Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures

MICHAEL PELLAUER, Intel[1]
ANGSHUMAN PARASHAR, Intel[1]
MICHAEL ADLER, Intel
BUSHRA AHSAN, Intel
RANDY ALLMON, Intel
NEAL CRAGO, Intel[1]
KERMIN FLEMING, Intel
MOHIT GAMBHIR, Intel
AAMER JALEEL, Intel[1]
TUSHAR KRISHNA, Intel[2]
DANIEL LUSTIG, Princeton University
STEPHEN MARESH, Intel
VLADIMIR PAVLOV, Intel
RACHID RAYESS, Intel
ANTONIA ZHAI, Univseity of Minnesota
JOEL EMER, Intel[1] and MIT

Recently there has been interest in exploring the acceleration of non-vectorizable workloads with *spatially-programmed* architectures that are designed to efficiently exploit pipeline parallelism. Such an architecture faces two main problems: A) how to efficiently control each *processing element* (PE) in the system, and B) how to facilitate inter-PE communication without the overheads of traditional shared-memory coherent memory. In this paper, we explore solving these problems using *triggered instructions*, and *latency-insensitive channels*. Triggered instructions completely eliminate the program counter and allow programs to transition concisely between states without explicit branch instructions. Latency-insensitive channels allow efficient communication of inter-PE control information, while simultaneously enabling flexible code placement and improving tolerance for variable events such as cache accesses. Together, these approaches provide a unified mechanism to avoid *over-serialized* execution, essentially achieving the effect of techniques such as dynamic instruction reordering and multithreading.

Our analysis shows that a spatial accelerator using triggered instructions and latency-insensitive channels can achieve 8× greater area-normalized performance than a traditional general-purpose processor. Further analysis shows that triggered control reduces the number of static and dynamic instructions in the critical paths by 62% and 64% respectively over a program-counter style baseline, increasing the performance of the spatial programming approach by 2.0×.

Categories and Subject Descriptors: C.1.3 [**Computer Systems Organization**]: Processor Architectures

[1]New affiliation: NVIDIA.
[2]New affiliation: Georgia Insititue of Technology

Author's addresses: Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, Stephen Maresh, Vladimir Pavlov, Rachid Rayess, and Joel Emer, Intel Corporation, 77 Reed Road, Hudson, MA 01749.

DOI: http://dx.doi.org/10.1145/0000000.0000000

Additional Key Words and Phrases: Spatial Programming, Reconfigurable Accelerators

ACM Reference Format:
Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, Stephen Maresh, Vladimir Pavlov, Rachid Rayess, and Joel Emer. Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures. *ACM Trans. Comput. Syst.* X, X, Article X (February 2015), 32 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Recently, SIMD/SIMT accelerators such as GPGPUs have been shown to be effective as offload engines when paired with general-purpose CPUs. This results in a complementary approach where the CPU is responsible for running the operating system and irregular programs, and the accelerator executes inner loops of uniform data-parallel code. The abundant number of threads allows the accelerator to bury long-latency events such as cache misses and maintain good utilization of its datapaths.

Unfortunately, not every workload exhibits sufficiently uniform data parallelism to take advantage of the efficiencies of this pairing. There remain many important workloads whose best-known implementation involves asynchronous actors performing different tasks, while frequently communicating with neighboring actors. The computation and communication characteristics of these workloads cause them to map efficiently onto *spatially-programmed* architectures such as Field-Programmable Gate Arrays (FPGAs). Furthermore, a number of important workload domains exhibit such kernels, such as signal processing, media codecs, cryptography, compression, pattern matching and sorting. As such, one way to boost the performance efficiency of these workloads is to add a new spatially-programmed accelerator to the system, complementing the existing SIMD/SIMT accelerators.

While FPGAs are very general in their ability to map the compute, control and communication structure of a workload, their *lookup table* (LUT) based datapaths are oriented towards arbitrary logic prototyping rather than algorithmic acceleration. An alternative is to use a tiled array of coarse-grained datapaths more like a processor's Arithmetic-Logic Unit (ALU) — a Coarse Grained Reconfigurable Array (CGRA) [Mirsky and DeHon 1996; Hauser and Wawrzynek 1997; Mei et al. 2003]. However, CGRAs come with several challenges as well. How should each individual ALU be controlled? How should the ALUs communicate data with each other, especially given that communication is frequent? If a producer is not *mapped* onto a datapath directly adjacent to a consumer will the program fail?

A number of prior works [Burger et al. 2004; Govindaraju et al. 2011; Swanson et al. 2007] have proposed spatial architectures with a network of ALU-based *processing elements* (PEs) onto which operations are scheduled in systolic or dataflow order, with limited or no autonomous control at the PE level. Other approaches incorporate autonomous control at each PE using a *program counter* (PC) [Taylor et al. 2002; Yu et al. 2006; Panesar et al. 2006]. Unfortunately, as we will show, PC sequencing of ALU operations introduces several inefficiencies when attempting to capture intra- and inter-ALU control patterns of a frequently-communicating spatially-programmed fabric.

In this paper, we explore addressing these issues using *triggered instructions* and *latency-insensitive channels*. Triggered instructions remove the program counter completely, instead allowing the processing element to concisely transition between states of one or more *finite-state machines* (FSMs) without executing instructions in the datapath to determine the next state. Latency-insensitive channels allow efficient communication of inter-PE control information, while simultaneously enabling flexible module placement and improving tolerance for unpredictable events such as cache ac-

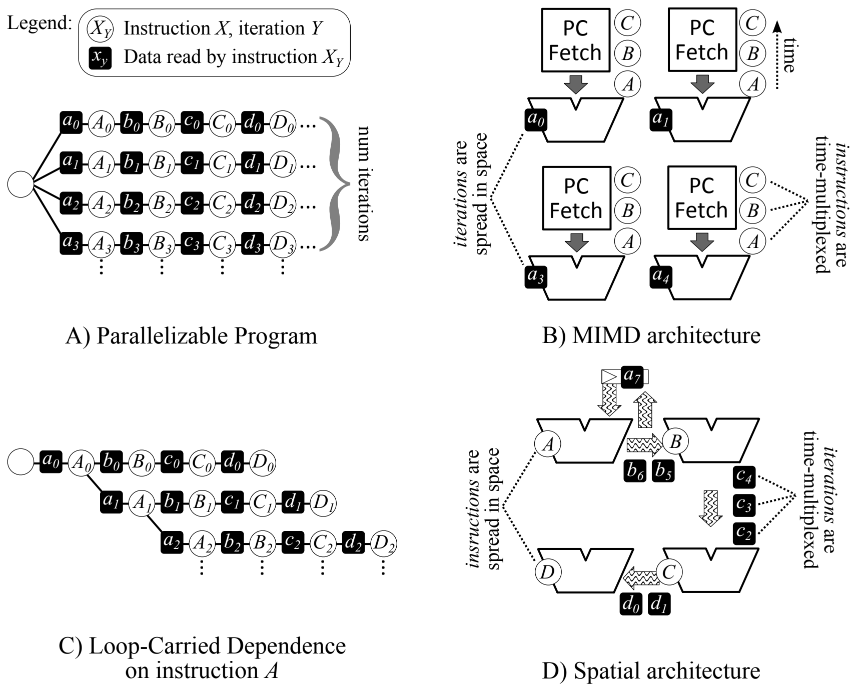

Legend:
Instruction X, iteration Y
Data read by instruction X_Y
num iterations
A) Parallelizable Program
PC Fetch
time
iterations are spread in space
instructions are time-multiplexed
B) MIMD architecture
C) Loop-Carried Dependence on instruction A
insructions are spread in space
iterations are time-multiplexed
D) Spatial architecture

Fig. 1. Overview of the spatial programming approach.

cesses. Together, these approaches provide a unified mechanism to avoid *over-serialized* execution, essentially achieving the effect of techniques such as dynamic instruction reordering and multithreading, which each require distinct hardware mechanisms in a traditional sequential architecture.

We evaluate out approach by simulating a spatially-programmed accelerator on a range of workloads. Our analysis for this set of workloads, which span a range of algorithm classes not known to exhibit extensive uniform data parallelism, shows that such an accelerator can achieve 8× greater area-normalized performance than a traditional general-purpose processor. We provide further analysis of both a set of common control idioms and the critical paths of the workload programs to illustrate how a triggered instruction architecture contributes to this performance gain.

2. BACKGROUND AND MOTIVATION

2.1. Spatially-Programmed Architectures

Spatial programming is a paradigm whereby an algorithm's dataflow graph is broken into regions, which are connected by producer-consumer relationships. Input data is then streamed through this pipelined graph. Figure 1 contrasts this to a more traditional Multiple-Instruction-Multiple-Data (MIMD) approach. In MIMD, core n is responsible for executing a full loop iteration containing each instruction $A, B, C, ...$ of the sequential body, keeping the intermediate data b_n, c_n, etc. locally in its register file. Cross-core communication is rare and is ideally protected with syncrhonization primitives such as barriers.

In contrast, in a spatial approach a single small core is responsible for executing instruction A for *all* iterations, with another core executing instruction B. Intermediate data is passed to the next datapath rather than being kept locally. Therefore no single

PE will execute an entire loop body—essentially the program has been transposed between time and space. Pedagocially, it is clearest to imagine the extreme approach of mapping one instruction to each PE, but in practice there can be benefits to keeping a small control sequence or Finite-State Machine (FSM) local. Ideally, the number of operations in each pipeline stage is kept small, as performance is usually determined by the *rate-limiting step*.

For an "embarassingly" parallel program like 1A, it may seem like a SIMD/SIMT architecture will be a strictly more efficient execution substrate, and so the spatial transposition is not needed. In practice there exists a large category of interesting programs that contain *loop-carried depedencies*. As Figure 1C shows, these require cross-iteration data passing, and skew the program's dataflow graph in such a way as to prevent vectorization. However, these programs are naturally implementable in the spatial approach simply by using local registers to bypass the cross-iteration data to the next instruction. In fact, the spatial program shown in Figure 1D can implement programs 1A and 1C with no change in code mapping and no degredation of performance—the only difference is whether the output of instruction A is recirculated via a local register or not.

Just as data-parallel algorithms see large efficiency boosts when run on a vector engine, workloads that are naturally amenable to spatial programming can see significant boosts when run on an enabling architecture. A traditional processor would execute such programs serially over *time*, but this does not result in any noticeable efficiency gain, and may even be slower than other expressions of the algorithm. A MIMD multicore can improve this by mapping different stages onto different cores, but the small number of cores available relative to the large number of stages in the dataflow graph means that each core must multiplex between several stages, so the rate-limiting step generally remains large. Additionally, cross-thread communication can generally only occur via *shared virtual memory* (SVM) which uses a *coherence protocol* to actually interact with the on-chip network (OCN) and perform the data transfers.

In contrast, a typical spatial-programming architecture is a fabric of hundreds of small processing elements (PE) connected directly via an OCN which is exposed directly to the ISA. Given enough PEs, an algorithm may be taken to the extreme of mapping the entire dataflow graph into the spatial fabric, resulting in a very fine-grained pipeline. This is the approach taken by a number of *reconfigurable* architectures.

FPGAs are the most successful spatially-programmed reconfigurable architecture in use today. FPGAs are designed to emulate a broad range of logic circuits because they are primarily targeted at ASIC prototyping and replacement. Consequently, they use very fine-grain reconfigurable elements such as *lookup tables* (LUTs) [Compton and Hauck 2002; Marquardt et al. 2000]. The LUTs are chained into larger operations using flexible-but-expensive OCNs. This generality limits the clock speed at which mapped designs can be run while also creating a large search space of solutions for place and route algorithms, leading to long compilation times.

When using reconfigurable architectures for direct algorithmic acceleration instead of logic prototyping, these issues can be partially addressed by the observation that the class of operations that the reconfigurable architecture needs to cover is more limited—particularly when used in conjunction with a traditional CPU. As observed by several efforts [Mirsky and DeHon 1996; Hauser and Wawrzynek 1997; Mei et al. 2003], this limited class of operations creates opportunities to achieve higher area density and better power/performance efficiency than conventional FPGAs while retaining sufficient flexibility. This has led to several proposals [Burger et al. 2004], [Panesar et al. 2006; Taylor et al. 2002; Swanson et al. 2007; Mirsky and DeHon 1996; Hauser and

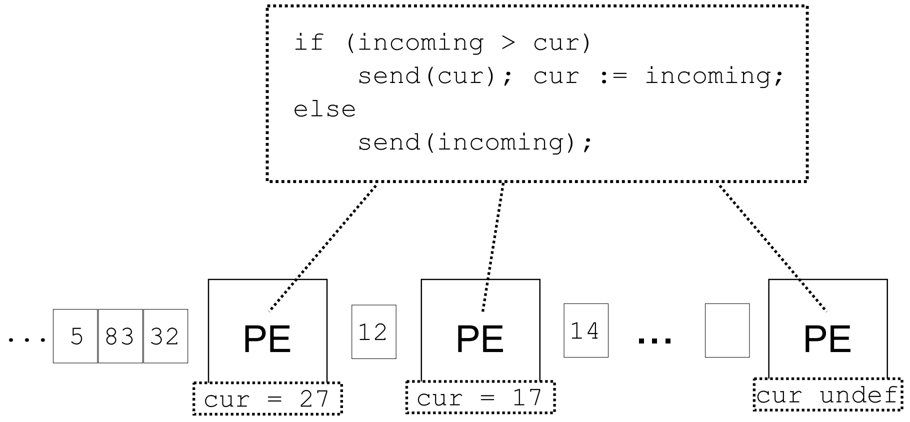


Fig. 2. Example of a spatially-programmed sort.

Wawrzynek 1997; Mei et al. 2003] that use an array of coarser-grained multi-bit ALUs as the datapath of PEs in a spatially-programmed architecture.

Within the domain of array-of-ALU approaches is a class of architectures that do not feature any autonomous control mechanism inside each ALU. These architectures are either purely systolic [Kung 1986], statically map only one operation per ALU [Govindaraju et al. 2011], or schedule operations onto the ALUs in strict dataflow order [Burger et al. 2004]. These architectures rely on being able to transform control-flow graphs into predicated dataflow graphs. Such approaches are effective at mapping the control structures of a subset of problems, but do not approach the flexibility or generality of architectures with internal autonomous control at each PE. Another class of proposals calls for general autonomously-controlled PEs [Taylor et al. 2002; Yu et al. 2006; Panesar et al. 2006] using variants of the existing PC-based control model.

The PC-based control model paired with SVM has historically been the best choice for MIMD CPUs that run arbitrary and irregular programs. In the remainder of this section, we demonstrate that these existing paradigms introduce unacceptable inefficiencies in the context of spatial programming.

2.2. Spatial Programming Example

As a concrete example, let us explore how a well-known workload can benefit from spatial programming. Consider the simple spatially-mapped sorting program shown in Figure 2. In this approach, the worker PEs communicate in a straight pipeline. The unsorted array is streamed in by the first PE. Each PE simply compares the incoming element to the largest element seen so far. The larger of the two values is kept, and the smaller sent on. Thus after processing k elements worker 0 will be holding the largest element, and worker $k-1$ the smallest. The sorted result can then be streamed out to memory through the same straightline communication network.

This example represents a limited toy workload in many ways—it requires k PEs to sort an array of size k, and worker 0 will do $k-1$ comparisons while worker $k-1$ will only do 1 (an insertion sort, with a total of k^2 comparisons). However, despite its naivete this workload demonstrates some remarkable properties. First, the utilization of the datapaths is quite good—in the final step all k datapaths can simultaneously execute a comparison, with an overall average of $\frac{k}{2}$ per cycle. Second, the communication between PEs is local and parallel—on a typical mesh network fabric it is easy to map this workload so that no network contention will ever occur. The communication

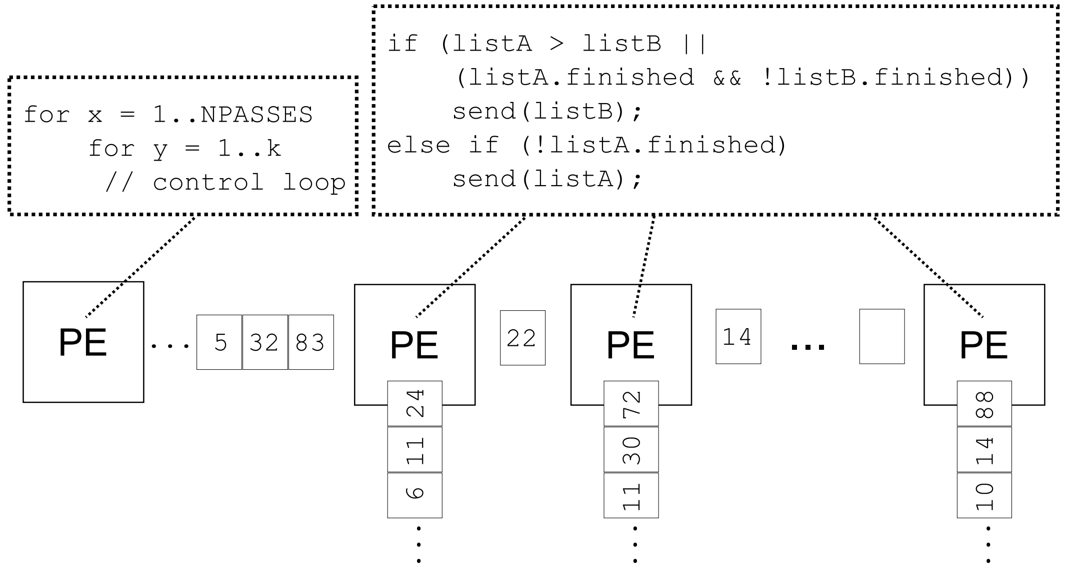


Fig. 3. A more realistic spatial merge sort.

flows are completely statically determined by the configuration—no dynamic packet routing is required. Finally—and most interestingly—this approach sorts an array of size k with exactly k loads and k stores. The loads and stores that a traditional CPU must use to overcome its relatively small register file are replaced by direct PE-to-PE communication. This reduction in memory operations is critical in understanding the benefits of spatial programming. We have been able to characterize the benefits as follows:

— Direct communication uses roughly $20\times$ lower power than communication through an L1 cache, as the overheads of tag matching, load-store queue search, address translation, and large data array read are removed.
— Cache coherence overheads, including network traffic and latency are likewise removed.
— Reduced memory traffic lowers cache pressure, which in turn increases effective memory bandwidth for remaining traffic.

Finally, it is straightforward to expand our toy example into a realistic merge sort engine able to sort a list of any size (Figure 3). First, we begin by programming a PE into a small control FSM that handles breaking the array into sub-arrays of size k and looping (this control loop could also be executed on the main CPU, as the sorting passes can be quite long). Second, we slightly change the worker PEs' programming so that they are doing a merge of two distinct sorted sub-lists. With these changes our toy workload is now a radix k merge sort capable of sorting a list of size n in $n * log_k(n)$ loads. Because k can be in the hundreds for a reconfigurable fabric, the benefits can be quite large. In our experiments we observed $17\times$ fewer memory operations compared to a general-purpose CPU and an area-normalized performance improvement of $8.8\times$ (Section 6), which is better than the currently best-known GPGPU performance [Merrill and Grimshaw 2010].

```
load_a:  ld      %r4, @r2        // Load listA tail ptr (written by producer)
check_a: cmp.ne  %r0, %r4, %r3  // If listA head == tail then q is empty
         bnez    check_b         // Not empty, so proceed to listB
         monitor @r2             // Wait for producer to update tail
         jump    load_a          // Re-load and check in case timeout occured
load_b:  ld      %r9, @r7        // Load listB tail ptr (written by producer)
check_b: cmp.ne  %r0, %r9, %r8  // If listB head == tail then q is empty
         bnez    check_o         // Not empty, so proceed to output list
         monitor @r7             // Wait for producer to update tail
         jump    load_b          // Re-load and check in case timeout occured
load_o:  ld      %r13, @r11      // Load listOut head ptr (written by consumer)
...
```

Static Insts	48
Avg Insts/Iteration	32
Avg Memory Ops/Iteration	10

Fig. 4. Merge sort worker representation using SVM queues introduces unacceptable overheads per comparison. Methods such as memory monitors can avoid active polling, but do not reduce pointer chasing and load/store latency between disparate caches.

2.3. Limitations of PC-based Control

To illustrate the inefficiencies of existing MIMD paradigms in the spatial programming context, let us code the merge sort PE shown in Figure 3. We first explore whether a completely unmodified ISA is suitable for this task. In a multicore system, the typical approach is to use SVM for the queue buffering, along with sophisticated polling mechanisms such as *memory monitors* for communicating occupancy. As shown in Figure 4, such a style introduces many inefficient instructions for pointer chasing, address offset arithmetic, and head/tail comparisons just to setup the user-specified sort comparison inherent to the algorithm. Even if the queues are not stalled, and the monitors unneccesary, the loop would execute an average of 32 instructions per sort-comparison in the best case, including 7 loads and 3 stores. Furthermore, in a spatially-programmed fabric having hundreds of PEs communicating using shared memory would create unacceptable bandwidth bottlenecks. It would also be wasteful—communicating with your neighbor should not have to go through a centralized location.

Instead, let us modify the ISA to expose direct communication channels between PEs as data registers and status bits. The ISA must contain a mechanism to query if the input channels are not empty, and output channels are not full, to read the first element, and to enqueue and dequeue. Furthermore we add an architecturally-visible tag to the channel that merge sort uses to indicate that the end of a sorted sub-list has been reached (`EOL`). We name the resulting theoretical assembly language PC+RegQueue, and give a representation of the merge sort PE in Figure 5. This code removes all the problematic memory references and pointer manipulation from the original example, but several inefficiencies are still noticeable. First, it uses *active polling* to test the register-mapped queue status, an obvious power waste. Second, it falls victim to *over-serialization*. For example, if new data on `listA` arrives before that on `listB` there is no opportunity to begin processing the `listA`-specific part of the code. Finally, the code is quite branch-heavy when compared to that typically found on a traditional core, and some of these branches are hard to predict.

This illustrates that simply augmenting a traditional RISC-style ISA with a channel-based communication paradigm is not sufficient to enable efficient spatial programming. In order to be fair to this PC-based ISA we must try to improve the architecture somehow. Figure 6 summarizes the techniques that we explore below.

```
check_a: beqz   %in0.notEmpty, check_a // listA
check_b: beqz   %in1.notEmpty, check_b // listB
check_o: beqz   %out0.notFull, check_o // outList
         beq    %in0.tag, EOL, a_done
         beq    %in1.tag, EOL send_a
         cmp.lt %r0, %in0.first, %in1.first
         bnez   %r0, send_a
send_b:  enq    %out0, %in1.first
         deq    %in1
         jump   check_a
send_a:  enq    %out0, %in0.first
         deq    %in0
         jump   check_a
a_done:  beq    %in1.tag, EOL, done
         jump   send_b
done:    deq    %in0
         deq    %in1
         return;
```

Static Insts	18
Avg Insts/Iteration	10
Avg Branches/Iteration	7
Speedup vs shared memory queues (Fig 4)	5.2× vs cache hits (no monitor needed) 14.0× vs misses and monitor case

Fig. 5. PC+RegQueue ISA merge sort worker representation using register-mapped queues.

Feature	Description	Notes
PC (Baseline)	PEs use program counters, communicate using SVM queues.	High latency, bottlenecks.
+RegQueue	Expose register-mapped queues to ISA, test via active polling.	Poor power efficiency.
+FusedDeq	Destructive read of queue registers without separate instructions.	Good improvement.
+RegQSelect	Allow indirect jump based on register queue status bits.	Minimal improvement.
+RegQStall	Issue stalls on queue input/output registers without special instructions.	Bubbles, over-serialization.
+QMultiThread	Stalling on empty/full queue yields thread.	Significant additional hardware.
+Predication	Predicate registers that can be set using queue status bits.	Boolean expressions don't compose.
+Augmented	ISA augmented with all of the above features except +QMultiThread.	Used in our evaluations (Section 6).

Fig. 6. Adding features to a PC-based ISA to improve efficiency for spatial programming.

One idea to improve queue accesses is to allow *destructive reads* of input channels. In such an ISA the SRC fields of the instruction are supplemented with a bit indicating whether a dequeue is desired. This is an important improvement because it reduces both static and dynamic instruction count. Merge sort's implementation on this architecture can remove 3 instructions compared to Figure 5.

```
start:          beq    %in0.tag, EOL, a_done
                beq    %in1.tag, EOL, send_a
                cmp.ge p2, %in0.first, %in1.first
send_b:  (p2)   enq    %out0, %in1.first (deq %in1)
send_a: (!p2)   enq    %out0, %in0.first (deq %in0)
                jump   start
a_done:         cmp.ne p2, %in1.tag, EOL
         (p2)   jump   send_b
                nop    (deq %in0, deq %in1)
                return;
```

Static Insts	9
Avg Insts/Iteration (Issued)	6
Avg Insts/Iteration (Committed)	5
Avg Branches/Iteration	3
Speedup vs PC+RegQueue (Fig 5)	1.4×

Fig. 7. PC+Augmented ISA merge sort worker.

The next idea is to replace the active polling with a *select*—an indirect jump based on queue status bits. This is a marginal improvement in instruction count but does not help power efficiency. A better idea is to add *implicit stalling* to the ISA. In this case the queue registers such as `%in0` would be treated specially—any instruction that attempts to read/write them would require the issue logic to test the empty/full bits and delay issue until the status becomes correct. Merge sort's implementation on this architecture is the same as in Figure 5, but removes the first three instructions entirely.

Of course, the downside of this is that the ALU will not be used when the PE is stalled. Therefore the next logical extension is to consider a limited form of *multi-threading*. In this ISA any read/write of a queue would make the thread eligible to be switched out and replaced with a ready one. This is a promising approach, but we believe that the overheads associated with it—duplication of state resources, additional muxing, and scheduling fairness—run counter to the fundamental spatial-architecture principle of replicating simple PEs. In other words, the cost-to-benefit ratio of multi-threading is unattractive. We reject *out-of-order issue* for similar reasons.

The final ISA extension we consider is *predication*. We define a variant of our ISA that is able to test and set a dedicated set of boolean predicate registers. Figure 7 shows a re-implementation of the merge sort worker in a language with predication, implicit stalling, and destructive reads. It is interesting to note how little predication improves the control flow of the example. This is because of several limitations:

— Instructions are unable to read multiple predicate registers at once (inefficient conjunction).
— Composing multiple predicates into more complex boolean expressions (disjunctions, etc) must be done using the ALU itself.
— Jumping between regions requires that the predicate expectations be set correctly. (Note that the branch from `a_done` is forced to use `p2` with a positive polarity.)
— Predicated false instructions introduce bubbles into the pipeline (Section 5).

Taken together, these inefficiencies mean that conditional branching remains the most efficient way to express the majority of the code in Figure 7. While we could continue to try to add features to PC-based schemes in order to improve efficiency, in the remainder of the paper we demonstrate that taking a different approach altogether

ALGORITHM 1: Traditional Guarded Action Merge Sort Worker

```
rule $sendA$
when $listA.first() \neq EOL$ and $listB.first() \neq EOL$ and $listA.data < listB.data$ do
    $outList.send(listA.first());$
    $listA.deq();$
end rule

rule $sendB$
when $listA.first() \neq EOL$ and $listB.first() \neq EOL$ and $listA.data \geq listB.data$ do
    $outList.send(listB.first());$
    $listB.deq();$
end rule

rule $drainA$
when $listA.first() \neq EOL$ and $listB.first() = EOL$ do
    $outList.send(listA.first());$
    $listA.deq();$
end rule

rule $drainB$
when $listA.first() = EOL$ and $listB.first() \neq EOL$ do
    $outList.send(listB.first());$
    $listB.deq();$
end rule

rule $bothDone$
when $listA.first() = EOL$ and $listB.first() = EOL$ do
    $listA.deq();$
    $listB.deq();$
end rule
```

can efficiently address these issues while simultaneously removing over-serialization and providing the benefits of multi-threading.

3. LOCAL PE CONTROL: TRIGGERED INSTRUCTIONS

A large degree of the inefficiency discussed in the previous section stems from the issue of efficiently composing boolean control flow decisions. In order to overcome this, we draw inspiration from the historical computing paradigm of *guarded actions*, a field that has a rich technical heritage including Dijkstra's language of guarded commands [Dijkstra 1975], Chandy and Misra's Unity [Chandy and Misra 1988], and the Bluespec hardware description language [Bluespec, Inc. 2007].

Computation in a traditional guarded action system is described using *rules* composed of *actions* — state transitions — and *guards* — boolean expressions that describe when a certain action is legal to apply. A *scheduler* is responsible for evaluating the guards of the actions in the system and posting *ready* actions for execution, taking into account both inter-action parallelism and available execution resources. Algorithm 1 illustrates our merge sort worker in traditional guarded action form. Note how this paradigm naturally separates the representation of data transformation (via actions) from the representation of control flow (via guards). Additionally, the inherent side-effect-free nature of the guards means that they are a good candidate for parallel evaluation by a hardware scheduler.

A *triggered instruction architecture* (TIA) applies this concept directly to controlling the scheduling of operations on a PE's datapath at an instruction-level granularity. In the historical guarded action programming paradigm, arbitrary boolean expressions are allowed in the guard, and arbitrary data transformations can be described in the action. To adapt this concept into an implementable ISA, both must be bounded in complexity. Furthermore, the scheduler must have the potential for efficient implementation in hardware. To this end, we define a limited set of operations and state updates that can be performed by the datapath (instructions) and a limited language of boolean expressions (triggers) built from a variety of possible queries on a PE's architectural state.

The *architectural state* of our proposed TIA PE is composed of the following elements:

— A set of **data registers** (R/W).
— A set of **predicate registers** (R/W).
— A set of **input-channel head elements** (R-only).
— A set of **output-channel tail elements** (W-only).

Each channel has three components — *data*, a *tag* and a *status* predicate that reflects whether an input channel is empty or an output channel is full. Tags do not have any special semantic meaning — the programmer can use them in a variety of ways.

A *trigger* is a programmer-specified boolean expression formed from the logical conjunction[3] of a set of queries on the PE's architectural state. Triggers are evaluated by a hardware scheduler (described shortly). The set of allowable trigger query functions are carefully chosen to maintain scheduler efficiency while allowing for a large degree of generality in the useful expressions. These query functions are:

— **Predicate Register Values (optionally negated):** A trigger can specify a requirement for one or more predicate registers to be either true or false, e.g., `p0 && !p1 && p7`.
— **Input/Output Channel Status (implicit):** The scheduler implicitly adds the empty status bits for each operand input channel to the trigger for an instruction. Similarly, a not-full check is implicitly added to each output channel an instruction attempts to write. The programmer does not have to worry about these conditions, but must understand while writing code that the hardware will check them. This facilitates convenient, fine-grained, producer/consumer interaction.
— **Tag Comparisons against Input Channels:** A trigger may specify a value that an input channel's tag must match, e.g., `in0.tag == EOL`.

An *instruction* represents a set of data and predicate computations on operands drawn from the architectural state. Instructions selected by the scheduler are executed on the PE's datapath. An instruction has the following read, compute and write capabilities:

— An instruction may **read** a number of operands, each of which can be data at the head of an input channel, a data register, or the vector of predicate registers.
— An instruction may **perform a data computation** using one of the standard functions provided by the datapath's ALU. It may also **generate one or more predicate values** that are either constants (true/false) or derived from the ALU result via a limited set of datapath-supported functions, e.g., reduction AND, OR and XOR operations, bit extractions, ALU flags such as overflow, etc.
— An instruction may **write** the data result and/or the derived predicate result into a set of destinations within the architectural state of the PE. Data results can be

[3]Although the architecture natively allows only conjunctions in trigger expressions, disjunctions can be emulated by creating a separate triggered instruction for each disjunctive term.

```
// p0 = Have we done a comparison yet?
// p1 = Result of comparison. Is listB.head > listA.head?

doCheck:
    when (!p0 && %in0.tag != EOL && %in1.tag != EOL) do
        cmp.ge p1, %in0.data, %in1.data (p0 := 1)

sendA:
    when (p0 && p1) do
        enq %out0, %in0.data (deq %in0, p0 := 0)

sendB:
    when (p0 && !p1) do
        enq %out0, %in1.data (deq %in1, p0 := 0)

drainA:
    when (%in0.tag != EOL && %in1.tag == EOL) do
        enq %out0, %in0.data (deq %in0)

drainB:
    when (%in0.tag == EOL && %in1.tag != EOL) do
        enq %out0, %in1.data (deq %in1)

bothDone:
    when (%in0.tag == EOL && %in1.tag == EOL) do
        nop (deq %in0, deq %in1)
```

Static Insts	6
Avg Insts/Iteration	2
Speedup vs PC+RegQueue (Fig 5)	5×
Speedup vs PC+Augmented (Fig 7)	3×

Fig. 8. Triggered instruction merge sort worker.

written into the tail of an output channel, a data register, or the vector of predicate registers. Predicate results can be written into one or more predicate registers.

Figure 8 shows our merge sort expressed using triggered instructions. Note the density of the trigger control decisions—each trigger reads at least two explicit boolean predicates. Additionally, conditions for the queues being *notEmpty* or *notFull* are recognized implicitly. Only the comparison between the actual multi-bit queue data values is done using the ALU datapath, as represented by the `doCheck` instruction. Predicate `p0` is used to indicate that the check has been performed, while `p1` holds the result of the comparison. Note also the lack of over-serialization. Only the explicitly programmer-managed sequencing using `p0` is present.

An example TIA PE is illustrated in Figure 9. The PE is pre-configured with a static set of instructions. The triggers for these instructions are then continuously evaluated by a dedicated hardware scheduler that dispatches legal instructions to the datapath for execution. At any given scheduling step, the trigger for zero, one, or more instructions can evaluate to true. The guarded action model — and by extension our triggered

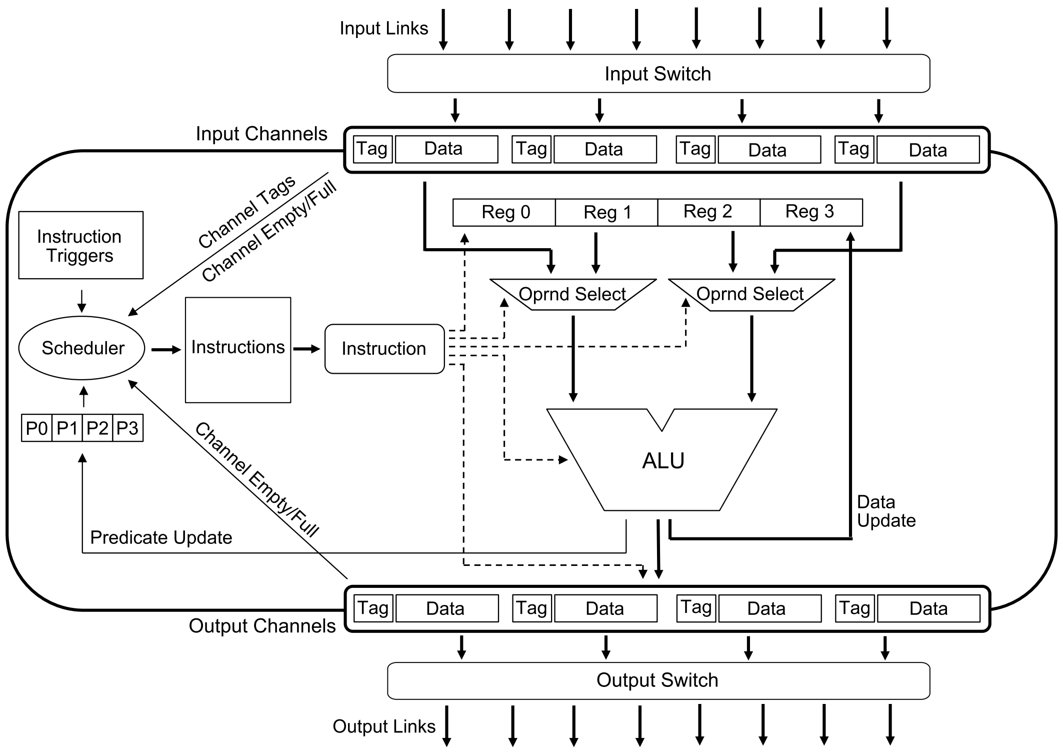


Fig. 9. A triggered-instruction based PE.

instruction model — allows all such instructions to fire in parallel subject to datapath resource constraints and conflicts.

The high-level microarchitecture of a TIA hardware scheduler is shown in Figure 10. The scheduler uses standard combinatorial logic to evaluate the programmer-specified query functions for each trigger based on values in the architectural state elements. This yields a set of instructions that are eligible for execution, among which the scheduler selects one or more depending on the datapath resources available. The example shown in this figure illustrates a scalar data-path that can only fire one instruction per cycle, therefore the scheduler selects one out of the available set of ready-to-fire instructions using a priority encoder.

As with any architecture, a triggered-instruction architecture is subject to a number of parameterization options and their associated cost-vs-benefit tradeoffs. *Architectural* parameters include the number of instances of each class of architectural state element (data registers, predicate registers, etc.), the set of data and predicate functions supported by the datapath, the scope and flexibility of the trigger functions, and the number of input operands and output destinations. The design space of *microarchitectural* alternatives includes scheduler implementation choices, scalar vs. superscalar datapaths, pipelining strategies, etc. An exhaustive investigation of the entire design space is outside the scope of this work. To provide the reader with some intuition on what a reasonably balanced TIA PE could look like, we provide an example architectural configuration in Figure 11. This is also the configuration we use for our evaluation in Section 6.

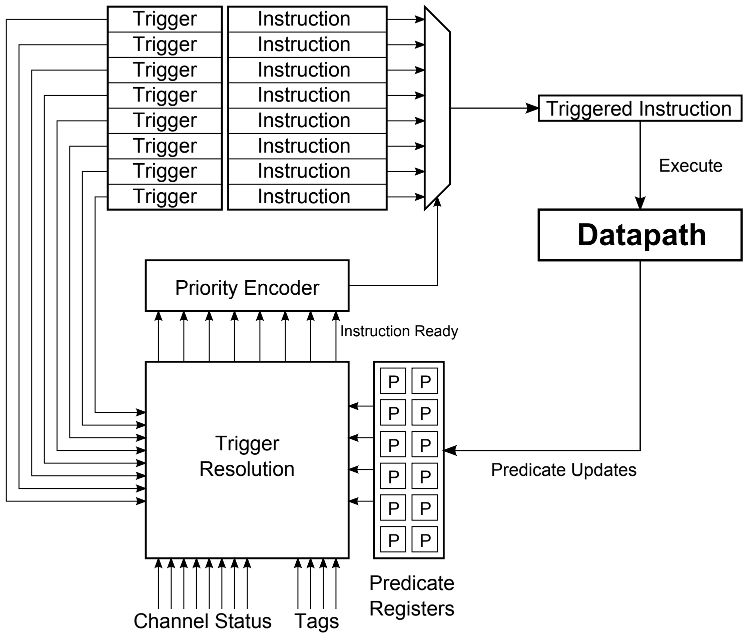


Fig. 10. Microarchitecture of a TIA scheduler.

Sources per Instruction	2
Registers	8
Predicates	8
Max Triggered Instructions per PE	16

Fig. 11. Example PE Architecture Parameters.

3.1. Observations on the Triggered Model

Having defined the basic structure of a triggered instruction architecture, we are now in a position to make some key observations:

— A TIA PE does not have a program counter or any notion of a static *sequence* of instructions. Instead, there is a limited *pool* of triggered instructions that are constantly bidding for execution on the datapath. This fits very naturally into a spatial programming model where each PE is statically *configured* with a small pool of instructions instead of streaming in a sequence of instructions from an instruction cache.

— Observe that there are no branch or jump instructions in the triggered ISA—every instruction in the pool is eligible for execution if its trigger conditions are met. Thus, every triggered instruction can be viewed as a multi-way branch into a number of possible states in an FSM.

— With clever use of predicate registers, a TIA can be made to emulate the behavior of other control paradigms. For example, a sequential architecture can be emulated by setting up a vector of predicate registers to represent the current state in a sequence—essentially, a program counter. Predicate registers can also be used to emulate classic predication modes, branch delay slots and speculative execution. Triggered instructions is a superset of many traditional control paradigms. The cost of this generality is scheduler area and timing complexity, which imposes a restriction on the number of triggers (and thus, the number of instructions) that the hardware can monitor at all times. While this restriction would be crippling for a temporally programmed architecture, it is reasonable in a spatially-programmed framework because of the low number of instructions typically mapped to a pipeline stage in a spatial workload.

— The hardware scheduler is built from combinatorial logic — it simply is a tree of AND gates. This means that only the state equations that require re-evaluation will cause the corresponding wires in the scheduler logic to swing and consume dynamic power. In the absence of channel activity or internal state changes, the scheduler does not consume any dynamic power whatsoever. The same control equations would have been evaluated using a chain of branches in a PC-based architecture.

Altogether, triggered instructions allow individual PEs to efficiently react to incoming messages, making an intelligent decision based on the local state of the PE. A TI scheduler uses a single unified scheme to monitor both the one-bit channel status registers and the local predicate registers in order to quickly and cheaply make a trigger decision. Spatial PEs are the "endpoints" of producer/consumer relationships. In the next section we present an efficient scheme for the actual transport of control and data between producers and consumers.

4. COMMUNICATIONS AND INTER-PE CONTROL

Triggered instructions capture the local control logic programmed onto a single PE, but the spatial fabric must also support some form of inter-PE coordination and data passing. One possibility is to design a global control architecture with a centralized "master" directing non-autonomous PEs. However, access to shared centralized control state is likely to be expensive and limit scalability of the fabric. Distributed control, though scalable, often makes it challenging to *map* an algorithm into a distributed program without either A) introducing bugs and deadlocks, or B) using knowledge about properties of the underlying network fabric. Existing spatial fabrics have solved this using *systolic* or *synchronous* networking schemes that cause neighboring PEs to move exactly in lockstep, which can limit performance.[4]

4.1. Latency-Insensitive Channels: A Flexible Architectural Foundation

In order to make this mapping process tractable while avoiding over-synchronization, we take inspiration from prior work on a fully distributed and scalable control paradigm based on *latency-insensitive design* [Carloni et al. 2001]. This paradigm has previously been successfully applied to hardware design [Pellauer et al. 2009; Vijayaraghavan and Arvind 2009], FPGA-based algorithm mapping [Fleming et al. 2012], and hardware-software communication [King et al. 2012].

In this design paradigm, local modules (whether hardware or software) are unable to make assumptions about the timing characteristics involved with communicating with other modules in the system. Instead, coordination is piggybacked onto data communications. The arrival (or absence) of a message on an input communication channel is itself an implicit form of control synchronization. This works well in a spatial architecture because of the significant amount of inter-PE dataflow required to achieve fine-grained producer-consumer pipelines. These dual-purpose communication channels are called *latency-insensitive channels* [Fleming et al. 2012].

A properly latency-insensitive system has the following properties:

[4]There is an analogy between a traditional Single-Instruction-Multiple-Data (SIMD) architecture and a systolic spatial fabric. Both approaches seek to obtain cheaper hardware implementation by potentially limiting performance to that of the slowest thread or PE, respectively.

— Modules connected via latency-insensitive channels do not share any data or control state *except through these channels*.
— Latency-insensitive channels are non-lossy, in-order channels. Transmitted messages remain in the channel until dequeued by the receiver.
— Traffic on any given latency-insensitive channel is not allowed to indefinitely block the delivery of traffic on any other channel.
— Latency-insensitive channels allow at least one message in flight, i.e., have at least one buffer. Each channel may allow more, but the exact amount may not be known statically.
— Modules connected via a latency-insensitive channel are not allowed to make any assumptions about the total amount of buffering available in the channel, nor about the the latency of message delivery along the channel.

As a corollary to the above points, if producer P_0 serially sends messages A and B down channel 0, and producer P_1 sends C and D down channel 1, then a consumer that reads both channels may receive the messages in the following legal orders: A, B, C, D or A, C, B, D or A, C, D, B or C, A, B, D or C, A, D, B or C, D, A, B. Which of these orderings is actually observed at runtime may be unpredictable (for instance, because of network congestion), and therefore a properly latency-insensitive program must be able to tolerate all of them (though they may not all be observed in practice).

We choose the latency-insensitive paradigm because the restrictions it imposes allow the architectural interface of the communication network (i.e., the channel interface described in Section 3) to be separated from the implementation and topology of the network. For example, a latency-insensitive channel can be implemented simply as a register with a valid bit, or as a deep memory-based circular ring buffer. A latency-insensitive module or program should be functionally oblivious to these implementation choices (though performance may vary).

Furthermore, properly latency-insensitive code may be deployed in a variety of different mapping scenarios. In one deployment, a producer module may be placed directly next to a consumer, with an uncontended direct link between them that contains a small amount of buffering. Later, the same code may be deployed in a scenario where the same producer is far from the consumer, and the data must travel heavily-contended network links, but is also given more buffering as a result. These scenarios may certainly differ in performance, but latency-insensitivity should ensure that the program always produces the correct result. This gives spatial architects a large degree of freedom when developing efficient mapping algorithms.

4.2. Efficient Channel Implementation via Static Virtual Circuits

As we will show in Section 6, a distributed program for typical spatial workload contains a large number of cross-PE channels. It is unrealistic to assume that hardware can provision dedicated wires to acheive full bandwidth for all flows— and indeed, nearest neighbor communication is too restrictive to cover all cases. Luckily, there is an abundance of prior art on building on-chip networks that create shared *virtual channels* (VCs) for traditional MIMD processors [Dally and Towles 2003; Peh and Jerger 2009]. These channels have the ordering and non-blocking requirements we desire, while efficiently multiplexing limited wire and buffer resources dynamically, and using established flow-control crediting techniques to acheive good bandwidth.

Unfortunately, these solutions use dynamic packet routers with deep pipelines that are not a good fit for spatial architectures. These routers are typically designed for a scenario where each network node is a traditional processor core, which amortizes the cost of the router. In contrast, each PE in a spatial architecture is much smaller, which makes a traditional dynamic router impractical. Furthermore, this level of dynamism is actually not needed in a spatial context. In a typical multicore, the overwhelming majority of traffic on the OCN is coherence-protocol traffic. Caching schemes intentionally hash the packet destination to distributed home nodes in order to avoid creating hotspots and bottlenecks. In such a scenario dynamic packet addressing and deadlock-free routing schemes are fundamental requirements.

In contrast, in the spatial scenario, the producer/consumer flows are statically determined when the program is loaded during the configuration step. Most flows are nearest-neighbor, and often can be mapped so that there is no contention (see Section 6). We can leverage these facts to create *statically allocated virtual circuits* through the on-chip network. We can then map the programming construct of latency-insensitive channels onto these circuits, which multiplex limited physical link wires between the flows mapped onto them on a hop-by-hop basis, while maintaining non-blocking and deadlock-freedom via reverse credit flows, which are similarly statically mapped and managed.

When connecting spatial PEs, these circuits have the following advantages over traditional VC multiplexing:

— A circuit's route can be determined soley by the mapper (or programmer) and need not take the most direct route to the destination, for instance for congestion avoidance.
— A uniform deadlock-free routing scheme such as X-Y routing is not needed, but rather deadlock freedom can be insured on a circuit-by-circuit basis.
— Packets can be routed based solely on the channel-ID that is transmitting, rather than through a control flit pre-pended to the message. This results in fewer bits transmitted per message and simpler hardware per router.
— Similarly, the routing decision for each circuit can be distributed at configuration time to each hop involved in that circuit. For example, if route 5 goes North-East-East, this would result in "North" configured to entry 5 of the routing table of the first hop, "East" to the second hop, and so on. This means that each router will only see information local to its particular switch, and will not pay energy transmitting routing directions for following hops.
— At configuration time, different buffering resources may be allocated to different flows depending on their expected criticality to program performance. This could be used either to increase fairness or quality-of-service of the overall system.

Figure 12 depicts an implementation of a single input and output channel as they interface with the PE's scheduler and datapath. State elements with dotted borders are written statically, during the configuration of the program, and define the virtual circuit that implements the channel. This includes the link in/out and channel ID of the next (or previous) hop that for data transmission. For the input channel, the multiplexer into the data buffer is controlled statically by an unchanging value indicating which incoming link's data should be latched. This has a beneficial effect on both power consumption and critical path. For output, the link direction a given channel is trying to transmit onto is similarly statically determined.

Elements with dashed borders represent micro-architectural scoreboarding that is invisible to the end user. This includes the current occupancy level of the local buffer, and current crediting status. Note that these credits are tracked between this hop and the immediate next, and do not represent end-to-end flow control, which can conservatively limit effective network bandwidth.

One important design decision is the placement of buffering both before and after the PE. From a correctness standpoint, this buffering is overkill—it has been shown that buffering either input or output exclusively is sufficient. Within the latency insensitive design paradigm, we choose to use both because it allows the system to better tolerate

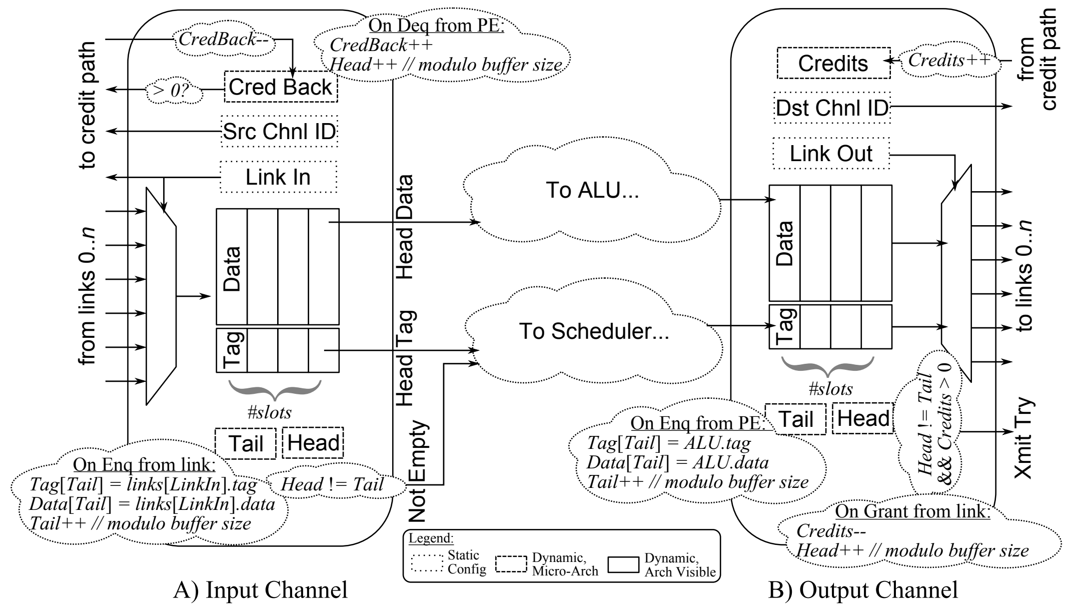


Fig. 12. Implementation details of the PE interface to a single input/output channel shown previously in Figure 9. For simplicity, head/tail pointers are depicted. Smaller buffers could be implemented with occupancy bits, or cross-channel buffer-slot sharing schemes considered.

variabilities in latency. If a downstream consumer becomes temporarily slowed due to congestion or cache misses, the producer PE may still pre-buffer results into the output channel, even if the channel itself is out of credits to downstream hops. Similarly, even if a PE is stalled on some rare-but-slow operation, the network is still able to pre-buffer some amount of new input, reducing congestion and allowing for quick resumption when the hiccup is resolved.

Note the natural fit between the input/output channels' NotFull and NotEmpty signals and the triggered instruction scheduler. To the trigger resolution logic, these are simply extra 1-bit inputs, not really any different than local predicate registers. Whether these 1-bit signals represent a complicated occupancy and/or credit status is immaterial to the instruction scheduling decision.

Finally, note that the PEs can also be programmed to operate as network routers in a multihop traversal by adding a triggered instruction that dequeues the data from an input channel and enqueues it into an output channel.

In the following sections we evaluate and quantify the benefit that the latency-insensitive communication mechanism brings to common control paradigms found in real workloads.

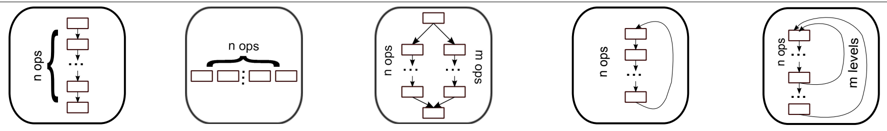
5. EVALUATION: CONTROL IDIOMS

In this section we evaluate the quantitative benefit of triggered instructions and latency-insensitive channels by examining a number of control idioms that arise frequently in spatially-programmed workloads.

Table I describes common control idioms that occur when controlling a single PE, and compares implementations of each idiom on a triggered architecture to implementations on the PC+RegQueue and PC+Augmented architectures described in Section 2. These idioms are generally related to efficiently encoding the graph of a single finite-state machine onto a given instruction set. Table II repeats this comparison using

Table I. Dynamic instruction cost of common *intra-PE* control idioms.

(A) Seq Composition (autonomous)	(B) Par Composition (autonomous)	(C) Control Dependence	(D) Loop (k iterations)	(E) Nested Loop (k iterations per level)



Idiom Legend

Idiom	PC+RegQueue	PC+Augmented	Triggered Instructions	TI Advantage over PC+Augmented
(A)	D.ops = n *(serialized)*	D.ops = n *(serialized)*	D.ops = n *(serialized)*	-
(B)	D.ops = n *(serialized)*	D.ops = n *(serialized)*	D.ops = n *(unordered)*	eliminates serialization
(C)	D.ops = m or $n+1^\dagger$ C.ops = 1 † *1 comparison*	D.ops = m or $n+1^\dagger$ F.ops = m or n † *1 comparison*	D.ops = m or $n+1^\dagger$ C.ops = 0; F.ops = 0 † *1 comparison*	eliminates m or n F.ops
(D)	D.ops = $n*k+k^\dagger$ C.ops = k † *k comparisons*	D.ops = $n*k+k^\dagger$ C.ops = k † *k comparisons*	D.ops = $n*k+k^\dagger$ C.ops = 0 † *k comparisons*	eliminates k C.ops
(E)	D.ops = $k^m*n + \frac{k(k^m-1)}{(k-1)}^\dagger$ C.ops = $\frac{k(k^m-1)}{(k-1)}$ † $\frac{k(k^m-1)}{(k-1)}$ *comparisons*	D.ops = $k^m*n + \frac{k(k^m-1)}{(k-1)}^\dagger$ C.ops = $\frac{k(k^m-1)}{(k-1)}$ † $\frac{k(k^m-1)}{(k-1)}$ *comparisons*	D.ops = $k^m*n + \frac{k(k^m-1)}{(k-1)}^\dagger$ C.ops = 0 † $\frac{k(k^m-1)}{(k-1)}$ *comparisons*	eliminates $\frac{k(k^m-1)}{(k-1)}$ C.ops

D.ops = data ops, C.ops = control ops, F.ops = predicated false ops, autonomous = internal activities of a PE

common inter-PE communication paradigms using latency-insensitive channels, thus demonstrating the efficacy of pairing each ISA with this scheme.

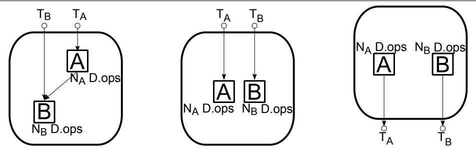
Across both tables we see some general patterns emerging. First, TI is never less efficient than a PC-based approach, i.e. it never requires more instructions. Second, TI removes all control operations such as branches. In a classic PC-based setting, the accepted rule of thumb is that about 1 in every 4-5 instructions is a branch [Emer and Clark 1984]. In this setting TI's expected benefit would be around 20%. However in Section 6.3 we demonstrate that the fine-grained producer-consumer nature of spatially-programmed codes means that control makes up 44% of all operations, which increases the benefit of TI significantly.

Finally, TI removes the over-serialization problem presented in Section 2.3. This has several benefits, but they are harder to quantify directly. First, as the equations in Table II demonstrate, there are certainly scenarios where over-serialization results in no penalty because the data arrives in the order that matches the static sequence chosen by the compiler. If the compiler can precisely schedule cross-PE data delivery rates then it is possible that this deficiency will never be exposed. In practice, the numerous sources of variable dynamic latency (memory hierarchy, network contention, data-dependent divergence, etc.) mean that there is plenty of opportunity to take advantage of the ability to break over-serialization. Additionally, dealing with messages as they arrive can allow backwards credit-flow to the producer PE to begin earlier, which can increase effective OCN throughput.

Breaking over-serialization can be accomplished by finding independent operations. These can be found from two sources. The first source is local parallelism in the PE's dataflow graph, in which case computation can start as the data arrives, i.e., classical dynamic instruction reordering. The second source arises when the spatial compiler chooses to place unrelated sections of the overall algorithm dataflow graph onto a sin-

Table II. Dynamic instruction cost of common *inter-PE* control idioms.

(F) Seq Composition (queue input)	(G) Par Composition (queue input)	(H) Par Composition (queue output)



Idiom Legend

Idiom	PC+RegQueue	PC+Augmented	Triggered Instructions	TI Advantage over PC+Augmented

(F)	D.ops = $N_A + N_B$
	Q.ops = 2
		D.ops = $N_A + N_B$
		Q.ops = 0
		wait = $T_A + max(T_B - T_A - N_A, 0)$
		wait = $T_A + max(T_B - T_A - N_A, 0)$
			D.ops = $N_A + N_B$
			Q.ops = 0
			wait = $T_A + max(T_B - T_A - N_A, 0)$
			wait = $T_A + max(T_B - T_A - N_A, 0)$
				-

(G)	D.ops = $N_A + N_B$
	Q.ops = 2
	(serialized $A \rightarrow B$)
		D.ops = $N_A + N_B$
		Q.ops = 0
		wait = if ($T_A > T_B$)
		T_A
		else
		$T_A + max(T_B - T_A - N_A, 0)$
		(serialized $A \rightarrow B$)
			D.ops = $N_A + N_B$
			Q.ops = 0
			wait = if ($T_A > T_B$)
			$T_B + max(T_A - T_B - N_B, 0)$
			else
			$T_A + max(T_B - T_A - N_A, 0)$
				if ($T_A > T_B$)
				$min(N_B, T_A - T_B)$
				wait filled

(H)	D.ops = $N_A + N_B$
	Q.ops = 2
	(serialized $A \rightarrow B$)
		D.ops = $N_A + N_B$
		Q.ops = 0
		wait = if ($T_A > T_B$)
		T_A
		else
		$T_A + max(T_B - T_A - N_B, 0)$
		(serialized $A \rightarrow B$)
			D.ops = $N_A + N_B$
			Q.ops = 0
			wait = if ($T_A > T_B$)
			$T_B + max(T_A - T_B - N_A, 0)$
			else
			$T_A + max(T_B - T_A - N_B, 0)$
				if ($T_A > T_B$)
				$min(N_A, T_A - T_B)$
				wait filled

D.ops = data ops, Q.ops = queue ops, wait = serialization penalty, queue = PE responding to external events, T_i = time of channel availability

gle PE, statically partitioning the registers between them and statically interleaving operations, i.e. compiler-directed multithreading. On a PC-based architecture, the serialization restriction is a significant barrier to a compiler's ability to statically partition one thread of control between unrelated sections of a single algorithm. The dynamic data production/consumption rates must be known to schedule the code—both for efficiency, and to avoid deadlock. On a TI architecture we expect compiler-directed multithreading of non rate-limiting PEs to be a common and important optimization.

To reiterate these benefits, since a TI architecture does not impose any ordering between instructions unless explicitly specified, it can gain the ILP benefits of an out-of-order issue processor *without the expensive instruction window and reorder buffer*. Simultaneously, a TI machine can take advantage of multi-threading *without duplicating data and control state*, but by the compiler partitioning resources as it sees fit. Of course there is a hardware cost associated with this benefit—the TI PE must have a scheduler (see Figure 10) that can efficiently evaluate the program's triggers.

6. EVALUATION: WORKLOADS

6.1. Approach

The objective of our quantitative evaluation in this section is threefold:

(1) To demonstrate the effectiveness of a TIA-based spatial architecture compared to a traditional high-performance sequential architecture.

(2) To demonstrate the benefits of using TIA-based PEs in a spatial architecture compared to PC-based PEs using the PC+RegQueue and PC+Augmented architectures described in Section 2.

(3) To demonstrate that latency-insensitive channels are an efficient and appropriate paradigm for inter-PE communication, and represent a viable alternative to transferring data via shared memory coherence protocols.

The main challenge with the first objective is that raw performance of a spatial accelerator is a function of area and memory bandwidth allocated to the accelerator, and parallelism available in the workload. Because spatial workloads generally exhibit good scalability, providing raw performance requires assessing a particular design point with a specific set of area/bandwidth values. However, since the purpose of this paper is to present a control paradigm for spatial architectures in general, we instead present performance numbers area-normalized against a typical host processor – namely a single 3.4 GHz out-of-order superscalar Intel® Core™ i7-2600 core.

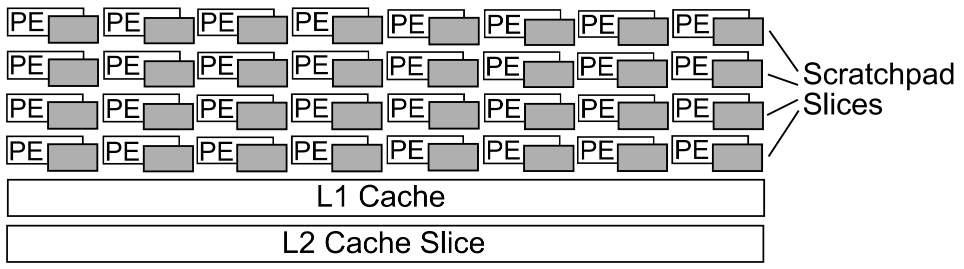
Our evaluation fabric is a scalable spatial architecture built from an array of TIA PEs organized into *blocks*, which form the granularity of replication of the fabric. Each block contains a grid of interconnected PEs, a set of scratchpad slices distributed across the block, a private L1 cache, and a slice of a shared L2 cache that scales with the number of blocks on the fabric. Figure 13 provides an illustration of a block and the parameters that we use in our evaluation. Note that our evaluation PEs use 32-bit integer/fixed-point datapaths and do not include hardware floating point units (which is orthogonal to triggered instructions and beyond the scope of this evaluation). Area estimates of each PE were obtained via implementation feasibility analysis discussed further in Section 6.4. Area estimates for the caches, register files, multipliers, and on-chip network were added using existing industry results. As a reference, 12 blocks (each including PEs, caches, etc.) are about the same size as our baseline i7-2600 core (including L1 and L2 caches), normalized to the same technology node.

We developed a detailed cycle-accurate performance model of our spatial accelerator using Asim, an established performance modeling infrastructure [Emer et al. 2002]. We model the detailed microarchitecture of each TIA PE in the array, the mesh interconnection network, L1 and L2 caches, and DRAM.

We evaluate our spatial fabric on application kernels from a variety of domains. We do this under the assumption that the computationally-intensive portions of the workload will be offloaded from the main processor, which will handle peripheral tasks like setting up the memory and handling rare-but-slow cases. As a baseline we used sequential software implementations running on the i7-2600 host processor. When possible, we chose existing optimized workload implementations. In other cases, we auto-vectorized the workload using the Intel® C/C++ compiler (`icc`) version 13.0, enabling processor-specific ISA extensions.

For our second evaluation objective, we analyze how much of the overall speedup benefit is attributable to triggered instructions (as opposed to spatial programming in general) using the same framework described above. We demonstrate this by examining the critical loops that form the rate-limiting steps in the spatial pipeline of our workloads. We implemented the loops on spatial accelerators using the traditional program-counter based approaches. This analysis demonstrates how frequently the triggered instruction control idiom advantage presented in Tables I and II translates to practical improvements.

For our third evaluation objective, we gather statistics of the channel usage of our workloads. We begin by showing what percentage of the channels are related to memory interaction, and what percentage represent more efficient direct inter-PE communuication. We also gather statistics on potential for network conflicts, demonstrating how different these static flows are from traditional dynamically routed packets. Finally, we gather dynamic usage statistics, showing how much memory traffic and link contention occur in practice, and to what extent latency-insensitive channels allow us to unlock the potential bandwidth of our on-chip network and cache hierarchy.



PEs	32
Network	Mesh (1 cycle link latency)
Scratchpad	8KB (distributed)
L1 Cache	4KB (4 banks, 1KB/bank)
L2 Cache	24 KB shared slice
DRAM	200 cycle latency
Estimated Clock Rate	2 GHz

Fig. 13. Block Illustration and Parameters.

Table III. Target Workloads for Evaluation.

Workload	**Berkeley Dwarf** [Asanovic et al. 2006]	**Domain**	**Comparison Software Implementations**
AES-CBC	Combinational Logic	Cryptography	Intel reference using AES - ISA extensions
KMP String Search	Finite State Machines	Various	Non-public optimized implementation
Dense Matrix Multiply	Dense Linear Algebra	Scientific Computing	Intel® MKL implementation [Geijin and Watts 1997]
FFT	Spectral Methods	Signal Processing	FFT-W with auto-vectorization
Graph500-BFS	Graph Traversal	Supercomputing	Non-public optimized implementation
k-means Clustering	Dense Linear Algebra	Data mining	MineBench implementation with auto-vectorization
Merge Sort	Map/Reduce	Databases	Non-public optimized implementation
Flow classifier	Finite State Machines	Networking	Non-public optimized implementation
SHA-256	Combinational Logic	Cryptography	Intel reference (x86 assembly)

6.2. Evaluation Application Kernels

For our analysis we have purposely chosen workloads spanning the space of data parallelism, pipeline parallelism, and graph parallelism. Table III presents an overview of the chosen kernels.

The triggered instruction versions of these kernels we implemented directly in our PE's assembly language and hand-mapped spatially across our fabric. (In the future we expect this to be done by automated tools from higher-level source code.) We offer these insights on the workloads' amenability to spatial programming:

— **AES-CBC:** Encryption with cipher-block chaining implemented using a memoized table in which byte substitution is performed. The algorithm is performed on a 4x4 grid of 8 bits apiece. One PE is responsible for providing the computation for a single byte, exposing 16-way parallelism.

Table IV. Percentage of dynamic instructions that are branches in rate-limiting step inner loop.

	AES	DMM	FFT	Flow Classifier	Graph-500
PC+RegQ	58%	50%	36%	50%	50%
PC+Aug	6%	33%	11%	50%	40%

	k-means	KMP Search	Merge Sort	SHA-256	Average
PC+RegQ	69%	8%	70%	63%	50%
PC+Aug	29%	14%	50%	22%	28%

— **Dense Matrix Multiply:** We adapt the SUMMA algorithm [Geijin and Watts 1997] by blocking problem size to the fabric. Input data is pipelined through loader PEs. Each worker PE operates on an 8*8 resultant matrix.
— **KMP String Search:** We adapt the Knuth-Morris-Pratt (KMP) [Knuth et al. 1977] string search algorithm by slicing the text string into small segments and distributing it across a large number of PE workers. Another set of PEs are configured as pattern state machine generators and servers. A spatial implementation is able to slide the string window by simply rotating the logical order of the workers, discarding the block of text from the oldest worker and shifting in a new block in its place.
— **FFT:** We adapt a Fast Fourier Transform by blocking the complex-multiply butterfly structure into a size specific to our number of PEs. A control FSM re-uses this block to compose an FFT of arbitrary size.
— **Flow Classifier:** Network packet masking is parallelized by allocating different segments of the packet to different PEs. The hash key calculation is pipelined through a large number of PEs. The final comparison for matching flows is parallelized by processing multiple segments of the flow in parallel on multiple PEs.
— **Graph500-BFS:** The graph500 benchmark is meant to span multiple nodes of a supercomputer. We simulate what a single node would look like if enhanced with a spatial accelerator. We are able to pipeline the loading, testing, and updating of the nodes to expose a large number of in-flight memory requests.
— **k-means Clustering:** Our implementation maps the Euclidean distance function for a single cluster to a PE. Input data, along with the current nearest cluster, is streamed through the PEs in order to compare against all clusters.
— **Merge Sort:** Described previously in Section 2.2.
— **SHA-256:** The tight inner-loop is spatially mapped across PEs, with each function being mapped to a separate PE. Key generation is parallelized.

6.3. Performance Results

Figure 14 demonstrates the magnitude of performance improvement that can be achieved from using a spatially-programmed accelerator. Across our workloads, we observe area-normalized speedup ratios ranging from 3× (FFT) to around 22× (SHA-256) compared to the performance of the traditional core, with a geometric mean of 8×.

Now let us analyze how much of this benefit is attributable to the use of triggered instructions by comparing the rate-limiting inner loops of our workloads to implementations on spatial architectures using the PC+RegQueue and PC+Augmented control schemes.

Table IV shows the average frequency of branches in the dynamic instruction stream for the PC-based spatial architectures. The branch frequency ranges from 8% to 70%, with an average of 50%. These inner loops are all very branchy and dynamic—far more than traditional sequential code.

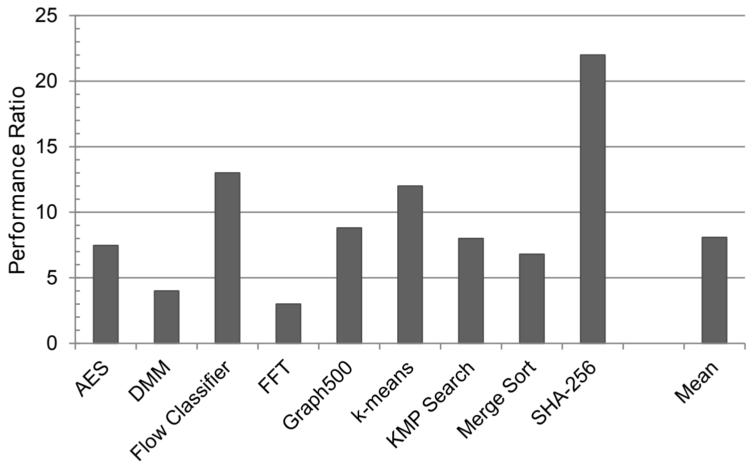


Fig. 14. Area-normalized performance ratio of a TIA-based spatial accelerator compared to a high-performance out-of-order core.

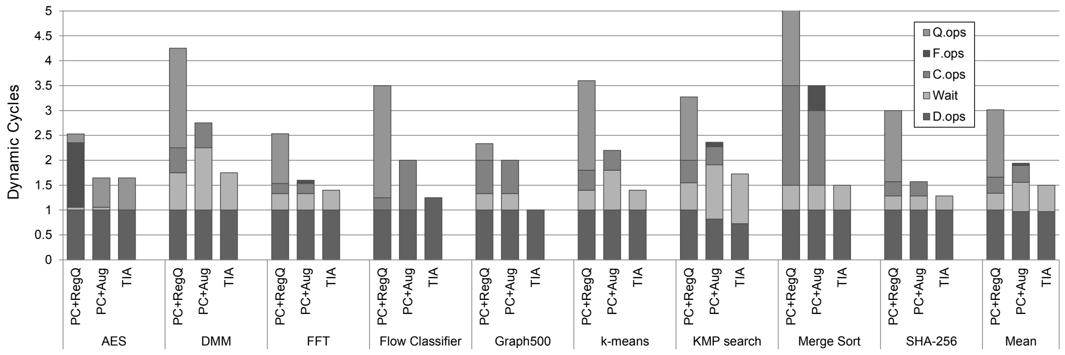


Fig. 15. Breakdown of dynamic execution cycles in rate-limiting inner loops normalized to D.ops executed by PC+RegQueue.

This dynamism manifests itself as additional control cycles for both PC-based architectures, as shown in Figure 15. This figure shows the dynamic execution cycles for all architectures broken down into cycles spent on operations in each category defined in in Section 5. The cycle counts are all normalized to the number of D.ops (Data Computation operations) executed by PC+RegQueue. We augment this data with Figures 16 and 17, which respectively show the static and dynamic (average) instruction/op counts in the inner loops of rate-limiting steps for each workload.

The data in these figures demonstrates that the control idiom efficiencies presented in Tables I and II are applicable to real-world kernels. Specifically:

— TIA demonstrates a significant reduction in dynamic instructions executed compared to both PC+RegQueue (64%) and PC+Augmented (28%) on average, and an average performance improvement of 2.0× vs. PC+RegQueue and 1.3× vs. PC+Augmented in the critical loops. A large part of the performance gained by PC+Augmented over PC+RegQueue is from the reduction of Queue Management ops. TIA benefits from this too but gets a further performance boost over PC+Augmented from a reduction in Control ops and Predicated-False ops.

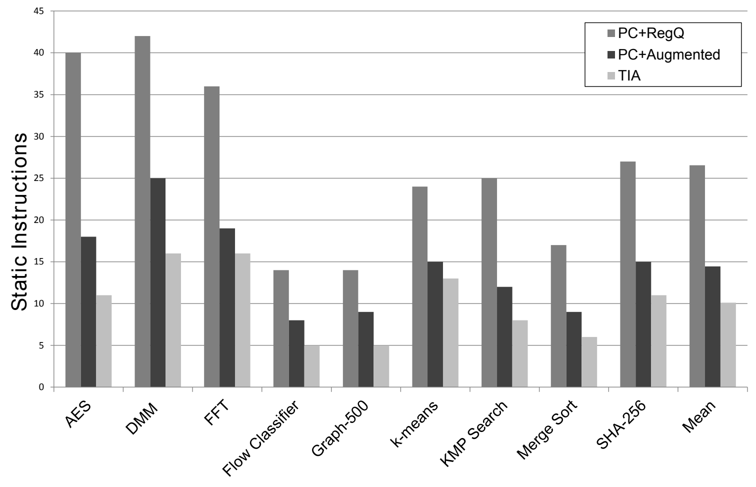


Fig. 16. Static instruction counts for rate-limiting inner loops.

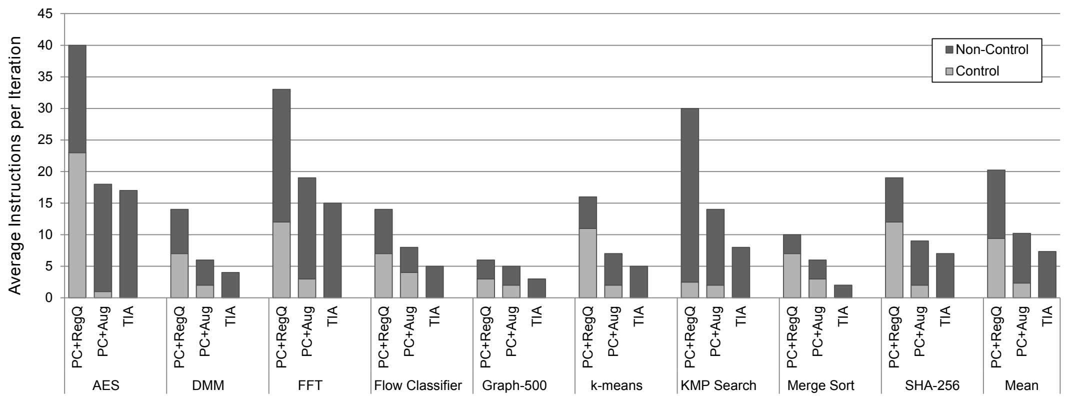


Fig. 17. Average dynamic instruction counts for rate-limiting inner loops.

— An additional benefit of TIA over PC+Augmented comes from a reduction in Wait cycles. This is most evident in the k-means (50%), Graph500 (100%) and SHA-256 (40%) workloads. This is due to the ability of triggered instructions to avoid unnecessary serialization. Note that because these are critical rate-limiting loops in the spatial pipeline, there are fewer opportunities for multiplexing unrelated work onto shared PEs. Despite this, the workloads show benefits from avoiding over-serialization.

— The workload that sees the largest benefit from triggered instructions is Merge Sort. Merge Sort has the highest dynamic branch rate (70%) of all workloads on the PC+RegQueue architecture. It also spends a number of cycles polling queues. PC+Augmented eliminates all the queue-polling cycles, resulting in 1.6× performance improvement in the rate-limiting step. TIA further cuts down a large number of control cycles, leading to a further 2.3× performance improvement vs. PC+Augmented and a cumulative 3.7× performance benefit over PC+RegQueue.

— On the average, PC+Augmented does not see a significant benefit from predicated execution for these spatially-programmed workloads.

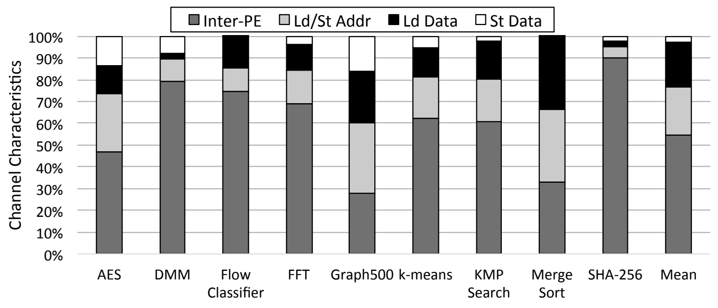


Fig. 18. Breakdown of workload latency-insensitive channels by type.

Table V. Average static hop count and contention potential for workload channels.

	AES	**DMM**	**FFT**	**Flow Classifier**	**Graph-500**
Total Channels	60	468	414	339	50
Avg Hops per Channel	1.3	1.12	1.28	1.64	2.06
Avg Channels per Link	2.00	2.10	2.68	1.75	1.58

	k-means	**KMP Search**	**Merge Sort**	**SHA-256**	**Average**
Total Channels	37	23	1155	40	287
Avg Hops per Channel	1.24	1.70	1.36	1.35	1.45
Avg Channels per Link	1.53	2.71	2.61	2.16	2.06

— Triggered instructions use a substantially smaller static instruction footprint. The reduction in footprint compared to PC+RegQueue is particularly significant — 62% on average. PC+Augmented's enhancements help reduce footprint but TIA still has 30% fewer static instructions on average.

The static code footprint of these rate-limiting inner loops is in general fairly small across all architectures. This observation, along with the real-world performance benefits we observed versus traditional high-performance architectures, provides strong evidence of the viability and effectiveness of the spatial programming model with small, tight loops arranged in a pipelined graph.

Regarding latency-insensitive channels, Figure 18 and Table V show the static breakdown of the number and type of each channel. These demonstrate that all workloads contain a large number of inter-PE channels, and that direct communication without going through shared memory is a feasible communication paradigm for these pipelined graphs. Furthermore, the low hop count indicate that much of this communication is neighbor-to-neighbor, and thus makes the most effective use possible of distributed network bandwidth. The channels-per-link tracks the number of virtual circuits multiplexed on the same physical links to show the *potential* for contention and bandwidth reduction in the network. For example, if a single link is shared between two channels, then the network will operate at full bandwidth as long as each channel is not injecting a message more frequently than every two cycles. Only the links with at least one virtual circuit using them are counted.

Figure 19 and Table VI supplement this information with dynamic traffic breakdowns for four workloads with differing static ratios of memory channels. The "effective" channel bandwidth refers to the percentage of time that channel traffic achieves full network bandwidth, without being slowed by contention—if this reaches 100% then the result is the same as if the hardware had provisioned dedicated wires for all channels in the system. These demonstrate that contention is rare in prac-

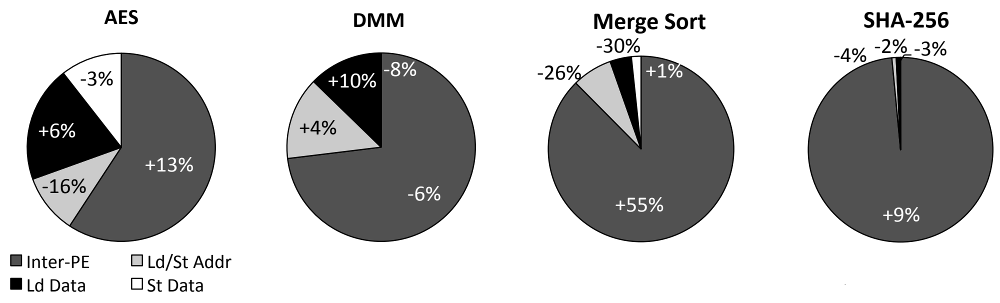


Fig. 19. Breakdown of dynamic traffic by channel type. The number is the change versus static percentage of channels (Figure 18). This demonstrates which channel classes see the heaviest dynamic use.

Table VI. Average dynamic contentions per link per cycle, organized by channel type.

	AES	DMM	Merge Sort	SHA-256
Average Delays from Contention: Memory-PE	1.08	0	0.6	0
Average Delays from Contention: Inter-PE	0	0	0	0.03
Effective LI-Channel Bandwidth: Memory-PE	47.96%	100%	61.93%	100%
Effective LI-Channel Bandwidth: Inter-PE	99.99%	100%	100%	97.23%

tice, and is almost always related to memory interfacing – “hotspots” around limited cache port resources. Overall, these results confirm that static virtual circuits are an efficient implementation of the latency-insensitive channel paradigm. Just as spatially-programmed loops have different branch and control ratios to traditional codes, spatially-programmed networks have static properties that allow architects to extract efficiency without over-provisioning hardware for unpredictable dynamic cases.

6.4. Implementation Feasibility Analysis

We collaborated with circuit-design experts to lay out a TIA PE in a state-of-the-art industry technology process. The resulting 2-stage pipelined PE has a comparable number of gate levels in the critical path to a high-performance commercial microprocessor. The large degree of replication in a spatial fabric would, however, justify even further design effort to optimize the PEs.

The hardware scheduler is the centerpiece of a TIA PE. Scheduler implementation cost is one of the primary factors that bounds the scalability of PE size in a triggered control model. Fortunately, the nature of spatial programming is such that small, efficient PEs are effective.

Our implementation analysis shows that the area cost of the TIA hardware scheduler is *less than 2%* of a PE’s overall area, much of which is occupied by its architectural state (registers, input/output channel buffers, predicates and instruction storage), datapath logic (operand multiplexers, functional units, etc.) and microarchitectural control overheads—none of which are unique to triggered control. This is not surprising—the core of the TIA scheduler is essentially a few 1-bit wide trees of AND gates feeding into a priority encoder. For our chosen parameterization, this logic is dwarfed by everything else in the PE.

Similarly, scheduler power consumption is small compared to the rest of the PE. The scheduler logic does not consume dynamic power unless there is a change in predicate states. When this happens, the only wires that swing are the ones that are recomputing the changed control signals. This manner of computing control is more power-efficient than executing datapath instructions to compute the same results. In a degenerate scenario where the PE is walking down a sequence of stages in a gray-coded FSM,

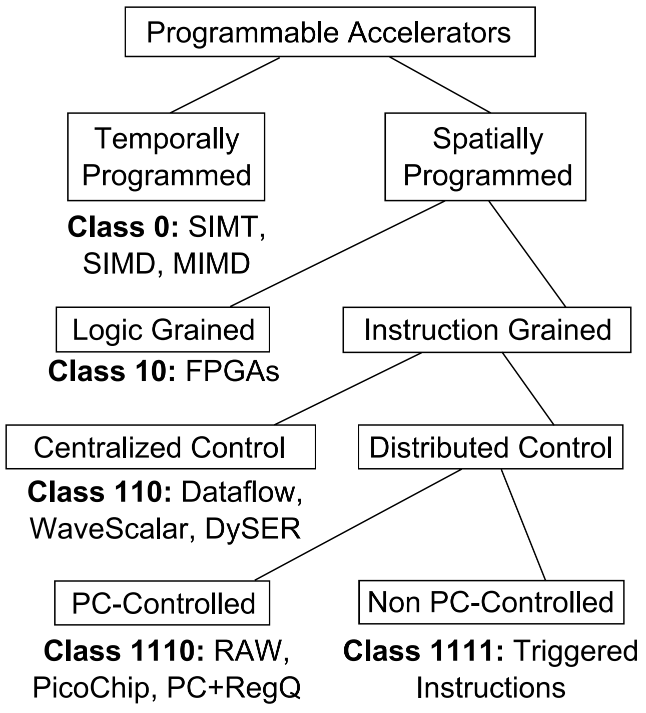

Programmable Accelerators
Temporally Programmed
Spatially Programmed
Class 0: SIMT, SIMD, MIMD
Logic Grained
Instruction Grained
Class 10: FPGAs
Centralized Control
Distributed Control
Class 110: Dataflow, WaveScalar, DySER
PC-Controlled
Non PC-Controlled
Class 1110: RAW, PicoChip, PC+RegQ
Class 1111: Triggered Instructions

Fig. 20. A taxonomy of programmable accelerators.

at most 1-2 predicate bits swing each cycle. The power consumed in this scenario is negligible.

7. RELATED WORK

We classify prior work on architectures for programmable accelerators according to the taxonomy shown in Figure 20 (although some have been proposed as standalone processors instead of accelerators complementing a general purpose CPU). Temporal architectures (class 0 in the taxonomy) are best suited for data-parallel workloads and are outside of the scope of this work. Within the spatial domain (classes $1x$), the tradeoffs between logic-grained architectures (class 10) such as FPGAs and instruction-grained architectures (classes $11x$) are well understood ([Mirsky and DeHon 1996; Hauser and Wawrzynek 1997; Mei et al. 2003]). In this section, we focus our attention on prior work on instruction-grained spatial architectures with centralized and distributed control paradigms.

7.1. Centralized PE Control Schemes

In the centralized approach (class 110), a fabric of spatial PEs is paired with a centralized control unit. This unit maintains overall program execution order, managing PE configuration. The results of PE execution may influence the overall flow of control, but in general the PEs are not making autonomous decisions.

Transport-Triggered Architectures [Hoogerbrugge and Corporaal 1994] is a scheme where the functional units in the system are exposed to the compiler, which then uses MOV operations to explicitly route data through the transport network. Overall control flow is maintained by a global program counter. Operation execution is triggered by the arrival of data from the network, but no other localized control exists.

TRIPs is an *explicit dataflow graph execution* (EDGE) processor which utilizes many small PEs to execute general-purpose applications [Burger et al. 2004]. TRIPs dynamically fetches and schedules large VLIW instruction blocks across the small PEs using centralized program-counter based control tiles. While large reservation stations within each PE enable "when-ready" execution of instructions, only single-bit predication is used within PEs to manage small amounts of control flow.

WaveScalar is a dataflow processor for general-purpose applications that does not utilize a program counter [Swanson et al. 2007]. A PE consists of an ALU, input and output network connections, and a small window of 8 instructions. Blocks of instructions known as waves are mapped down onto the PEs, and additional "WaveAdvance" instructions are allocated at the edges to help manage coarse grained or loop-level control. Conditionals are handled by converting control flow instructions to data flow, resulting in filtering instructions that conditionally pass values to the next part of the dataflow graph. In WaveScalar there is no local PE register state; when an instruction issues the result must be communicated to another PE across the network.

DySER integrates a circuit-switched network of ALUs inside the datapath of contemporary processor pipeline [Govindaraju et al. 2011]. DySER maps a single instruction to each ALU and does not allow memory or complex control flow operations within the ALUs. TIA enables efficient control flow and spatial program mapping across PEs, enabling high-utilization of ALUs with PEs without the need for an explicit control core. Other recent work such as Garp [Hauser and Wawrzynek 1997], Chimaera [Ye et al. 2000], and ADRES [Mei et al. 2003] similarly integrate LUT-based or coarse grained reconfigurable logic controlled by a host processor, either as a coprocessor or within the processor's datapath.

MATRIX [Mirsky and DeHon 1996] is an array of 8-bit function units with a configurable network. With different configurations, MATRIX can support VLIW, SIMD or Multiple-SIMD computations. The key feature of the MATRIX architecture was claimed to be its ability to deploy resources for control based on application regularity, throughput requirements and space available.

PipeRench [Schmit et al. 2002] is a coarse-grained RL system designed for virtualization of hardware to support high-performance custom computations through self-managed dynamic reconfiguration. It is constructed from 8-bit Processing Elements. The functional unit in each PE contains eight 3-input LUTs that are identically configured.

Note that in the *dataflow* computing paradigm, instructions are dispatched for execution when tokens associated with input sources is ready. Each instruction's execution results in the broadcast of new tokens to dependent instructions. Classical dataflow architectures such as [Dennis and Misunas 1975; Arvind and Nikhil 1990] used this as a centralized control mechanism for spatial fabrics. However other projects such as [Burger et al. 2004], [Swanson et al. 2007] use token triggering to issue operations in the PEs, whereas the centralized control unit uses a more serialized approach.

In a dataflow-triggered PE, the token-ready bits associated with input sources are managed by the micro-architecture. The TI approach, in contrast, replaces these bits with a vector of architecturally-visible predicate registers. By specifying triggers that span multiple predicates, the programmer can choose to use these bits to indicate data readiness, but can also use them for other purposes, such as control flow decisions. In classic dataflow multiple pipeline stages are devoted to marshaling tokens, distributing tokens, and scoreboarding which instructions are ready. A "Wait-Match" pipeline stage must dynamically pair incoming tokens of dual-input instructions. In contrast, the set of predicates to be updated by an instruction in the TI is encoded in the instruction itself. This both reduces scheduler implementation cost and removes the token-related pipeline stages.

Smith et al. [Smith et al. 2006] extend the classic static dataflow model by allowing each instruction to be gated on the arrival of a predicate of a desired polarity. This approach adds some control-flow efficiency to dataflow, providing for implicit disjunction of predicates by allowing multiple predicate-generating instructions to target a single destination instruction, and implicit conjunction by daisy-chaining predicate operations. While this makes conjunctions efficient, it can lead to an over-serialization of the possible execution orders inherent in the original non-predicated dataflow graph. In contrast, compound conjunctions are explicitly supported in triggered instructions, allowing for efficient mapping of state transitions that would require multiple instructions in dataflow predication.

7.2. Distributed PE Control Schemes

In the distributed approach (classes $111x$), a fabric of spatial PEs is used without a central control unit. Instead, each PE makes localized control decisions, and overall program-level coordination is established using distributed software synchronization. Within this domain, PC-based control model (long established for controlling distributed temporal architectures—class 0) is a tempting choice, as demonstrated by this rich body of prior work. By removing the program counter, the TI approach (class 1111) offers many opportunities to improve efficiency (Section 6.3).

The RAW project is a coarse-grained computation fabric, consisting of 16 large cores with instruction and data caches that are directly connected through a register-mapped and circuit-switched network [Taylor et al. 2002]. While applications written for RAW are spatially mapped, program counter management and serial execution of instructions reduces efficiency, and makes the cores on RAW sensitive to variable latencies, which TIA overcomes using instruction triggers.

The Asynchronous Array of simple Processors (AsAP) is a 36-PE processor for DSP applications, with each PE executing independently using instructions in a small instruction buffer and communicating using register-mapped network ports [Yu et al. 2006]. While early research on AsAP avoided the need to poll for ready data, later work extended the original architecture to support 167-PEs and zero-overhead looping to reduce control instructions [Truong et al. 2009]. Triggered instructions not only reduce the amount of control instructions but also enable data-driven instruction issue, overcoming the serialization of AsAP's program-counter based PE.

Picochip is a commercially available 308-PE accelerator for DSP applications [Panesar et al. 2006]. Each PE has a small instruction and data buffer, and communication is performed with explicit *put* and *get* commands. A strength of Picochip is compute density, but the architecture is limited to serial 3-way LIW instruction issue using a program counter. Triggered instructions enable control flow at low cost and dynamic instruction issue dependent on data arrival, resulting in less instruction overhead.

8. CONCLUSION

We believe that spatial parallelism is a promising computing paradigm with the potential to achieve significant performance improvement over traditional high-performance architectures for a number of important workloads, many of which do not exhibit uniform data parallelism. Our simulated performance estimates on a triggered-instruction based spatial architecture confirm the potential of this style of computing, showing an average area-normalized performance that is $8\times$ better than a high-end sequential processor across a range of workloads.

Triggered instructions provide a uniform solution to the control problem for a PE in a spatially-programmed architecture, allowing the PE to execute autonomous control loops efficiently as well as react quickly to messages on communication channels. The latency-insensitive channel paradigm allows the mapping, routing and buffering of this communication to be separated and cleanly abstracted from the PE programming. Together, these mechanisms also avoid over-serialization, providing the benefits of dynamic instruction reordering and multithreading without any additional hardware. Our evaluation demonstrates the cumulative benefits of all these effects, with our triggered-instruction PE achieving 2.0× better performance than a baseline PE with PC-based control, and 1.3× better performance than an optimized version.

The triggered control model is feasible within a spatially-programmed environment because the amount of static instruction state that must be maintained in each PE is small, allowing for inexpensive implementation of a triggered-instruction hardware scheduler. Our implementation analysis confirms this, showing that the scheduler occupies less than 2% of the area of the PE.

These results provide a solid foundation of evidence for the merit of a triggered-instruction based spatial architecture. The ultimate success of this paradigm will be premised on overcoming a number of challenges, including providing a tractable memory model, dealing with the finite size of the spatial array, and providing a high-level programming and debugging environment. Our ongoing work makes us optimistic that these challenges are surmountable.

REFERENCES

Arvind and R. S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

Bluespec, Inc. 2007. Bluespec System Verilog Reference Guide. (2007).

Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. 2004. Scaling to the End of Silicon with EDGE Architectures. *Computer* 37, 7 (July 2004), 44–55. DOI:http://dx.doi.org/10.1109/MC.2004.65

L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 20, 9 (Sep 2001), 1059–1076. DOI:http://dx.doi.org/10.1109/43.945302

K. Mani Chandy and Jayadev Misra. 1988. *Parallel Program Design: a Foundation*. Addison-Wesley.

Katherine Compton and Scott Hauck. 2002. Reconfigurable Computing: A Survey Of Systems and Software. *ACM Computer Survey* 34, 2 (June 2002), 171–210. DOI:http://dx.doi.org/10.1145/508352.508353

William Dally and Brian Towles. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Jack B. Dennis and David P. Misunas. 1975. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd annual Symposium on Computer Architecture*. 126–132.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. DOI:http://dx.doi.org/10.1145/360933.360975

Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. 2002. Asim: A Performance Model Framework. *Computer* 35, 2 (2002), 68–76.

Joel S. Emer and Douglas W. Clark. 1984. A Characterization of Processor Performance in the vax-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*. 301–310. DOI:http://dx.doi.org/10.1145/800015.808199

Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. 2012. Leveraging Latency-insensitivity to Ease Multiple FPGA Design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 175–184. DOI:http://dx.doi.org/10.1145/2145694.2145725

Robert A. Van De Geijin and Jarell Watts. 1997. *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. Technical Report.

Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of 17th International Conference on High Performance Computer Architecture (HPCA)*.

J.R. Hauser and J. Wawrzynek. 1997. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 12–21.

Jan Hoogerbrugge and Henk Corporaal. 1994. Transport-Triggering vs. Operation-Triggering. In *Lecture Notes in Computer Science 786, Compiler Construction*. Springer-Verlag, 435–449.

Myron King, Nirav Dave, and Arvind. 2012. Automatic Generation of Hardware/Software Interfaces. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 325–336.

Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM Journal of Computing* 6, 2 (1977), 323–350.

H. T. Kung. 1986. The CMU Warp Processor. In *Supercomputers: Algorithms, Architectures, and Scientific Computation*, F. A. Matsen and T. Tajima (Eds.). 235–247.

A. Marquardt, V. Betz, and J. Rose. 2000. Speed and Area Tradeoffs in Cluster-Based FPGA Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8, 1 (Feb. 2000), 84 –93. DOI:http://dx.doi.org/10.1109/92.820764

B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of 13th International Conference on Field-Programmable Logic and Applications*. 61–70.

Duane G. Merrill and Andrew S. Grimshaw. 2010. Revisiting Sorting for GPGPU Stream Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 545–546. DOI:http://dx.doi.org/10.1145/1854273.1854344

E. Mirsky and A. DeHon. 1996. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 157–166.

Gajinder Panesar, Daniel Towner, Andrew Duller, Alan Gray, and Will Robbins. 2006. Deterministic Parallel Processing. *International Journal of Parallel Programming* 34, 4 (Aug. 2006), 323–341. DOI:http://dx.doi.org/10.1007/s10766-006-0019-9

Li-Shiuan Peh and Natalie Enright Jerger. 2009. *On-Chip Networks* (1st ed.). Morgan and Claypool Publishers.

M. Pellauer, M. Adler, D. Chiou, and J. Emer. 2009. Soft connections: Addressing the hardware-design modularity problem. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. 276–281.

H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R.R. Taylor. 2002. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the 2002 IEEE Custom Integrated Circuits Conference*. 63–66.

Aaron Smith, Ramadass Nagarajan, Karthikeyan Sankaralingam, Robert McDonald, Doug Burger, Stephen W. Keckler, and Kathryn S. McKinley. 2006. Dataflow Predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. 89–102.

Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The WaveScalar Architecture. *ACM Transactions on Computer Systems* 25, 2, Article 4 (May 2007), 54 pages. DOI:http://dx.doi.org/10.1145/1233307.1233308

M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.W. Lee, W. Lee, and others. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (2002), 25–35.

D.N. Truong, W.H. Cheng, T. Mohsenin, Zhiyi Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, Zhibin Xiao, E.W. Work, J.W. Webb, P.V. Mejia, and B.M. Baas. 2009. A 167-Processor Computational Platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits* 44, 4 (April 2009), 1130–1144. DOI:http://dx.doi.org/10.1109/JSSC.2009.2013772

Muralidaran Vijayaraghavan and Arvind Arvind. 2009. Bounded Dataflow Networks and Latency-insensitive Circuits. In *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'09)*. IEEE Press, Piscataway, NJ, USA, 171–180. http://dl.acm.org/citation.cfm?id=1715759.1715781

Z-A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. 2000. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*. 225–235.

Zhiyi Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. 2006. An Asynchronous Array of Simple Processors for DSP Applications. In *Solid-State Circuits Conference (ISSCC), Digest of Technical Papers*. 1696–1705.