ShiftQ: A Buffered Interconnect for Custom Loop Accelerators

Shail Aditya, Michael S. Schlansker
HP Laboratories Palo Alto
HPL-2001-255
October 10th , 2001*

E-mail: aditya@hpl.hp.com, schlansk@hpl.hp.com

ShiftQs are hardware structures consisting of registers and switches which buffer and transport operands among function units within custom hardware loop accelerators. ShiftQs help minimize buffering and interconnect costs by customizing the hardware to the given schedule and by intelligent sharing of register and interconnect resources. This paper describes the ShiftQ schema and a method to automatically synthesize them from modulo-scheduled loops. We also evaluate the cost savings by comparing them against traditional storage and interconnect mechanisms.

* Internal Accession Date Only Approved for External Publication?
To be published in and presented at CASES '01, November 16-17, 2001, Atlanta, GA
©Copyright ACM

ShiftQ: A bufferred interconnect for custom loop accelerators

Shail Aditya
Hewlett-Packard Laboratories
1501 Page Mill Road, MS 3L-5
Palo Alto, CA, USA
aditya@hpl.hp.com

Michael S. Schlansker
Hewlett-Packard Laboratories
1501 Page Mill Road, MS 3L-5
Palo Alto, CA, USA
schlansk@hpl.hp.com

ABSTRACT

ShiftQs are hardware structures consisting of registers and switches which buffer and transport operands among function units within custom hardware loop accelerators. ShiftQs help minimize buffering and interconnect costs by customizing the hardware to the given schedule and by intelligent sharing of register and interconnect resources. This paper describes the ShiftQ schema and a method to automatically synthesize them from modulo-scheduled loops. We also evaluate the cost savings by comparing them against traditional storage and interconnect mechanisms.

1. INTRODUCTION

With the recent explosion of smart appliances, mobile communication and hand-held digital media processing devices, the industry is faced with a tremendous design challange: how to design the wide variety of embedded electronic systems inside these devices with supercomputer performance requirements at consumer-level cost and bring them to market at a breakneck speed. The PICO (Program-In-Chip-Out) project[1, 11] addresses this challange by providing an automatic methodology to design programmable and non-programmable accelerators that are customized to applications starting from a high-level algorithmic description such as a C program.

This paper describes technologies used within PICO-NPA, the subsystem of PICO that designs non-programmable accelerators, to execute compute-intensive loop kernels. These custom accelerators offer far lower cost and higher performance than competing programmable approaches. This is accomplished using hardware optimizations that take advantage of very specific knowledge of the application. Cost-effective hardware for parallelizable loops is generated by customizing control, datapath, and storage within the accelerator to highly specific application needs. This allows a level of efficiency within each processor that is unmatched using general purpose systems. Further, for a broad class of highly-parallelizable loops, these accelerators can be efficiently expanded by adding identical processors to provide linear increases in performance.

A major obstacle in designing parallel accelerators is the design of the buffered interconnects that carry operands among function units. This paper describes techniques that greatly reduce buffering and interconnect costs by tailoring interconnects to specific application needs. Operand transport mechanisms have been developed for a variety of custom solvers [2, 3, 7]. However, unlike prior work, this paper focuses on transport mechanisms for software pipelined loops. This problem can be approached from one of two extreme viewpoints. A fully centralized view places all virtual registers in a common shared file. This maximizes the potential that multiple virtual registers utilize the same storage but also requires an expensive highly ported file which is impractical for highly-parallel machines. A fully decentralized view places each virtual register in its own physical register file. While this minimizes file porting, physical registers are not shared among virtual registers, and too many switches may be required to connect function units to a large number of physical files. The ShiftQ mechanism proposed in this paper takes an intermediate position between these two extremes, attempting to provide the best of both worlds in terms of hardware cost. We describe the architecture of this buffered interconnect and an automatic way of designing these interconnects for a given loop schedule. In addition, we evaluate the cost of these structures within PICO loop accelerators.

The outline of the rest of the paper is as follows. Section 2 states the problem in more detail. Section 3 decsribes the ShiftQ structure, while Section 4 describes an algorithmic way to obtain it from a given loop schedule. Section 5 compares our synthesis results on a set of benchmarks and Section 6 concludes.

2. LOOP ACCELERATION PROBLEM

A loop body consists of a network of operations that are repeatedly applied to a stream of input data to compute a stream of results. Within this network, edges connect operations and represent data flow from producers to consumers. We define the **initiation interval** (II) as the execution time delay between adjacent loop iterations. Using software pipelining techniques [5], II can be made much shorter than the time needed to complete one full iteration of the loop. This increases throughput of the computation by overlapping the computation of subsequent loop iterations.

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'01, November 16-17, 2001, Atlanta, Georgia, USA.
Copyright 2001 ACM 1-58113-399-5/01/0011 ...$5.00.

Software pipelining is traditionally used as a loop acceleration technique in conjunction with programmable VLIW processors that use multiported register files to store operands from the time that they are first computed until their final use. The use of rotating register files [4] allows register lifetimes to persist for multiple loop iterations without being overwritten in subsequent iterations. Multiple copies of the same virtual register (VR) corresponding to multiple loop iterations executing simultaneously are collectively referenced as **expanded virtual registers** (EVRs) [9]. EVRs provide a virtual stack of storage elements for each variable so that old values for an EVR are retained even after a new value is written to that EVR. EVRs "connect" a def-use network of operations in the sense that a value produced by any of the producing operations (defs) for a single EVR may reach any of the consuming operations (uses) for that EVR. When disconnected def-use networks artificially share a common EVR, register renaming is used to replace a single EVR with multiple EVRs in order to split this graph into distinct subgraphs. We assume that such renaming has already been performed on the loop code. Furthermore, we initially restrict our discussion to singly assigned EVRs where each EVR is computed by a single static producer because this represents a very common siutuation. A less common situation where multiple operations compute the same EVR, possibly under distinct predicates, will be considered later.

2.1 Custom Loop Accelerators

Custom hardware accelerators designed for software pipelined loops with $II = 1$ can take advantage of a natural one-to-one correspondence between hardware function units and operations within the program graph. If every operation in the loop body has a dedicated function unit which executes the operation once a cycle, then all units execute loop code with full efficiency. A similar correspondence can be established between dataflow edges in the program graph and actual hardware datapaths so that all datapaths can be used with full efficiency. Such designs are easy to generate because no heuristics are needed to decide which operations share function units or which operands share registers.

However, $II = 1$ designs are often too high in rate and too costly. Less expensive ($II > 1$) designs reuse resources, over time, to execute more than one operation. Function units execute multiple operations, hardware datapaths transmit multiple operands, and registers store multiple operands. Efficient techniques are needed to design custom hardware for $II > 1$ designs so that the cost of required hardware decreases with decreasing computational needs. Several algorithms have been developed in the past to determine a program schedule that specifies which operations should be placed on a common function unit and at what time these operations should occur [5, 10]. While scheduling should be performed in a hardware cost cognizant manner [6, 8], this paper assumes that a program schedule is fully specified and focuses on techniques to design efficient register structures to transport operands for a given software pipeline schedule.

2.2 Register Organizations in Custom Loop Accelerators

Figure 1 illustrates a processor that uses a unified rotating register file to store all operands. In this example, a program references five EVRs (V1, ..., V5) that are all held in a common file. While multi-ported files are very useful for programmable processors, they are often too expensive for highly-parallel and customized non-programmable processors. The use of multi-ported files imposes two significant costs. First, as the number of read and write ports into a multi-ported file is increased the area of the file increases and the performance of the file decreases. Thus, this approach does not efficiently acccommodate a large number of function units. Second, register files require controllers that generate an address for each port in order to select the desired register element at each time step. The need to generate these register addresses increases the complexity of non-programmable accelerators.

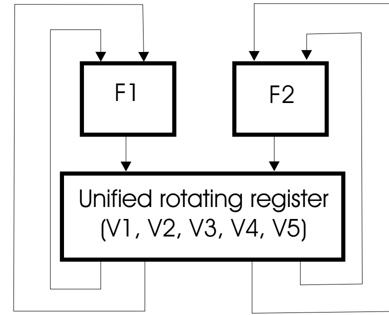

Figure 1: The Unified Register File Scheme.

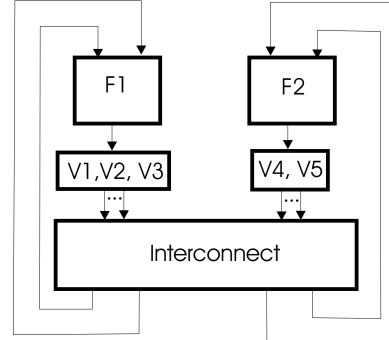

Figure 2: One ShiftQ per Function Unit Scheme.

Software pipelining packs operations on to function units so that most function units are heavily utilized. As a result, it is often the case that a dense stream of operands emerges from each function unit on each cycle. In this paper, we introduce **ShiftQ** as a hardware interconnect structure that is optimized to receive such a stream of operands emerging from a single function unit and transport it to receiving function units at a later time.

Figure 2 illustrates the use of separate ShiftQs each receiving a stream of operands from a function unit output. Each EVR in a loop schedule has an associated function unit on which it is computed and the EVR's values are stored within the ShiftQ associated with that unit. By *partitioning* register storage to one ShiftQ per function unit, this approach decreases the maximum number of input (one) and output ports to each ShiftQ[1]. However, each ShiftQ is still allowed to support more than one EVR. This often allows a single register element within a ShiftQ to support multiple

[1]Alternate partitionings are possible [3] but they preclude subsequent customization as discussed later.

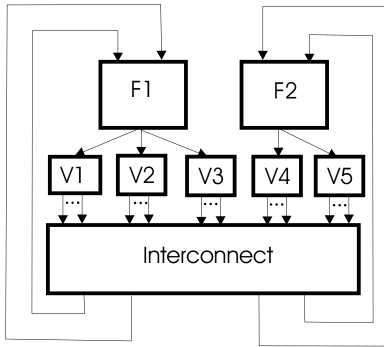

Figure 3: One ShiftQ per EVR Scheme.

EVRs whose lifetimes do not overlap. Together, this approach can substantially decreases the cost of higly-parallel, non-programmable accelerators as opposed to using a unified file.

Given a fully specified software pipeline schedule, hardware costs may be further decreased when a general register file structure is replaced by a *customized* ShiftQ that is specialized to a specific repeating computational pattern. In addition, such ShiftQs may also be specialized in bitwidth as each register element within the ShiftQ needs to be only as wide as the widest operand that it must support.

Another typical approach for designing fully customized datapaths is to assign a separate storage structure to each EVR in the program [2, 7] as shown in Figure 3. When operands are of uniform width, this approach increases overall cost as it precludes the sharing of register storage elements among multiple EVRs. However, when operands have differing bitwidth, the storage of these operands in separate ShiftQs allows each shiftQ to be separately specialized to the needs of its operands. This additional specialization can decrease cost over the one ShiftQ per Function unit schema.

This paper describes the ShiftQ schema; it explains the tradeoffs that favor the one ShiftQ per function unit approach and the one ShiftQ per EVR approach; it presents heuristics that reduce hardware cost by carefully selecting the best apprach and; it presents experiments that illustrate the benefits of these heuristics.

3. SHIFTQ SCHEMA

A ShiftQ is a set of linearly connected register cells as shown in Figure 4. The write port of the uppermost cell is connected to a function unit which generates a downward flowing stream of values into the cell under a modulo schedule. A set of control signals $C_0, \ldots, C_{II-1}$ are used to control the ShiftQ. These signals are called the **modulo phase bus**. This bus generates control signals such that each signal C_i is asserted on all cycles t where $i = t \bmod II$. One modulo phase bus signal is always true on each cycle as time "rotates" through various phases.

Loading of each cell in the ShiftQ is controlled by a **shift enable** (SE) control signal. When SE is low a cell retains its old value; when SE is high a new value is latched from input. The ShiftQ shown in Figure 4 is currently configured as a conventional shift register; the ShiftQ shifts on every cycle because SE for every cell is driven by the OR of all the phase bus signals. But in general, the ShiftQ en-

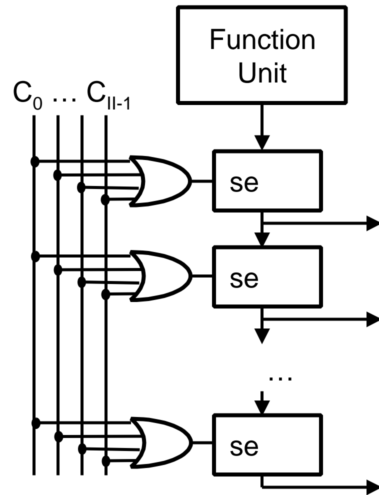

Figure 4: The ShiftQ schema.

tries need not be shifted every cycle. By eliminating terms, each cell may be "programmed" to shift as infrequently as needed. This scheme has some attractive properties. First, operands are not replicated within the ShiftQ, a single copy of each operand is held within its corresponding ShiftQ. Second, buffering is minimized by sharing cells among different operands within a single ShiftQ while keeping each operand alive a minimum number of cycles. Finally, reducing the number of cells also helps to reduce interconnect costs. When two operands are read (at distinct moments in time) from the same register and by the same function unit, a single data path supports both data transfers.

We now intuitively describe how a particular ShiftQ structure such as that shown in Figure 4 may be designed while minimizing its size. Consider the topmost ShiftQ cell in Figure 4. A new value is shifted into this cell whenever a new value is generated by the function unit. On other cycles when the function unit generates no value, the ShiftQ does not shift. Given a modulo schedule, this is easily accomplished by removing a phase bus signal from the OR gate which controls the topmost cell for every cycle in which the function unit produces no result. Subsequent cells hold values which shift out of prior cells. A shift into a subsequent cell occurs *only* when the value within the previous cell is live (still needed by the program) and when the previous cell must shift. The repeated application of these simple rules, for each cell, provides a concise definition of a ShiftQ structure that uses a minimal number of registers and is easy to control. The shifting control is implemented by eliminating OR gate input terms in the control logic for each cell. A final cell terminates the ShiftQ when no data needs to be shifted out.

With a clear definition of exactly when all shifts occur within a ShiftQ, it is also possible to track where each value within the ShiftQ lies at each moment in time. When a value residing within a ShiftQ is read by an operation at a specific moment in time, the ShiftQ register cell holding the value is identified, and a hardware data path is connected from that cell to the function unit port which uses the data. In this manner, ShiftQs can support the sequencing of operands between function units in an $II > 1$ loop accelerator.

3.1 A ShiftQ Example

Figure 5 shows an example ShiftQ design with $II = 8$ and housing three EVRs (V_1, V_2, V_3) that are produced and consumed at cycles as shown below.

EVR	Production time (mod 8)	Consumption time (#cycles after prodn.)
V_1	0	3, 4
V_2	2	3
V_3	4	5, 7, 13

The register lifetimes and usage is shown pictorially in Figure 5 (a) arranged in a modulo fashion. The markings V_1, V_2, V_3 in the figure identify the cycles at which the corresponding EVR has a consuming reference.

Following the above intuitive description of how a ShiftQ may be designed for these EVRs, we can build a time line for the values present in each ShiftQ cell at each cycle as shown in Table 1. A hyphen (-) at a cycle shows that a cell does not contain a live value at that cycle. We also superscript each register value flowing through the cells by the iteration number in which it is produced starting from 0.

We can see that a total of three ShiftQ cells are needed since the value in cell 2 becomes dead on cycle 17 before a new value will be shifted into it on cycle 18. The pattern of live values repeats itself after cycle 16 (17 is like cycle 9 etc.).

The ShiftQ structure corresponding to this example is shown in Figure 5 (b). The first cell shifts whenever the function unit computes a new value, *i.e.*at cycles 0, 2, and 4 (mod 8). The second cell also shifts on all these cycles 0, 2, and 4 (mod 8) because the value contained in the first cell just before these cycles (*i.e.*, on cycles 7, 1, and 3 mod 8) is still live. On the other hand, the third cell needs to shift only on cycle 2 (mod 8) since the value contained in the second cell is live only before that cycle (see cycle 9). Also note that the various consumer references are so timed that no output ports needs to exist for the first cell, while that for second cell is shared among multiple EVRs.

The ShiftQ schema can be further augmented so as to support loop initialization (livein values) as well as loop finalization (liveout values). Loop initialization and loop finalization pose special requirements which do not conform to a steady state modulo schedule. In the next section, we will describe an algorithmic way to arrive at a ShiftQ structure for a given loop schedule taking into account its steady-state, initialization and finalization requirements.

4. SHIFTQ GENERATION

A number of design decisions must be made before a hardware ShiftQ can be defined and optimized. A schedule of operations and an assignment of EVRs to ShiftQs are necessary before the ShiftQ construction procedure can begin. Also, assignments to some EVRs may precede the loop body in order to perform livein initialization. Likewise, references to liveout EVRs may follow the loop body and allow the use of values computed within the loop body to be used outside the loop.

A software pipeline schedule assigns to each operation, a function unit which executes the operation, and a schedule time $\mathbf{tsched}_{op}$ when the operation starts execution relative to the start of an iteration. The scheduler guarantees that when the single iteration schedule is applied repeatedly corresponding to the initiation of a new iteration every II cycles, the schedule is still legal, *i.e.*, all dependences (both inter- and intra-iteration) are satisfied, and no resource is used simultaneously by multiple operations.

The following section describes the process of designing and optimizing a ShiftQ according to the one ShiftQ per function unit output scheme as illustrated in Figure 2. Subsequent sections describe how ShiftQs are accessed, how liveins and liveouts are handled, and how EVRs with multiple producing operations are handled, generalizing the static single assignment restriction imposed in Section 2.

4.1 Steady state

Figure 6 presents the pseudocode to design a steady state ShiftQ for a given function unit output operating under a modulo schedule. The ShiftQ is designed to shift cells only when absolutely necessary, *i.e.*, a cell shifts only if the previous cell shifts and the value within the previous cell is still live. This strategy guarantees that the length of the ShiftQ will be no more than the maximum number of simultaneously live EVRs produced by a functional unit.

A number of assumptions are needed to understand the pseudocode. We assume that virtual registers are numbered and can be identified by an integer index r. The vector $\mathbf{ts}_r$ represents the time when the single assignment EVR r is produced relative to the start of that iteration. Likewise, $\mathbf{te}_r$ represents the end of the lifetime of EVR r (the time of the last use) relative to the start of the iteration. These start and end times represent the time that the data enters the ShiftQ interconnect and the time at which the data is last needed within the interconnect.

The calculation of the time of production must accommodate actual function unit output latency, $\mathbf{to}$, at which a destination operand r is available relative to the start of the operation p that produces r. That is,

$$\mathbf{ts}_r = \mathbf{tsched}_p + \mathbf{to}_p^m$$

where, the operand r is the mth result operand of the operation p. Likewise, a source operand r is consumed at a time that is adjusted by the input sample time $\mathbf{ti}$ relative to the start of the consuming operation c. Therefore, the time of last use is computed as the max of such times over all consumer references cr. That is,

$$\mathbf{te}_r = \max_{cr} \left(\mathbf{tsched}_c + \mathbf{ti}_c^n\right)$$

where the operand r is the nth source operand of the operation c.

Each cell of the ShiftQ is represented in the pseudocode as a set of state variables that identify various properties of that cell at all (modulo) times. The pseudocode computes state variables for the jth cell as a function of the state variables for the $(j-1)^{th}$ cell. The solution progresses inductively starting with the 0th cell as a basis step (Lines 3-8), growing the ShiftQ as necessary (Lines 12-28). The following state variables are computed at each inductive step. The variable $\mathbf{shift_set}_j$ defines the set of modulo times (phase bus cycles) on which the jth cell shifts. For the 0th cell, this is just the set consisting of production time $\mathbf{ts}_r$ mod II for all the EVR r produced by this functional unit (Line 5-6). The variable $\mathbf{live_vr}_{j,s}$ defines the EVR held within the jth cell on cycles equal to s mod II. The variable $\mathbf{flight_time}_{j,s}$ defines the number of cycles that the EVR held within the jth cell at time s has been within the ShiftQ since it was

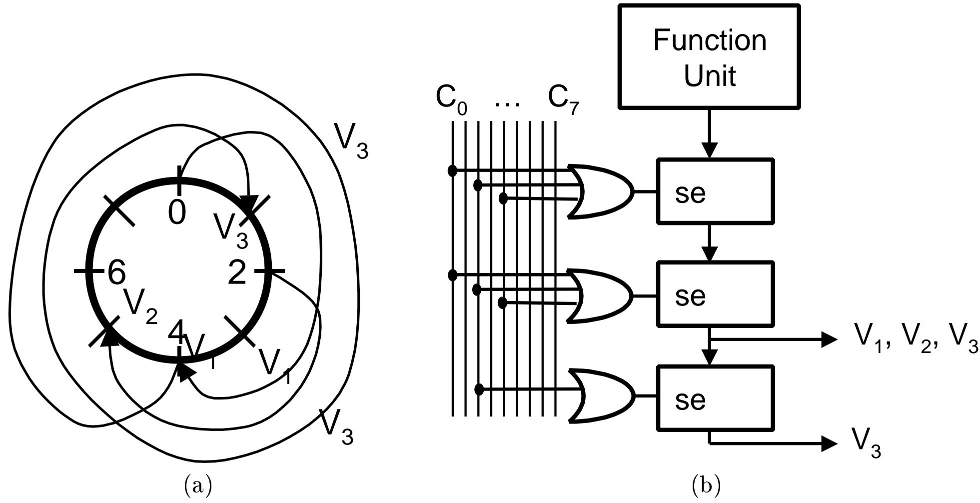

Figure 5: A ShiftQ Example. (a): Overlapping EVR lifetimes in a modulo schedule. (b): A ShiftQ design for these EVRs.

Cell#	EVR contained during cycle																		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	V_1^0	V_1^0	V_2^0	V_2^0	V_3^0	V_3^0	V_3^0	V_3^0	V_1^1	V_1^1	V_2^1	V_2^1	V_3^1	V_3^1	V_3^1	V_3^1	V_1^2	V_1^2	V_2^2
1	–	–	V_1^0	V_1^0	V_2^0	–	–	–	V_3^0	V_3^0	V_1^1	V_1^1	V_2^1	–	–	–	V_3^1	V_3^1	V_1^2
2	–	–	–	–	–	–	–	–	–	–	V_3^0	V_3^0	V_3^0	V_3^0	V_3^0	V_3^0	V_3^0	–	V_3^1

Table 1: A time line of values present in each ShiftQ cell.

```
Procedure Compute_ShiftQ_Steady_State(FUoutput)
01: // Basis step for induction identifies shift times for cell 0.
02: // Shift occurs with insertion of each EVR r generated on this function unit's output.
03: shift_set[0] = nil
04: for each EVR r generated at this FUoutput:
05:   s = ts[r] mod II        // calcuate modulo start time for EVR r
06:   shift_set[0] += s       // add s to shift_set for cell 0
07:   live_vr[0,s] = r        // cell 0 at time s holds EVR r
08:   flight_time[0,s] = 0    // cell 0 at time s holds EVR which just entered ShiftQ
09:
10: // Given all state variables for cell[j-1], calculate state variables for cell[j].
11: // Shift occurs into cell[j] if value in cell[j-1] is live and cell[j-1] shifts.
12: j=1                               // begin with second cell
13: while shift_set[j-1] not empty do // continue growing ShiftQ until no live EVR is left
14:   shift_set[j] = nil
15:   for each s in shift_set[j-1]    // cell[j-1] shifts at s. Does cell[j] shift at s?
16: sp = prev(s, shift_set[j-1])  // previous to s in shift_set[j-1]
17: r = live_vr[j-1,sp]           // cell[j-1] held EVR r before s
18: // calculate relative time at which data would enter cell[j]
19: flight_delta = (s==sp) ? II : (s-sp) mod II  // range is 1 .. II
20: flight_t = flight_time[j-1,sp] + flight_delta
21: if (flight_t < (te[r] - ts[r])) // Is EVR r still live?
22:   live_vr[j,s] = r              // record EVR r as still live
23:   flight_time[j,s] = flight_t   // record its life completed at cell[j] cycle s
24:   shift_set[j] += s             // shift into cell[j] at time s mod II
25: endif
26:   endfor
27:   j=j+1
28: endwhile
endprocedure
```

Figure 6: Pseudocode to determine steady-state ShiftQ structure.

```
Procedure Find_ShiftQ_Cell(r, t)
01: // Find the cell position j where EVR r will be
02: // available t cycles after its production
03: j = 0
04: loop
05:   for each cycle s in 0..(II-1)
06: if (live_vr[j,s]==r && flight_time[j,s]==t)
07:   return j
08:   endfor
09:   j = j+1
10: forever
endprocedure
```

Figure 7: Pseudocode for accessing ShiftQs.

first computed. Note that this pseudocode only computes the state variables at sparse moments in time corresponding to actual shifts taking place. Interpolation to evaluate each state variables at intervening moments in time when a cell does not shift is straightforward and not shown.

The inductive ShiftQ computation proceeds as follows. For each cycle s at which the $(j-1)^{th}$ cell shifts, we need to identify whether the previously contained value is still live or not, and therefore would need to shift into cell j. The previously contained value of cell $j-1$ is just the one that was shifted at the one previous shift sp (Line 16). Note that this could be the same as s due to the modulo nature of the schedule. The positive time difference between the previous shift sp and the current shift s together with the flight time already spent in cell $j-1$ gives the total flight time of this EVR. This flight time is compared against the EVR's actual lifetime (Line 21) and the shift s is recorded if the variable is still live.

Given the number of ShiftQ cells as computed above and the $\mathbf{shift_set}_j$ of each cell to control the shifting, a hardware ShiftQ structure as shown in Figure 4 can easily be built.

4.2 ShiftQ access

Each EVR reference in the loop is characterized by its EVR index r and a relative **iteration distance** d which identifies the value of EVR r produced d iterations ago. The ShiftQ cell position at which this value (r, d) is stored is not fixed, but it may shift down as newly produced values displace old ones. Therefore, we need to compute the cell position holding a value (r, d) within the ShiftQ at any given time. Since each consumer reference accesses the value at a fixed time $\mathbf{tc}$ relative to the start of the iteration[2], this cell position is fixed for each consumer operation. We can compute this position easily using the state variables of the ShiftQ as shown in Figure 7. For a consumer reference at time $\mathbf{tc}$ referencing the EVR value (r, d), this EVR must have been produced at time given by,

$$\mathbf{tp} = \mathbf{ts}_r - d * II$$

The flight time of the value (r, d) since its production is then simply $\mathbf{tc} - \mathbf{tp}$ which is used to search for the appropriate cell position. The pseudocode shown in Figure 7 attempts to search for the EVR r in each modulo cycle s of every ShiftQ cell starting from the zeroth. The first cell in which the flight time reaches the desired value is returned.

[2] As stated earlier, this time of access includes the input sample latency, $\mathbf{ti}_r$.

4.3 Livein / Liveout values

In general, the software pipelined loop may have livein and liveout values that must be initialized before entering the loop body, and must be referenced after the loop terminates, respectively. In the modulo schedule, the livein values appear to have been produced by operations executing in iterations before the first one. Likewise, the liveout values appear to be consumed by operations executing in iterations after the last one. We term the corresponding producer/consumer operation as a *virtual* producer/consumer. We need to identify positions within the ShiftQ where such values will be stored and create a mechanism to seamlessly integrate them with the steady state schedule when the loop starts or terminates. Below we describe the livein handling mechanism and architecture in detail. The liveout mechanism is symmetric and hence not shown for brevity.

The livein initialization scheme is computed using the pseudocode shown in Figure 8. First, the overall livein references of the loop body are classified according to the ShiftQ that houses the corresponding EVR. Each livein EVR reference (r, d) identifies the iteration distance d, which is the number of iterations ago it appears to have been produced. Then, assuming that the loop starts at absolute time T_s, the absolute virtual production time T_{vp} of this EVR reference is computed as $d * II$ cycles earlier than its absolute production time for the first iteration (Line 4). If this time is earlier than the start of the loop T_s (Line 5), then the value appears to have been already produced and reside in one of the steady state ShiftQ registers. This initializing position in the ShiftQ at the start of the loop is identified by computing how far the value would have shifted down during the time elapsed between its (virtual) production and the start of the loop (Line 7).

If the virtual production time of a livein is later than the start of the loop, as in the case when the producer operation is scheduled very late in the loop schedule, we can not initialize the livein value into the steady state registers since they are being used to store other EVRs during this time. Therefore, all such liveins are collected separately and are sorted according to their entry time into the ShiftQ. These values are assigned separate storage called a **PrefixQ** where they may be initialized and flow into the ShiftQ at the appropriate time.

The hardware scheme integrating ShiftQ with the PrefixQ is shown in Figure 9. The important point to note is that now a multiplexor is introduced in the path of the function unit to the ShiftQ, which is flipped occasionally in the direction of the PrefixQ rather than the function unit *via* the prefix shift enable (PSE) control signal. The PSE also causes the entire PrefixQ to shift down by one element. The PSE is "programmed" to allow such shifting at precisely the virtual production times $\mathbf{tv}$ of the various liveins (Figure 8, Line 19). This correctly emulates the fact that a value was produced by the FU at that time.

4.4 Multiple Producer ShiftQs

Until now, we have been considering only singly assigned EVRs having only one producer operation (and hence function unit). Now, let us consider the case when multiple operations may conditionally write to the same EVR. In this case, the lifetime of an operand r begins with the earliest

```
Procedure Compute_ShiftQ_Livein(ShiftQ)
01: // Assume: The modulo schedule starts at absolute time Ts.
02: Prefix_list = nil
03: For each livein reference (r,d) for this ShiftQ:
04:   Tvp = Ts + ts[r] - d*II  // absolute time of virtual producer
05:   if (Tvp < TS)            // initialize in actual ShiftQ
06: flight_time = TS - Tvp // time elapsed since virtual prodn.
07: c = Find_ShiftQ_Cell(r, flight_time)
08: Initialize (r,d) in ShiftQ cell[c]
09:   else                     // initialize in separate PrefixQ
10: prefix_list += (Tvp, (r,d))
11:   endif
12: endfor
13: // Construct PrefixQ
14: sort Prefix_list by ascending key Tvp
15: j = -1
16: For each (Tvp, (r,d)) in Prefix_list // earliest livein first
17:   Allocate PrefixQ cell[j]
18:   Initialize (r,d) in cell[j]
19:   Shift PrefixQ by one at time Tvp   // emulate production
20:   j = j-1
21: endfor
endprocedure
```

Figure 8: Psuedocode for ShiftQ livein initialization.

scheduled producer reference pr. That is,

$$\mathbf{ts}_r = \min_{pr} \ (\mathbf{tsched}_p + \mathbf{to}_p^m)$$

Furthermore, the EVR is uniquely housed in the ShiftQ corresponding to the function unit on which the earliest producer is scheduled. If there are multiple operations producing the EVR at the same time (under mutually exclusive predicates), then we can choose any one of the corresponding ShiftQs to house the EVR.

The ShiftQ and PrefixQ generation algorithm described above remains exactly the same. However, the hardware schema shown in Figure 9 gets modified by allowing remote function units producing the same EVR to overwrite the current value of the EVR housed in this ShiftQ. This is easily accomplished by adding a multiplexor in front of the ShiftQ cell that needs to be overwritten much in the same way as the PrefixQ is multiplexed at the start of the ShiftQ. The cell position to be overwritten is determined in the same way the position of a consumer access is determined as described in Section 4.2.

5. EXPERIMENTAL EVALUATION

In this section, we will evaluate and compare the cost of the various register organization schema presented in Section 2.2 across several embedded applications. First, we present an intuitive description of some cost reducing heuristics used in generating ShiftQs.

5.1 ShiftQ Optimization Heuristics

One drawback of the one ShiftQ per function unit output scheme (SHQ-FU) as presented in Section 4 is that the sharing of register values is limited to those produced by a single function unit. Therefore, the total number of registers needed in a computation may in general exceed those allocated in a unified rotating register file configuration (URR). However, this fact becomes an advantage when we allow customization of individual ShiftQs based on the datawidths of EVR values contained in them. The width of a ShiftQ needs to be only as wide as the maximum width EVR produced by the function unit feeding it. Therefore the total number of bits stored in all the ShiftQs (as a measure of the hardware cost) may still be less than the unified register file configuration which must be as wide as the maximally wide EVR over the whole loop schedule.

While the scheme where only one EVR is allowed per ShiftQ (SHQ-EVR) does not share registers and hence may have a higher register count, it allows additional customization of datawidth to just one EVR. Some of this benefit may be applied even to the SHQ-FU scheme by using the following width reducing technique. If values with longer lifetimes are narrower than those that are short-lived, we can "taper" the ShiftQs in length since all values enter one end of the ShiftQ (the wide end) and then die at different times moving towards the other end (the narrow end). We use this tapering technique by default in all our experiments.

Sometimes, when EVRs with widely different widths are produced on the same function unit output and they have similar and overlapping lifetimes, then the SHQ-FU scheme may end up wasting bits as compared to the SHQ-EVR scheme due to unwarranted register sharing. In this case, a simple cost reducing heuristic is to partition the EVRs being housed in a ShiftQ into classes of similar widths (within a threshold percentage) and form multiple separate ShiftQs, one for each class. We call this the partitioned ShiftQ (PSHQ) scheme. Note that, although we no longer have one ShiftQ per function unit output, the ShiftQ generation algorithms are still valid, albeit on a reduced set of EVRs. In our experiments, this heuristic is applied with a width threshold of 20%, *i.e.*, a new ShiftQ class is generated if the width of an EVR differs from an existing class by more than 20%.

Finally, another space saving heuristic used traditionally in unified rotating register files is to allocate EVRs in a modulo wrapped manner (closed-loop model) [4] thereby saving

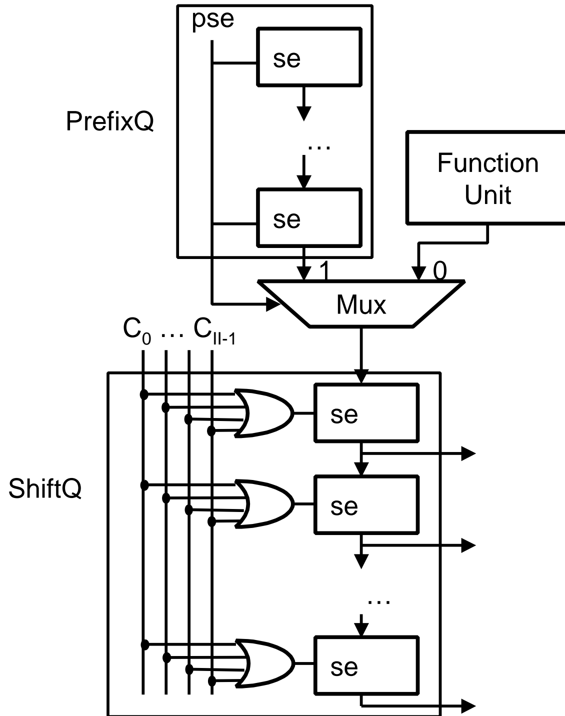

Figure 9: The ShiftQ-PrefixQ schema.

some space as opposed to allocating them in a sliding window (open-loop model). This takes advantage of the fact that all registers in the unified file effectively shift at the same time. In the SHQ-FU scheme, however, more than one EVRs may exist that cause the various cells to shift at potentially unequal intervals. Therefore, the close-loop model does not work for the SHQ-FU scheme and it may lose some opportunity to save space. However, when only one EVR is present in a ShiftQ, its shifting pattern is the same for every cell and therefore the closed-loop allocation model may be used. In our experiments, we assume that this optimization has been made for SHQ-EVR and PSHQ schema (in every partition that has only one EVR).

5.2 Experimental Setup

Table 2 shows several embedded applications from various domains for which loop accelerators with ShiftQs have been designed using the PICO-NPA system. These include telecom applications (atmcell, viterbi), signal processing (fir), image filtering (linescreen, taubman, sobel), and data compression (dct, huffman). The table shows the lines of C code and the depth of the kernel loop nest that is converted into a hardware accelerator.

The multi-dimensional computation intensive kernels of these applications are first automatically transformed into a single-dimension loop that can be modulo scheduled at a specified throughput given by II. The resulting code is synthesized into hardware for each of the interconnect schemes described above. For cell, dct, and fir, we have synthesized several versions with different II in order to show the effect of increasing II which effectively decreases the achieved throughput of the accelerator. For others, we have fixed II to a typical value.

5.3 Results

The loop accelerators designed by PICO-NPA are non-programmable, highly specialized hardware structures designed to exploit very high levels of instruction-level parallelism (ILP) in the applications. The range of throughput achieved in the applications reported here was from 1 op/cycle for fir at II=8, to 128 ops/cycle for atmcell at II=1. In general, the throughput increases linearly as the II is decreased. Beyond II=1, we can scale the performance of the accelerators by allocating more than one processor. For simplicity, all the designs considered in this study are single processor designs.

Another important point to note is that the cost of the loop accelerators designed by PICO-NPA is typically a fraction of the cost of a general purpose processor (or a DSP processor) because of the highly customized datapath designed specifically for that one application as shown in this paper. The range of estimated gate cost of the accelerators designed in this study was from 6122 gates for fir at II=8, to 68122 gates for dct at II=1. For the same application, the cost typically decreases (sub-linearly) as the II is increased. Such low cost/performance ratios are not achievable in a general purpose processor which must implement a programmable solution.

We compare loop accelerators designed by PICO-NPA using each of the register interconnect schemes presented in this paper for each of the applications at a given II. Table 3 shows the number of registers allocated and the total number of bits allocated in each of the schemes for each application. The number of registers is useful as an estimate of the degree of sharing among the various EVRs. Maximal degree of sharing is possible in the case of a unified rotating register file (URR) while no sharing is possible in the case of the SHQ-EVR scheme[3]. However the total number of bits allocated is still quite high in the URR scheme since all registers have the same width (32-bits) and therefore a lot of bits are wasted for narrow EVRs. This effect is clear in Figure 10 where we have plotted the total number of bits used in each of the schema. The biggest penalty for the URR scheme is for atmcell which contains a lot of 1-bit data. On the other hand, for some applications (lyapunov and huffman) URR yields the smallest number of bits over all schema because the application is dominated by 32-bit data and hence allows maximal sharing of registers without wasting too many bits. Still, in terms of actual silicon area, we expect the URR scheme to be much more expensive than the other schemes due to a highly ported centralized organization.

The two ShiftQ schemes take an intermediate position on register sharing while enabling width optimizations. The SHQ-FU scheme uses less bits than either URR or SHQ-EVR scheme except in two cases. First, when $II = 1$ SHQ-EVR scheme wins out because there is no sharing anyway and a closed-loop model yields the best allocation. Second, for certain applications such as taubman, an accidental sharing between a wide and a narrow EVR wastes too many bits. The PSHQ scheme is designed to capture both these scenarios. In general, it may yield a little worse bit cost than the SHQ-FU scheme due to potential loss of sharing but it is

[3]Sometimes the number of registers in URR scheme are higher than those in SHQ-EVR scheme becuase an open-loop allocation model was assumed for URR while a closed-loop model was assumed for SHQ-EVR.

app.II	kernel size	Description
fir.1,2,4,8	1(2D)	16-tap Finite impulse response filter (convolution)
dct.1,2,4,8	42(2D)	8-element Discrete Cosine transformation on 8x8 blocks
atmcell.1,2,4,8	135(1D)	ATM cell recognition in a stream of bits
huffman.8	16(2D)	Huffman coding of blocks of 64 elements
linescreen.6	59(2D)	Halftoning algorithm on an image stripe
lyapunov.6	12(3D)	Matrix equation solver
sobel.8	18(2D)	Edge detection algorithm
taubman.8	4(6D)	Image demosaicing algorithm
viterbi.6	63(2D)	Error-correcting decoding of a signal

Table 2: Application kernels for evaluation.

app.II	URR		SHQ-EVR		SHQ-FU		PSHQ	
	#regs	#bits	#regs	#bits	#regs	#bits	#regs	#bits
fir.1	34	685	26	378	28	416	26	378
fir.2	16	295	19	298	17	258	18	266
fir.4	10	196	13	215	11	175	12	183
fir.8	8	163	11	182	9	142	10	150
dct.1	139	3115	139	1797	146	1965	139	1797
dct.2	78	1814	112	1673	88	1254	89	1252
dct.4	50	1197	104	1648	65	908	66	909
dct.8	36	904	98	1627	47	685	48	679
atmcell.1	293	5563	300	550	304	616	300	550
atmcell.2	262	4850	303	553	295	595	283	533
atmcell.4	243	4490	306	576	257	532	267	537
atmcell.8	242	4365	310	596	259	508	264	476
huffman.8	19	453	34	594	24	483	28	464
linescreen.6	48	916	78	499	68	460	69	468
lyapunov.6	72	1374	117	1995	82	1450	86	1474
sobel.8	122	2912	170	1045	166	726	168	746
taub.8	95	2265	84	677	116	1553	79	610
viterbi.6	89	1763	164	1104	123	943	131	982

Table 3: Results of using various register interconnect schemes.

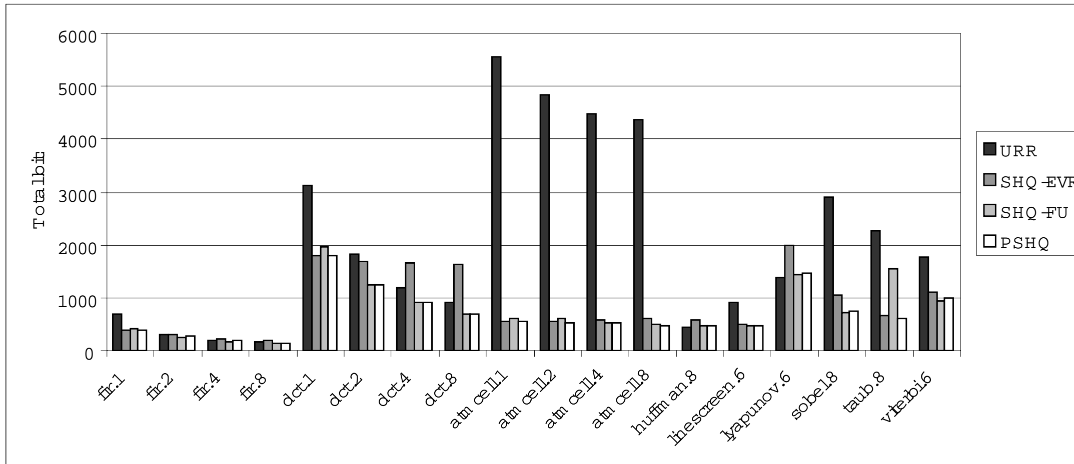

Figure 10: Total number of bits used in various register interconnect schema.

more robust for all values of II and for applications with varying data widths. In a real world situation, it is also possible to design and evaluate all three schema and pick the best.

6. CONCLUSIONS

In this paper we have presented a novel, low-cost, buffer and interconnect mechanism called ShiftQs to be used in custom hardware accelerators executing modulo scheduled loops. The scheme results in smaller total number of bits needed than either a unified register file design or a design in which each virtual register is allocated in a separate hardware structure. A more detailed cost analysis (*e.g.*, including multiplexing and interconnect) is the subject of ongoing investigation.

We have shown an algorithmic way to derive the ShiftQ design and its initialization logic automatically from the loop schedule. The scheme has been implemented within the PICO-NPA high-level synthesis tool and has already produced successful RT-level designs for several embedded application kernels.

7. REFERENCES

[1] S. Aditya, B. R. Rau, and V. Kathail. Automatic Architecture Synthesis of VLIW and EPIC Processors. In *Proceedings of the 12th International Symposium on System Synthesis, San Jose, California*, pages 107–113, November 1999.

[2] S. Devadas and R. Newton. Algorithms for hardware allocation in data path synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7), July 1989.

[3] J. Hoogerbrugge and H. Corporaal. Register file port requirements of transport triggered architectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 191–195, December 1994.

[4] V. Kathail, M. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Feb. 1994.

[5] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[6] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11), November 2001.

[7] S. Note, W. Guerts, F. Catthoor, and H. D. Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 597–601, June 1991.

[8] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6), June 1989.

[9] B. R. Rau. Data flow and dependence analysis for instruction level parallelism. In U. B. et al., editor, *Proc. Fourth Intl. Wshop. on Languages and Compilers for Parallel Computing*, pages 236–250. Springer-Verlag, 1992.

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

[11] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, G. Snider, S. Anik, and S. Abraham. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *Proc. Application-Specific Architectures and Processors*, 2000.