b}\overline{c}$
S
$a\overline{b}\overline{c}$
V
R
$\overline{a_1}a_2\overline{b}\overline{c}$
$\overline{a_1}a_2 b\overline{c}$
$\overline{a}b\overline{c}$
D

Fig. 5. Configuring Marking and Acknowledgement Algorithms for Composite-Path Routes

H. *Minimizing Latency of Virtual FIFOs*

As described in Section IV-F, all VFIFOs are guaranteed bandwidth and buffer space during synthesis, but we make no guarantees about latency. VFIFOs with longer paths (many hops) will have greater latency. This will not matter when there are no tight feedback paths.

While we cannot make latency guarantees about all VFIFOs or even a large number of VFIFOs, we can provide minimum latencies for a few critical VFIFOs, associated with feedback and identified during profiling whose increased latency will directly degrade performance. The connecting module pairs corresponding to critical VFIFOs are kept in the same partition during the placement step (cf. Section IV-E) for as long as possible, so VFIFO lengths are minimized. Prior to the routing LP step, a direct route is chosen for each of these VFIFOs, with no splits to avoid packet reordering at the destination. The bandwidth of each of the links $c(u, v)$ in the route is reduced by the corresponding d_i, and the buffer space in each of the n FVUs comprising the chosen VFIFO route is reduced by $\max(m_i/n, p_i)$. The two LP's are run as before to produce routes and buffer allocations for the remaining VFIFOs.

V. Results on an H.264 Decoder

The profiling result of the H.264 decoder shows that Inter-Prediction occupies most of the computation time. Therefore, modules of H.264 decoder were grouped and partitioned as shown in Figure 4 to enhance throughput by increasing parallelism. Figure 6 shows throughput demands of each VFIFO given by the profiling step assuming a 1 GHz substrate clock.

A number of candidate placements were generated using the heuristic of Section IV-E. For each placement the capacity and the flow conservation constraints were generated and this LP problem was solved by ILOG CPLEX. Figure 7 provides routing results for different link bandwidths. The throughput of each route is indicated. When the link bandwidth is 200MB/s, a feasible route without composite paths is found. Composite paths allow the fulfilment of the demanded throughput with smaller link bandwidth, such as 100MB/s (Figure 7(b)) and 60MB/s (Figure 7(c)). In Figure 7(b), for example, the route from C4 to C9 is split because the link between them cannot deliver more than 100MB/s. However, if the link bandwidth is too small, the routing algorithm will determine that there is no

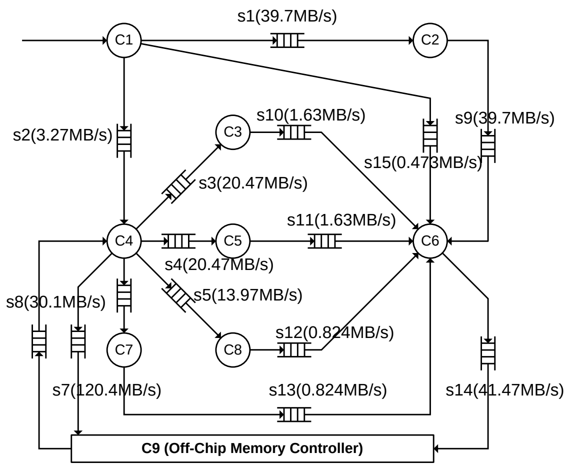


Fig. 6. Demand throughput of each VFIFO in H.264 decoder

feasible route, and report the best route for the given network (Figure 7(d)).

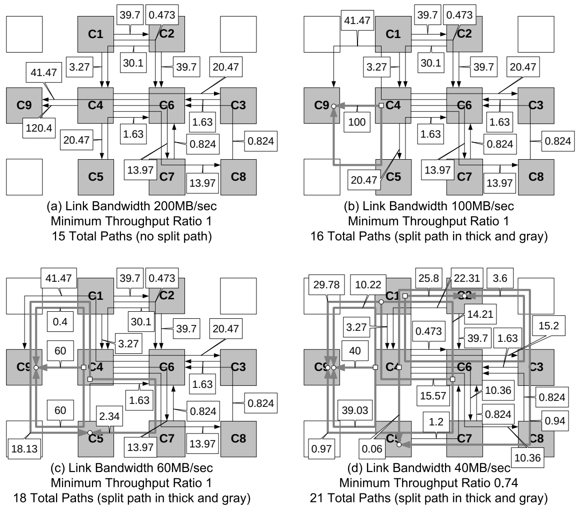

(a) Link Bandwidth 200MB/sec
Minimum Throughput Ratio 1
15 Total Paths (no split path)
(b) Link Bandwidth 100MB/sec
Minimum Throughput Ratio 1
16 Total Paths (split path in thick and gray)
(c) Link Bandwidth 60MB/sec
Minimum Throughput Ratio 1
18 Total Paths (split path in thick and gray)
(d) Link Bandwidth 40MB/sec
Minimum Throughput Ratio 0.74
21 Total Paths (split path in thick and gray)

Fig. 7. Routing results of H.264 decoder on a 4x4 array for different link bandwidths. The bottom four PEs are not shown.

Of course, different placements result in different routes. For example, Figure 8 illustrates how a placement can give a better routing result than another. When there is no feasible route as in Figure 7(d), the placement that maximizes the minimal fraction of the throughput is chosen from amongst the generated candidate placements (cf. Section IV-E).

The total synthesis time is very fast – a few seconds for this example. The time required to synthesize a Verilog description of H.264 to an FPGA is approximately 46 minutes for logic synthesis and 52 minutes for place and route [7]. Efficient synthesis is enabled because the specification deals with packets rather than bits, because compilation to processors is fast, and because the synthesis algorithms used here are efficient. We note that profiling took 2 minutes for this example.

After finding a feasible or the best route, the synthesis tool allocates buffers for VFIFOs in each FVU. Taking the route in Figure 7 (c), Table II summarizes z_i, p_i and m_i values as

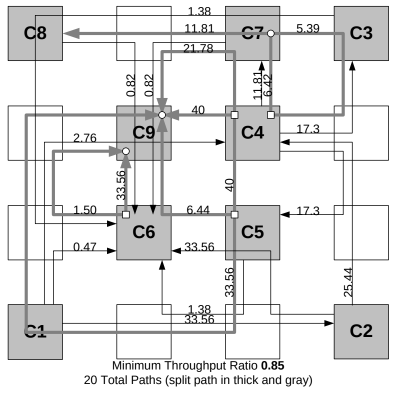

C8
C7
C3
C9
C4
C6
C5
C1
C2
Minimum Throughput Ratio 0.85
20 Total Paths (split path in thick and gray)

Fig. 8. Routing results of H.264 decoder on a 4x4 array for a different placement and with the link bandwidth of Figure 7(d).

defined in Section IV. Here m_i was obtained from the profiling step, and z_i and p_i from the specification. The buffer allocation result is given in Figure 9.

TABLE II
BUFFER SIZES FOR H.264.

virtual FIFO	p_i (bits)	m_i (bits)	z_i (packets)
s1	128	512	1
s2	512	1536	1
s3-s6	680	2024	1
s7	768	1536	1
s8	15552	31104	1
s9	3072	6144	1
s10-s11	1024	2048	1
s12-s13	512	1024	1
s14	3072	6144	1
s15	96	384	1

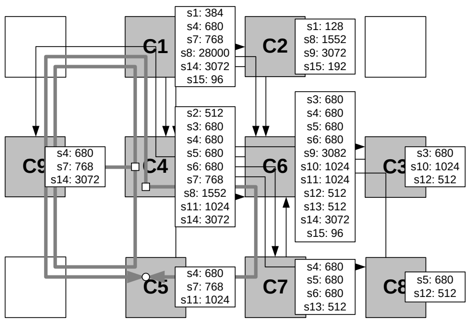


Fig. 9. Buffer allocation of H.264 decoder on a 4x4 array for the route of Figure 7 (c). The bottom four PEs are not shown.

Additional architectural details and results on a processor performance modeling benchmark [8] that includes tight feedback due to bypass paths can be found in [9].

VI. Related Work

Systolic arrays [10] have been used to efficiently run many regular applications such as matrix multiplication. These SIMD processors contain synchronously-operating elements which receive and send data in a highly regular manner through a processor array. Data transfer timing in MIMD diastolic arrays is more relaxed than in systolic arrays.

Dally's virtual channels [11] allocate buffer space for virtual channels in a decoupled way from bandwidth allocation; diastolic arrays guarantee bandwidth as well as buffer space, and implement multiple-hop, virtual composite channels. iWarp [12] implemented virtual channels across single links.

Diastolic arrays are simpler than commercial multicores or architectures such as Raw [13] and Tilera [14], and also target a smaller class of throughput-sensitive, latency-insensitive applications. Unlike Raw, diastolic arrays allow sharing of physical FIFOs by virtual FIFOs in a non-blocking way for data transfers. Tilera has five different networks that interconnect tiles including a static network, whereas diastolic arrays implement a single logically static network that supports sharing of flows, split flows and buffer allocation. TRIPS [15] uses significantly larger cores that are 16-issue. Asynchronous Array of simple Processors (AsAP) [16] is a multicore processor for DSP applications, which consists of a 2-D array of simple processors connected through dual-clock FIFOs in a Globally Asynchronous Locally Synchronous (GALS) fashion. The FIFO sizes in AsAP are appreciably smaller than those in diastolic arrays; these FIFOs are mainly used to interface two clock domains and hide communication latencies rather than optimizing average case performance as in diastolic arrays.

Ambric [17] uses a circuit-switched network as opposed to a packet-switched network, with a small amount of FIFO buffering. Channels are set up by configuring the network much like in an FPGA, and synthesis to the Ambric chip is similar to FPGA synthesis, though significantly faster due to structure provided by the designer [18].

A multi-path routing strategy is presented in [19], which uses packet identifiers to avoid deadlock and requires multi-paths to be *non-intersecting*. (The composite-path of Figure 5 is *not* non-intersecting due to FVU D.) A set of non-intersecting paths is heuristically selected, and LP is run to find flow rates for each path. In contrast, our throughput-optimal LP determines paths and flow rates simultaneously.

VII. Conclusions and Limitations

By focusing on throughput, by requiring the specification to be written in a particular way, and by designing a diastolic array architecture with appropriate hardware mechanisms, we have developed a synthesis flow that we believe is significantly easier to implement and optimize than conventional reconfigurable substrate synthesis flows. Implementing FIFOs and processors using BRAM and CLBs on an FPGA is quite expensive, and so a custom hardware implementation of a diastolic array is necessary. We plan to more comprehensively evaluate candidate architectures on applications such as H.264 encoding and detailed processor performance modeling, prior to undertaking a hardware implementation. The architectural tradeoffs corresponding to supporting composite-path routes or only supporting single-path routes, and varying the ISA, FIFO or memory sizes need to be explored. Since LP may produce complex routes with highly composite paths, we are developing heuristic methods which can be finely tuned that use the LP formulation to determine upper bounds on throughput, and which limit the maximum number of VFIFOs that share a link. Finally, we need to characterize what applications are readily and naturally expressible as finite state machines interacting through FIFOs, and extend our synthesis flow or the architecture to deal with applications where average throughput varies significantly.

Acknowledgment

We thank Joel Emer, Vijay Ganesh, Mieszko Lis, Michael Pellauer, Bill Thies and David Wentzlaff for useful feedback.

References

[1] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[2] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *International Conference on Compiler Construction*, 2002.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, Pipeline Parallelism in Stream Programs," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, pp. 291–307, February 1970.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press/McGraw-Hill, 2001.

[6] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[7] K. Fleming, C.-C. Lin, N. Dave, Arvind, G. Raghavan, and J. Hicks, "H.264 Decoding: A Case Study in Late Design-Cycle Changes," in *Proceedings of the Sixth MEMOCODE Conference*, 2008.

[8] M. Pellauer, M. Vijayaraghavan, M. Adler, J. Emer, and Arvind, "Quick Performance Models Quickly: Timing-Directed Simulation on FPGAs," in *International Symposium on Performance Analysis of Systems and Software (ISPASS 2008)*, April 2008.

[9] M. H. Cho, "Diastolic Arrays: Throughput-Driven Reconfigurable Computing," Master's thesis, Massachusetts Institute of Technology, May 2008. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-504/memo504.pdf

[10] H. T. Kung, "Why Systolic Architectures?" in *Computer Magazine*, January 1982.

[11] W. Dally, "Virtual-channel flow control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 03, no. 2, pp. 194–205, 1992.

[12] T. Gross and D. R. O'Hallaron, *iWarp: anatomy of a parallel computing system*. Cambridge, MA, USA: MIT Press, 1998.

[13] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," in *IEEE Computer*, September 1997, pp. 86–93.

[14] David Wentzlaff et al, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sept/Oct 2007.

[15] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. K. Kim, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture," in *International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 422–433.

[16] Z. Yu, "High performance and energy efficient multi-core systems for DSP applications," Ph.D. dissertation, U. C. Davis, 2007.

[17] M. Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, Sept/Oct 2007.

[18] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 55–64.

[19] S. Murali, D. Atienz, L. Benini, and G. D. Micheli, "A Method for Routing Packets Across Multiple Paths in NoCs with In-Order Delivery and Fault-Tolerance Gaurantees," *VLSI Design*, vol. 2007, 2007.