Efficient Spatial Processing Element Control via Triggered Instructions

Angshuman Parashar
Michael Pellauer
Michael Adler
Bushra Ahsan
Neal Crago
Intel

Daniel Lustig
Princeton University

Vladimir Pavlov
Intel

Antonia Zhai
University of Minnesota

Mohit Gambhir
Aamer Jaleel
Randy Allmon
Rachid Rayess
Stephen Maresh
Joel Emer
Intel

The authors present triggered instructions, a novel control paradigm for arrays of processing elements (PEs) aimed at exploiting spatial parallelism. Triggered instructions eliminate the program counter and allow programs to transition concisely between states without explicit branch instructions. The approach also allows efficient reactivity to inter-PE communication traffic and provides a unified mechanism to avoid overserialized execution.

Recently, single-instruction, multiple-data (SIMD) and single-instruction, multiple-thread (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. Unfortunately, not every workload exhibits sufficiently uniform data parallelism to exploit 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, many important workload domains exhibit such kernels, including signal processing, media codecs, cryptography, compression, pattern matching, and sorting.

As such, one way to boost these workloads' performance efficiency is to add a new spatially programmed accelerator to the system, complementing the existing SIMD/SIMT accelerators. Although FPGAs are very general in their ability to map a workload's computation, control, and communication structure, their datapaths based on lookup tables (LUTs) are deficient in computational density compared to a traditional microprocessor—much less a SIMD engine. Furthermore, FPGAs suffer from a low-level programming model

Published by the IEEE Computer Society

inherited from logic prototyping that includes unacceptably long compilation times, no support for dynamic context switching, and often inscrutable debugging features.

Tiled arrays of coarse-grained arithmetic logic unit- (ALU)-style datapaths can achieve higher computational density than FPGAs.[1-3] Several prior works 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.[4-6] Other approaches incorporate autonomous control at each PE using a program counter (PC).[7-9] 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. (For more information, see the "Related Work in Instruction-Grained Spatial Architectures" sidebar.)

In this article, we present *triggered instructions,* a novel control paradigm for ALU-style datapaths for use in arrays of PEs aimed at exploiting spatial parallelism. Triggered instructions remove the program counter completely, letting the PE transition between states of one or more finite-state machines (FSMs) without executing instructions in the datapath to determine the next state. This also lets the PE react quickly to incoming messages on communication channels. In addition, triggered instructions provide a unified mechanism to avoid overserialized execution, essentially achieving the effect of techniques such as dynamic instruction reordering and multithreading, which require distinct hardware mechanisms in a traditional sequential architecture.

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

Background and motivation

To understand the benefits that triggered instructions can provide to a spatially programmed architecture, we must first understand how spatially programmed architectures in general can play a role in the computational landscape, and why traditional program-counter-based approaches are limited in this context.

Spatial programming architectures

Spatial programming is a paradigm whereby an algorithm's dataflow graph is broken into regions connected by producer-consumer relationships. Input data is then streamed through this pipelined graph. Ideally, the number of operations in each stage is kept small, because performance is usually determined by the rate-limiting step.

Just as vectorizable 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 could even be slower than other expressions of the algorithm. A shared-memory 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.

In contrast, a typical spatial-programming architecture is a fabric of hundreds of small PEs connected directly via an on-chip network. Given enough PEs, an algorithm can be taken to the extreme of mapping a single operation in the kernel's dataflow graph to each PE, resulting in a very fine-grained pipeline. In practice, it is desirable to have a small number of local operations in each PE, allowing for a better balance between local control decisions and pipeline parallelism.

To illustrate this, let's explore how a well-known workload can benefit from spatial programming. Consider the simple spatially

Related Work in Instruction-Grained Spatial Architectures

We classify prior work on architectures for programmable accelerators according to the taxonomy shown in Figure A (although some have been proposed as stand-alone 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 this article's scope. Within the spatial domain (classes 1x), the trade-offs between logic-grained architectures (class 10), such as field-programmable gate arrays (FPGAs) and instruction-grained architectures (classes 11x), are well understood.[1-3] In this sidebar, we focus on prior work on instruction-grained spatial architectures with centralized and distributed control paradigms.

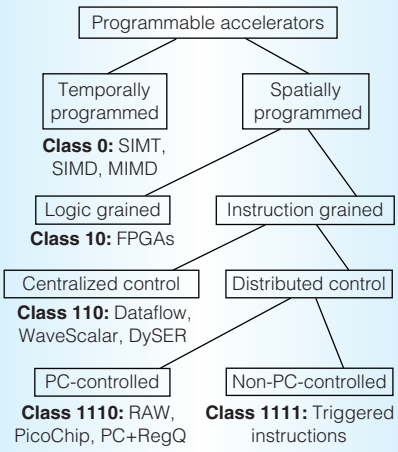


Figure A. A taxonomy of programmable accelerators. Each leaf node represents a distinguishable class of previously proposed architectures.

Centralized processing element control schemes

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

In the Transport-Triggered Architectures scheme, the system's functional units are exposed to the compiler, which then uses MOV operations to explicitly route data through the transport network.[4] A global program counter maintains overall control flow. Operation execution is triggered by the arrival of data from the network, but no other localized control exists.

Trips (Tera-op, Reliable, Intelligently adaptive Processing System) is an explicit dataflow graph execution (EDGE) processor that uses many small PEs to execute general-purpose applications.[5] Trips dynamically fetches and schedules very-large instruction word (VLIW) blocks across the small PEs using centralized program-counter-based control tiles. Although 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 doesn't use a program counter.[6] A PE consists of an arithmetic logic unit (ALU), I/O network connections, and a small window of eight instructions. Blocks of instructions called waves are mapped 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 (Dynamically Specialized Execution Resource) integrates a circuit-switched network of ALUs inside the datapath of contemporary processor pipeline.[7] DySER maps a single instruction to each ALU and doesn't 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,[2] Chimaera,[8] and ADRES[3] (Architecture for Dynamically Reconfigurable Embedded System) similarly integrate lookup-table-based or coarse-grained reconfigurable logic controlled by a host processor, either as a coprocessor or within the processor's datapath.

Matrix is an array of 8-bit function units with a configurable network.[1] With different configurations, Matrix can support VLIW, SIMD, or Multiple-SIMD computations. The key feature of the Matrix architecture is its ability to deploy resources for control based on application regularity, throughput requirements, and space available.

PipeRench is a coarse-grained reconfigurable logic system designed for virtualization of hardware to support high-performance custom computations through self-managed dynamic reconfiguration.[9] It is constructed from 8-bit PEs. The functional unit in each PE contains eight three-input lookup tables (LUTs) that are identically configured.

In the dataflow computing paradigm, instructions are dispatched for execution when tokens associated with input sources are ready. Each instruction's execution results in the broadcast of new tokens to dependent instructions. Classical dataflow architectures used this as a centralized control mechanism for spatial fabrics.[10,11] However, other projects use token triggering to issue operations in the PEs,[5,6] whereas the centralized control unit uses a more serialized approach.

In a dataflow-triggered PE, the microarchitecture manages the token-ready bits associated with input sources. The triggered-instruction approach, in contrast, replaces these bits with a vector of architecturally visible predicate registers. By specifying triggers that span multiple predicates, the programmer use these bits to indicate data readiness or for other purposes, such as control flow decisions. In a classic dataflow architecture, 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 triggered-instruction approach is encoded in the instruction itself. This reduces scheduler implementation cost and removes the token-related pipeline stages.

Smith et al. extend the classic static dataflow model by allowing each instruction to be gated on the arrival of a predicate of a desired polarity.[12] This approach adds some control-flow efficiency to the dataflow model, 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. Although this makes conjunctions efficient, it can lead to an overserialization of the possible execution orders inherent in the original nonpredicated 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.

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, the PC-based control model (long established for controlling distributed temporal architectures—class 0) is a tempting choice, as demonstrated by a rich body of prior work. By removing the program counter, the triggered-instruction approach (class 1111) offers many opportunities to improve efficiency.

The Raw project is a coarse-grained computation fabric comprising 16 large cores with instruction and data caches that are directly connected through a register-mapped and circuit-switched network.[13] Although 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.[14] Although 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.[15] 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.[16] Each PE has a small instruction and data buffer, and communication is performed with explicit put and get commands. One strength of Picochip is its computational density, but the architecture is limited to serial three-way LIW instruction issue using a program counter. Triggered instructions enable control flow at low cost and dynamic instruction issue that is dependent on data arrival, resulting in less instruction overhead.

References

1. E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1996, pp. 157-166.
2. J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1997, pp. 12-21.
3. B. Mei et al., "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," *Proc. 13th Int'l Conf. Field-Programmable Logic and Applications*, 2003, pp. 61-70.
4. J. Hoogerbrugge and H. Corporaal, "Transport-Triggering vs. Operation-Triggering," *Compiler Construction*, LNCS 786, Springer-Verlag, 1994, pp. 435-449.
5. D. Burger et al., "Scaling to the End of Silicon with EDGE Architectures," *Computer*, vol. 37, no. 7, 2004, pp. 44-55.
6. S. Swanson et al., "The WaveScalar Architecture," *ACM Trans. Computer Systems*, vol. 25, no. 2, 2007, pp. 4:1-4:54.
7. V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for Energy Efficient Computing," *Proc. 17th Int'l Conf. High Performance Computer Architecture* (HPCA), 2011, pp. 503-514.
8. Z.-A. Ye et al., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," *Proc. 27th Int'l Symp. Computer Architecture*, 2000, pp. 225-235.
9. H. Schmit et al., "PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology," *Proc. IEEE Custom Integrated Circuits Conf.*, 2002, pp. 63-66.
10. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Proc. 2nd Ann. Symp. Computer Architecture*, 1975, pp. 126-132.
11. K. Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Computers*, vol. 39, no. 3, 1990, pp. 300-318.
12. A. Smith et al., "Dataflow Predication," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2006, pp. 89-102.
13. M. Taylor et al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, 2002, pp. 25-35.
14. Z. Yu et al., "An Asynchronous Array of Simple Processors for DSP Applications," *Proc. Solid-State Circuits Conf.*, 2006, pp. 1696-1705.
15. D. Truong et al., "A 167-Processor Computational Platform in 65 nm CMOS," *IEEE J. Solid-State Circuits*, vol. 44, no. 4, 2009, pp. 1130-1144.
16. G. Panesar et al., "Deterministic Parallel Processing," *Int'l J. Parallel Programming*, vol. 34, no. 4, 2006, pp. 323-341.

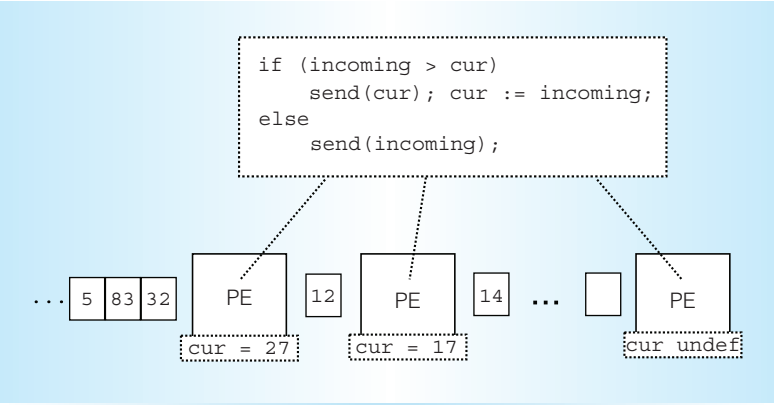


Figure 1. Example of a spatially programmed sort. Although a pedagogical example, this workload demonstrates several interesting properties.

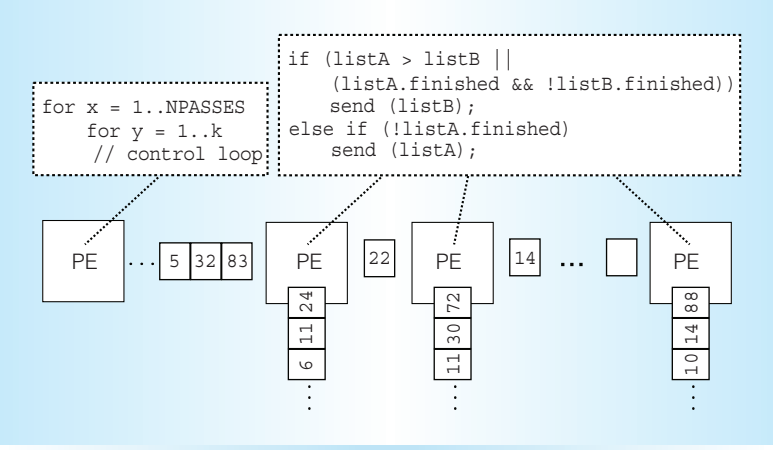


Figure 2. Expanding the example to a more realistic spatial merge sort capable of sorting lists of any size. The large merge radix results in fewer total loads and stores to sort the list, replacing them with more efficient direct PE-to-PE communication.

mapped sorting program shown in Figure 1. 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 is 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 do only one (an insertion sort, with a total of k^2 comparisons). However, despite its naiveté, this workload demonstrates some remarkable properties. First, the system's peak utilization is good—in the final step, all k datapaths can simultaneously execute a comparison. Second, the communication between PEs is local and parallel—on a typical mesh fabric, it's easy to map this workload so that no network contention will ever occur. 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 characterize the benefits as follows:

- Direct communication uses roughly 20 times lower power than communication through a level-1 (L1) cache, as the overheads of tag matching, load-store queue search, 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 the remaining traffic.

Finally, it is straightforward to expand our toy example into a realistic merge sort engine that can sort a list of any size (see Figure 2). First, we begin by programming a PE into a small control FSM that handles breaking the array into subarrays of size k and looping over the subarrays. Second, we slightly change the worker PEs' programming so that they are doing a merge of two distinct sorted sublists. With these changes, our toy workload is now a radix k merge sort capable of sorting a list of size n in $n \times \log_k(n)$ 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, which is better than the current best-known GPGPU performance.[10]

Limitations of PC-based control

The PC-based control model has historically been the best choice for stand-alone CPUs that run arbitrary and irregular programs. Unfortunately, this model introduces unacceptable inefficiencies in the context of spatial programming. To understand these inefficiencies, let us code the merge sort PE shown in Figure 2. We must first address the representation of the queues that pass the sorted sublists between workers. In a multicore system, the typical approach is to use shared memory for the queue buffering, along with sophisticated polling mechanisms such as memory monitors. In a spatially programmed fabric, having hundreds of PEs communicating using shared memory would create unacceptable bandwidth bottlenecks—in addition to increased overheads of pointer chasing, address offset arithmetic, and head and tail occupancy comparisons. Thus, we don't consider shared memory communication queues in this article.

Instead, let us assume that the instruction-set architecture (ISA) directly exposes data registers and status bits corresponding to direct communication channels between PEs. 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 sublist has been reached (`EOL`). Figure 3 shows an example of the merge sort in this theoretical assembly language. Several inefficiencies are immediately noticeable. First, the worker uses active polling to test the queue status—an obvious power waste. Second, it falls victim to overserialization. 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 branch heavy when compared to that typically found on a traditional core, and some of these branches are hard to predict.

To be fair to this PC-based ISA, we must try to improve the architecture somehow. Table 1 summarizes the techniques that we explore.

One idea to improve queue accesses is to allow destructive reads of input channels. In such an ISA, the instruction's source fields 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 three instructions compared to Figure 3.

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, queue registers such as `%in0` would be treated specially—any instruction that attempts to read or write them would require the issue logic to test the empty or full bits and delay issue until the status becomes correct. Merge sort's implementation on this architecture is the same as in Figure 3, but removes the first three instructions entirely.

```
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.first, EOL, done
         jump    send_b
done:    deq     %in0
         deq     %in1
         return;
```

Static instructions: 18
Average instructions per iteration: 10
Average branches per iteration: 7

Figure 3. PC+RegQueue instruction-set architecture (ISA) merge sort worker representation using register-mapped queues. First, queue status is tested, then the end-of-list (EOL) condition is evaluated. Finally, the actual data comparison results in either a swap or pass. This results in a poor ratio of control decisions to useful work.

Table 1. Adding features to a PC-based ISA to improve efficiency for spatial programming.

Feature	Description	Notes
PC (baseline)	PEs use program counters and communicate using shared-memory queues	High latency, bottlenecks
+RegQueue	Expose register-mapped queues to ISA and 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 I/O registers without special instructions	Bubbles, overserialization
+QMultiThread	Stalling on empty or 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

```
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.first, EOL
       (p2)  jump    send_b
             nop     (deq %in0, deq %in1)
             return;
```

Static instructions: 9
Average instructions per iteration (Issued): 6
Average instructions per iteration (committed): 5
Average branches per iteration: 3
Speedup versus PC+RegQueue (see Figure 3): 1.4 times

Figure 4. PC+Augmented ISA merge sort worker takes advantage of the following features: implicit stalls on queue enqueue and dequeue, destructive queue reads, and classical predication. Together, these features reduce the overhead of the program counter, but the ratio of branches to useful work remains high.

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 multithreading. In this ISA, any read or 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 multiplexing logic, and scheduling fairness—run counter to the fundamental spatial-architecture principle of replicating simple PEs. In other words, the cost-to-benefit ratio of multithreading 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 can test and set a dedicated set of Boolean predicate registers. Figure 4 shows a reimplementation of the merge sort worker in a language with predication, implicit stalling, and destructive reads, which we name PC+Augmented. Note how little predication improves the control flow of the example. This is because of several limitations:

- Instructions can't read multiple predicate registers at once (inefficient conjunction).
- Composing multiple predicates into more complex Boolean expressions (such as disjunctions) must be done using the ALU itself.
- Jumping between regions requires that the predicate expectations be set correctly. (Note that the branch from `a_finished` is forced to use `p2` with a positive polarity.)
- Predicated false instructions introduce bubbles into the pipeline.

Taken together, these inefficiencies mean that conditional branching remains the most efficient way to express the majority of the code in Figure 4. Although we could continue to try to add features to PC-based schemes in order to improve efficiency, in the rest of this article we demonstrate that taking

```
rule sendA
when listA.first() != EOL && listB.first() != EOL && listA.data < listB.data do
       outList.send(listA.first()); listA.deq();
end rule
rule sendB
when listA.first() != EOL && listB.first() != EOL && listA.data >= listB.data do
       outList.send(listB.first()); listB.deq();
end rule
rule drainA
when listA.first() != EOL && listB.first() == EOL do
       outList.send(listA.first()); listA.deq();
end rule
rule drainB
when listA.first() == EOL && listB.first() != EOL do
       outList.send(listB.first()); listB.deq();
end rule
rule bothDone
when listA.first() == EOL && listB.first() == EOL do
       listA.deq(); listB.deq();
end rule
```

Figure 5. Traditional guarded-action merge sort worker algorithm. This paradigm naturally separates the representation of data transformation (via actions) from the representation of control flow (via guards). This results in a higher level of code readability, because the control decisions related to each action are naturally grouped and isolated.

a different approach altogether can efficiently address these issues while simultaneously removing overserialization and providing the benefits of multithreading.

Triggered instructions

A large degree of the inefficiency we have discussed here stems from the issue of efficiently composing Boolean control-flow decisions. 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,[11] Chandy and Misra's Unity,[12] and the Bluespec hardware description language.[13]

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. Figure 5 illustrates our merge sort worker in traditional guarded-action form. 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.

Triggered-instruction architecture

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 several possible queries on a PE's architectural state.

The architectural state of our proposed TIA PE is composed of four elements:

- A set of data registers (read/write)
- A set of predicate registers (read/write)
- A set of input-channel head elements (read only)
- A set of output-channel tail elements (write 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 many ways.

A *trigger* is a programmer-specified Boolean expression formed from the logical conjunction of a set of queries on the PE's architectural state. (Although the architecture natively allows only conjunctions in trigger expressions, disjunctions can be emulated by creating a separate triggered instruction for each disjunctive term.) A hardware scheduler evaluates triggers. The set of allowable trigger query functions is carefully chosen to maintain scheduler efficiency while allowing for much generality in the useful expressions. The query functions include the following:

- *Predicate register values (optionally negated):* A trigger can specify a requirement for one or more predicate registers to be either true or false (for example, `p0 && !p1 && p7`).
- *I/O 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 that an instruction attempts to write. The programmer doesn't 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 might specify a value that an input channel's tag must match (such as `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, computation, and write capabilities:

- An instruction can 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 can perform a data computation using one of the standard functions provided by the datapath's ALU. It can 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, such as reduction AND, OR, and XOR operations, bit extractions, and ALU flags such as overflow.
- An instruction can write the data result and the derived predicate result into a set of destinations within the PE's architectural state. Data results can be 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 6 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 multibit queue data values is done using the ALU datapath, as represented by the `doCheck` instruction. Predicate `p0` indicates that the check has been performed, whereas `p1` holds the result of the comparison. Note also the lack of overserialization.

Only the explicitly programmer-managed sequencing using p0 is present.

Figure 7 shows an example TIA PE. The PE is preconfigured 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-instruction model—allows all such instructions to fire in parallel subject to datapath resource constraints and conflicts.

Figure 8 shows the TIA hardware scheduler's high-level microarchitecture. The scheduler uses standard combinatorial logic to evaluate the programmer-specified query functions for each trigger on the basis of 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 in this figure illustrates a scalar datapath 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.

```
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 instructions: 6
Average instructions per iteration: 2
Speedup versus PC+RegQueue (see Figure 3): 5 times
Speedup versus PC+Augmented (see Figure 4): 3 times
```

Figure 6. The triggered instruction merge sort worker retains the clean separation of control and data transformation of the generalized guarded action version shown in Figure 5. The restriction is that the control decisions must be stored in single-bit predicate registers, and the action is limited to the granularity of one instruction. As a result, the sendA and sendB rules are refactored such that the comparison takes place in the earlier doCheck rule, which sets up predicate register p1 with the result of the comparison.

Observations about the triggered model

Having defined the basic structure of a TIA, we can now make some key observations.

A TIA PE doesn't 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.

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 multiway branch into a few 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 instruction is a superset of many traditional control paradigms. The costs of this generality are scheduler area and timing complexity, which impose a restriction on the number of triggers (and thus, the number of instructions) that the hardware can monitor at all times. Although this restriction would be crippling for a temporally programmed architecture, it

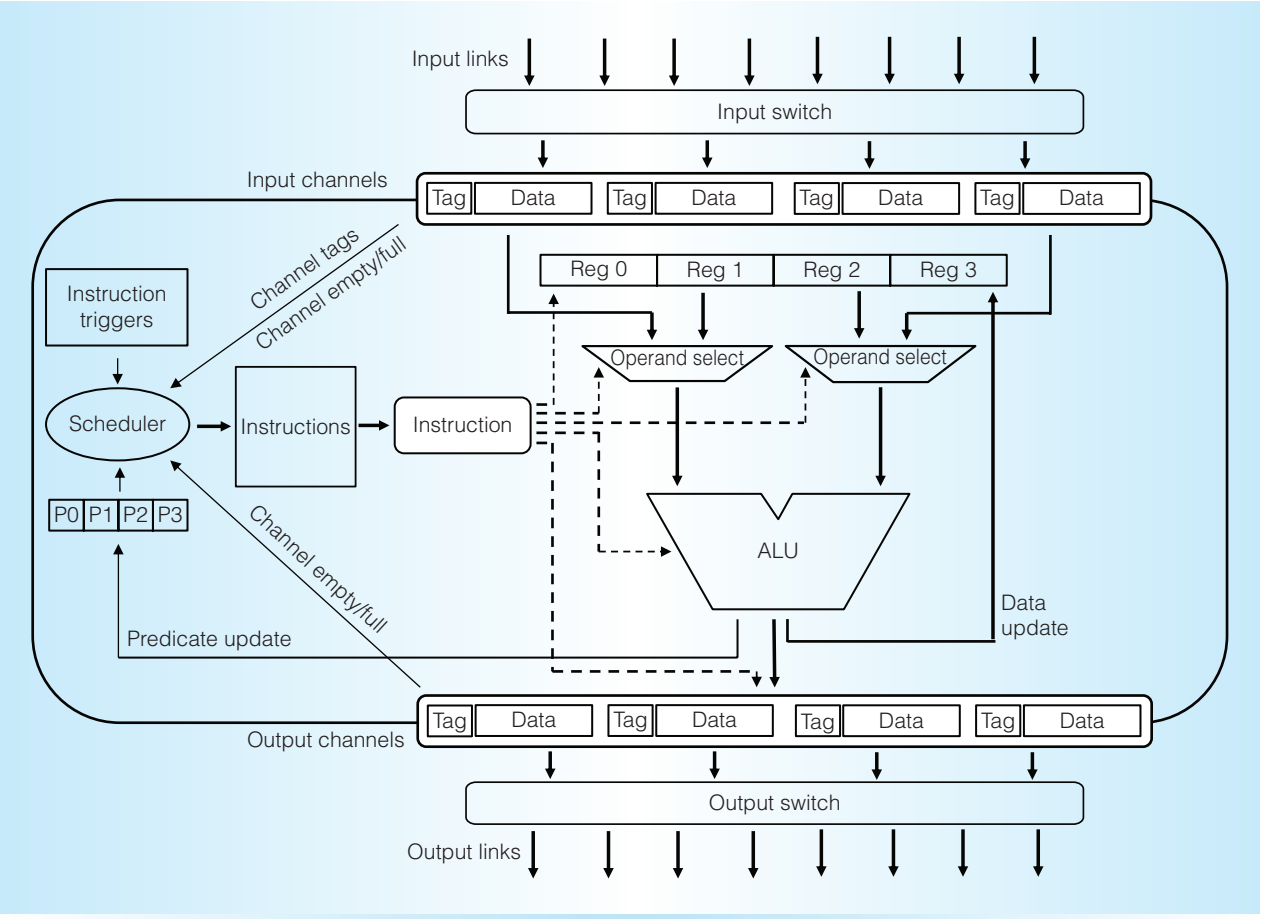


Figure 7. A PE based on our triggered-instruction architecture (TIA). The PE is preconfigured with a static set of instructions.

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 is simply a tree of AND gates. Thus, only the state equations that require reevaluation 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 doesn't consume any dynamic power whatsoever. The same control equations would have been evaluated using a chain of branches in a PC-based architecture.

Evaluation of workloads

In this section, we quantitatively demonstrate both the applicability of the spatial-programming approach to a set of workloads, and the efficiency that a triggered-instruction architecture provides within the spatial domain.

Approach

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 9 provides an illustration of a block and the parameters used in our evaluation. Each PE has the following architectural parameters:

- Datapath: 32 bits
- Sources per instruction: 2

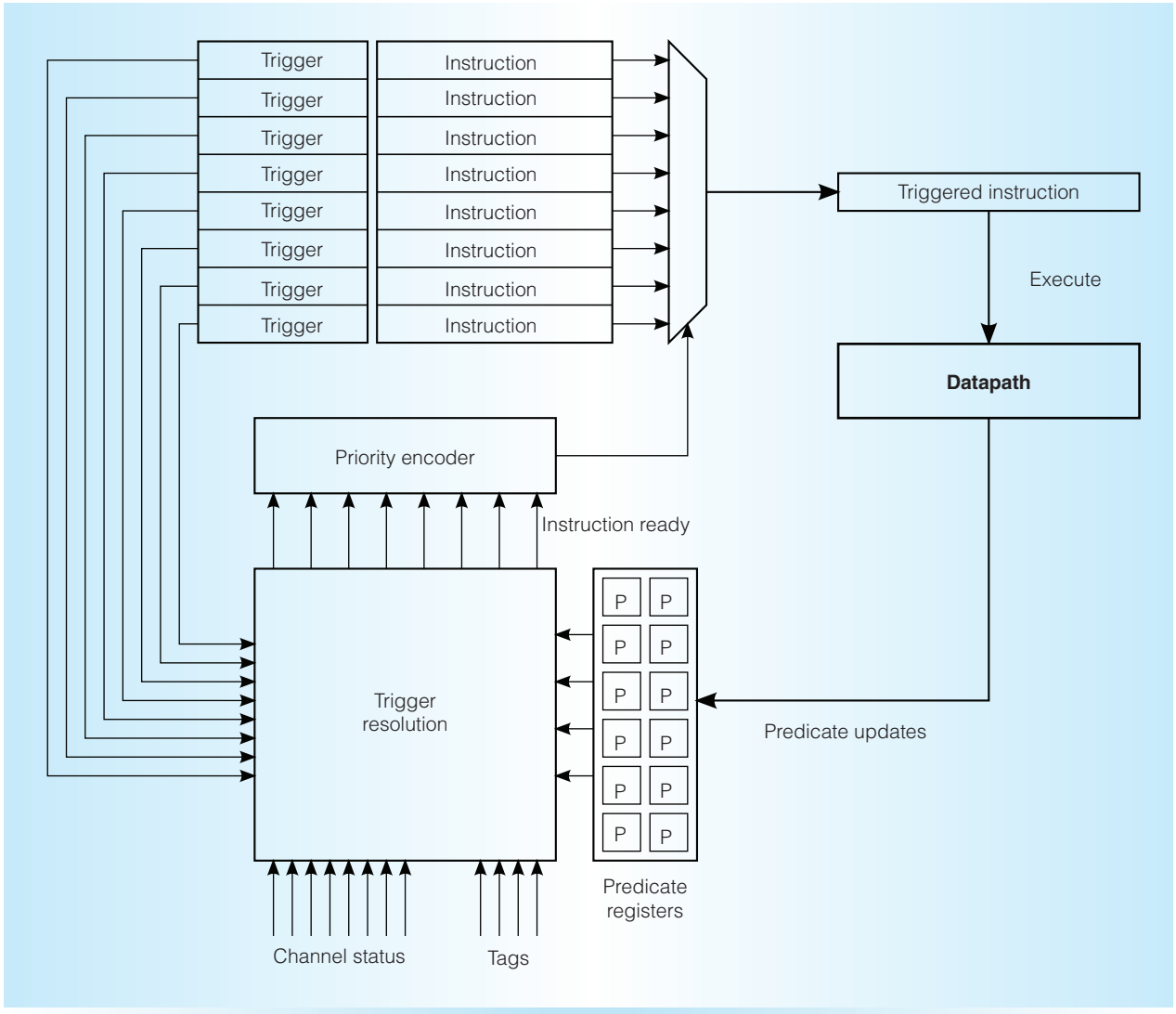


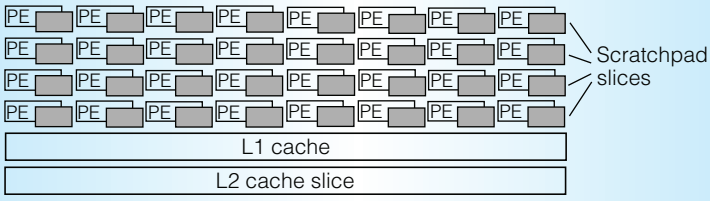
Figure 8. Microarchitecture of a TIA scheduler. The Trigger Resolution stage is implemented as combinational logic. This is a low-power approach because only local state updates and I/O channel activity consume dynamic power.

- Registers: 8
- Predicates: 8
- Maximum triggered instructions: 16

We obtained area estimates of each PE via the implementation feasibility analysis discussed in detail in our paper for the 2013 International Symposium on Computer Architecture.[14] 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, and so on) are about the same size as a single core of an Intel Core i7-2600 processor (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.[15] We model the detailed microarchitecture of each TIA PE in the array, the mesh interconnection network, the L1 and L2 caches, and the DRAM.

We evaluate our spatial fabric on application kernels from several domains. We do this under the assumption that the workload's computationally intensive portions will be



(a)

PEs: 32
Network: Mesh (one-cycle link latency)
Scratchpad: 8 Kbytes (distributed)
L1 cache: 4 Kbytes (four banks, 1 Kbyte per bank)
L2 cache: 24-Kbyte shared slice
DRAM: 200-cycle latency
Estimated clock rate: 2 GHz

(b)

Figure 9. Data used in our evaluation. Block illustration (a) and parameters (b). 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.

offloaded from the main processor, which will handle peripheral tasks like setting up the memory and handling rare, but slow, cases.

Our quantitative evaluation has two objectives:

- to demonstrate the effectiveness of a TIA-based spatial architecture compared to a traditional high-performance sequential architecture, and
- 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.

For the first objective, we 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. As a baseline, we used sequential software implementations running on the 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 the second 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 earlier. 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 translates to practical improvements.

For our analysis, we chose workloads spanning data parallelism, pipeline parallelism, and graph parallelism. Table 2 presents an overview of the chosen kernels.

We implemented the triggered instruction versions of these kernels directly in our PE's assembly language and hand-mapped them spatially across our fabric. (In the future, we expect this to be done by automated tools from higher-level source code.)

Performance results

Figure 10 demonstrates the magnitude of performance improvement that can be achieved using a spatially programmed accelerator. Across our workloads, we observe area-normalized speedup ratios ranging from 3 times (fast Fourier transform) to about 22 times (SHA-256) compared to the traditional core's performance, with a geometric mean of 8 times.

Now let's 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 3 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 percent, with an average of 50 percent. These inner loops are all branchy and dynamic—far more than traditional sequential code.

This dynamism manifests itself as additional control cycles for both PC-based architectures. Figure 11 shows the dynamic execution cycles for all architectures broken down into cycles spent on operations in relevant categories. The cycle counts are all normalized to the number of data computation

Table 2. Target workloads for evaluation.

Workload	Berkeley Dwarf[16]	Domain	Comparison software implementations
Advanced Encryption Standard with cypher-block chaining (AES-CBC)	Combinational logic	Cryptography	Intel reference using AES-ISA extensions
Knuth-Morris-Pratt (KMP) string search	Finite state machines	Various	Nonpublic optimized implementation
Dense matrix multiply (DMM)	Dense linear algebra	Scientific computing	Intel Math Kernel Library (MKL) implementation[17]
Fast Fourier transform (FFT)	Spectral methods	Signal processing	FFT-W with auto-vectorization
Graph500-BFS	Graph traversal	Supercomputing	Nonpublic optimized implementation
k-means clustering	Dense linear algebra	Data mining	MineBench implementation with auto-vectorization
Merge sort	Map/reduce	Databases	Nonpublic optimized implementation
Flow classifier	Finite state machines	Networking	Nonpublic optimized implementation
SHA-256	Combinational logic	Cryptography	Intel reference (x86 assembly)

operations (D.ops) executed by PC+RegQueue. We augment this data with Figures 12 and 13, which respectively show the static and dynamic (average) instructions in the inner loops of rate-limiting steps for each workload. The data in these figures demonstrates that the triggered-instruction approach has measurable benefits over program counters in real-world kernels.

First, TIA demonstrates a significant reduction in dynamic instructions executed compared to both PC+RegQueue (64 percent) and PC+Augmented (28 percent) on average, and an average performance improvement of 2.0 times versus PC+RegQueue and 1.3 times versus 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 operations. TIA benefits from this, too, but gets a further performance boost over PC+Augmented from a reduction in Control operations and Predicated-False operations.

Second, an additional benefit of TIA over PC+Augmented comes from a reduction in wait cycles. This is most evident in the *k*-means (50 percent), Graph500 (100 percent), and SHA-256 (40 percent) workloads. This is because of the ability of triggered instructions to avoid unnecessary serialization. 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 overserialization.

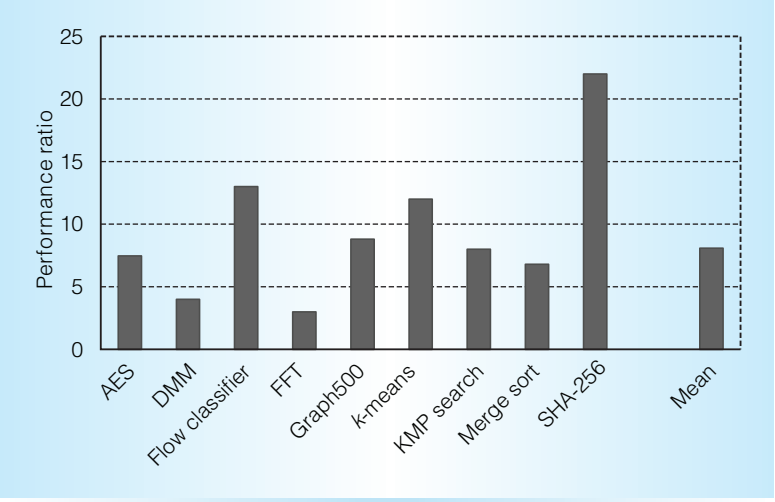


Figure 10. Area-normalized performance ratio of a TIA-based spatial accelerator compared to a high-performance out-of-order core. Area-normalized speedup ratios range from 3 times to about 22 times compared to the traditional core's performance.

Third, the workload that sees the largest benefit from triggered instructions is Merge Sort. Merge Sort has the highest dynamic

Table 3. Percentage of dynamic instructions that are branches in rate-limiting step inner loop.

Control scheme	AES (%)	DMM (%)	FFT (%)	Flow (%)	Classifier (%)	Graph-500 (%)	*k*-means (%)	KMP (%)	Search (%)	Merge sort (%)
PC+RegQ	58	50	36	50	50	69	8	70	63	50
PC+Aug	6	33	11	50	40	29	14	50	22	28

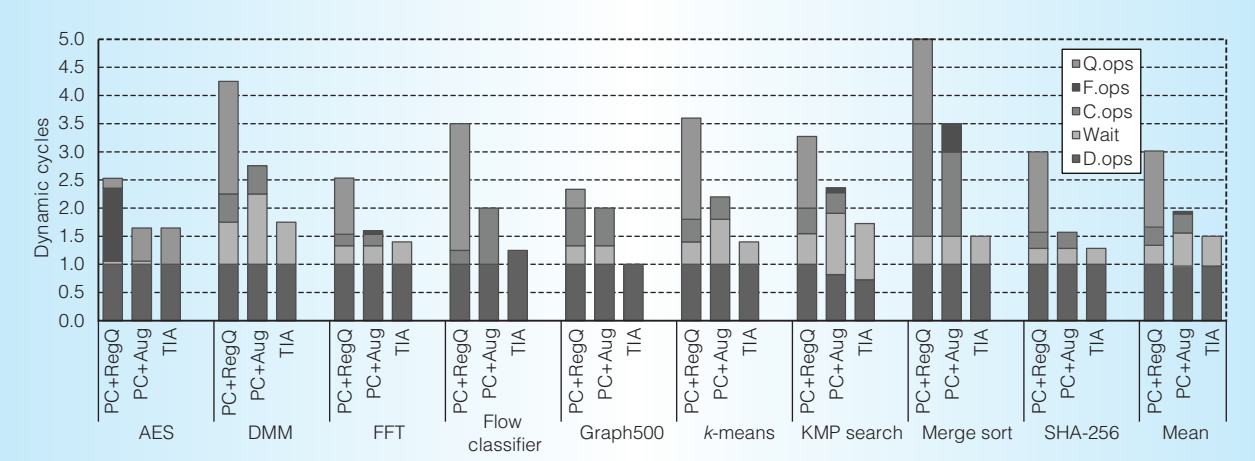


Figure 11. Breakdown of dynamic execution cycles in rate-limiting inner loops normalized to data computation operations (D.ops) executed by PC+RegQueue. This demonstrates the ability of triggered instructions to reduce queuing, control and predicated-false operations, and wait cycles arising from over-serialization.

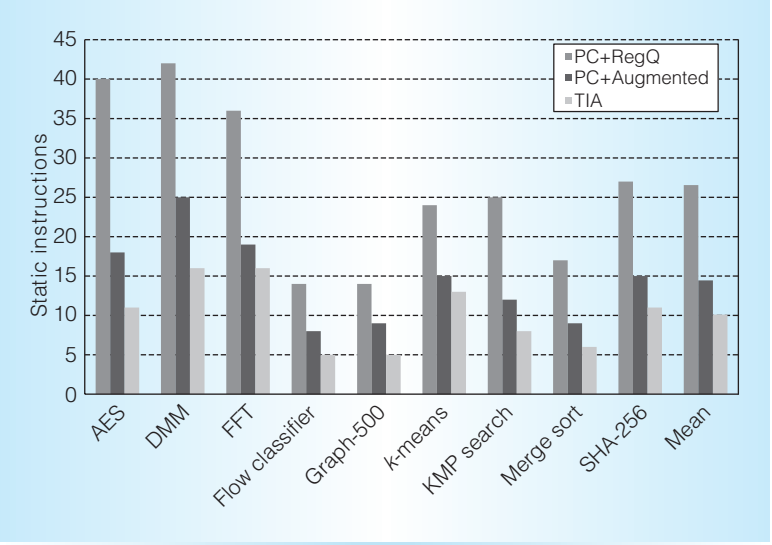


Figure 12. Static instruction counts for rate-limiting inner loops. See our previous work[14] for an analysis of why triggered instructions can never result in an increase in instruction count compared to PC-based approaches.

branch rate (70 percent) of all workloads on the PC+RegQueue architecture. It also spends several cycles polling queues. PC+Augmented eliminates all the queue-polling cycles, resulting in 1.6-times performance improvement in the rate-limiting step. TIA further cuts down a large number of control cycles, leading to a further 2.3-times performance improvement versus PC+Augmented and a cumulative 3.7-times performance benefit over PC+RegQueue.

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

Finally, triggered instructions use a substantially smaller static instruction footprint. The reduction in footprint compared to PC+RegQueue is particularly significant—62 percent on average. PC+Augmented's enhancements help reduce footprint, but

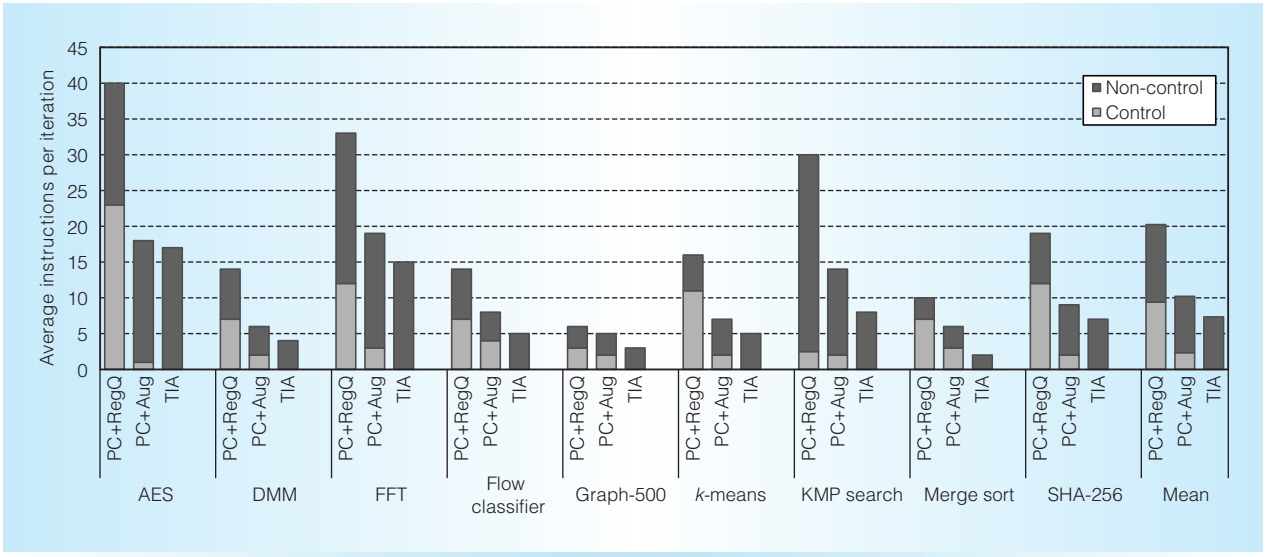


Figure 13. Average dynamic instruction counts for rate-limiting inner loops. In this context, removal of instructions can directly translate into workload speedup.

TIA still has 30 percent 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.

Our 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 several 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. MICRO

References

1. E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1996, pp. 157-166.
2. J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1997, pp. 12-21.
3. B. Mei et al., "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," *Proc. 13th Int'l Conf. Field-Programmable Logic and Applications*, 2003, pp. 61-70.
4. D. Burger et al., "Scaling to the End of Silicon with EDGE Architectures," *Computer*, vol. 37, no. 7, 2004, pp. 44-55.
5. V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for Energy Efficient Computing," *Proc. 17th Int'l Conf. High Performance Computer Architecture* (HPCA), 2011, pp. 503-514.
6. S. Swanson et al., "The WaveScalar Architecture," *ACM Trans. Computer Systems*, vol. 25, no. 2, 2007, pp. 4:1-4:54.
7. M. Taylor et al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, 2002, pp. 25-35.
8. Z. Yu et al., "An Asynchronous Array of Simple Processors for DSP Applications," *Proc. Solid-State Circuits Conf.*, 2006, pp. 1696-1705.
9. G. Panesar et al., "Deterministic Parallel Processing," *Int'l J. Parallel Programming*, vol. 34, no. 4, 2006, pp. 323-341.

10. D.G. Merrill and A.S. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques*, 2010, pp. 545-546.
11. E.W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Comm. ACM*, vol. 18, no. 8, 1975, pp. 453-457.
12. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
13. Bluespec, *Bluespec SystemVerilog Reference Guide*, 2007.
14. A. Parashar et al., "Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures," *Proc. Int'l Symp. Computer Architecture*, 2013, pp. 142-153.
15. J. Emer et al., "Asim: A Performance Model Framework," *Computer*, vol. 35, no. 2, 2002, pp. 68-76.
16. K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, tech. report UCB/EECS-2006-183, Electrical Eng. and Computer Science Dept., Univ. California, Berkeley, Dec. 2006.
17. R.A. van de Geijin and J. Watts, *SUMMA: Scalable Universal Matrix Multiplication Algorithm*, tech. report, TR-95-13, Dept. of Computer Sciences, Univ. of Texas at Austin, 1995.

Angshuman Parashar is an architecture research engineer in the VSSAD group at Intel. His research interests include spatial architectures, hardware/software interfaces, and quantitative evaluation of computer systems. Parashar has a PhD in computer science and engineering from the Pennsylvania State University.

Michael Pellauer is an architecture research engineer in the VSSAD group at Intel. His research interests include spatial architectures and high-level hardware description languages. Pellauer has a PhD in computer science from the Massachusetts Institute of Technology.

Michael Adler is a principal engineer in the VSSAD group at Intel. His research interests include building flexible microarchitecture timing models of large systems using FPGAs and building OS-like services to simplify FPGA programming. Adler has a BA in philosophy from the University of Pennsylvania.

Bushra Ahsan is a component design engineer at Intel. Her research focuses on memory systems architecture design and workloads for spatial architectures. Ahsan has a PhD in electrical and computer engineering from the City University of New York.

Neal Crago is an architecture research engineer in the VSSAD group at Intel. His research interests include spatial and energy-efficient architectures. Crago has a PhD in computer engineering from the University of Illinois at Urbana-Champaign.

Daniel Lustig is a PhD candidate in the Department of Electrical Engineering at Princeton University. His research focuses on the design and verification of memory systems for heterogeneous computing platforms. Lustig has an MA in electrical engineering from Princeton University.

Vladimir Pavlov is a senior software engineer at Intel. His research focuses on programming and exploration tools for novel programmable accelerators, such as application-specific instruction set processors and spatial architectures. Pavlov has an MS from the State University of Aerospace Instrumentation, Saint Petersburg, Russia.

Antonia Zhai is an associate professor in the Department of Computer Science and Engineering at the University of Minnesota. Her research focuses on developing novel compiler optimizations and architecture features to improve both performance and nonperformance features, such as programmability, security, testability, and reliability. Zhai has a PhD in computer science from Carnegie Mellon University.

Mohit Gambhir is an architecture modeling engineer in the VSSAD group at Intel. His research interests include modeling and simulation, performance analysis, and SoC architectures. Gambhir has an MS in

computer science from the North Carolina State University.

Aamer Jaleel is a principal engineer in the VSSAD group at Intel. His research interests include memory system optimizations, application scheduling, and performance modeling. Jaleel has a PhD in electrical engineering from the University of Maryland, College Park.

Randy Allmon is a senior principal engineer in the VSSAD group at Intel. His research interests include low-power, high-performance circuit and layout design and soft-error mitigation research. Allmon has a BS in electrical engineering from the University of Cincinnati.

Rachid Rayess is a silicon architecture engineer in the MMDC group at Intel. His research focuses on memory architecture and memory design automation. Rayess has an MS in electrical engineering from North Carolina State University.

Stephen Maresh is a performance modeling engineer at Intel. His research focuses on fabrics, spatial architectures, cache circuit design, and microprocessor design integration. Maresh has an MS in electrical and computer engineering from Northeastern University.

Joel Emer is an Intel Fellow and director of microarchitecture research at Intel, where he leads the VSSAD group. He is also a professor of the practice at the Massachusetts Institute of Technology. His research interests include spatial architectures, performance modeling, and memory hierarchies. Emer has a PhD in electrical engineering from the University of Illinois. He is a Fellow of IEEE.

Direct questions and comments about this article to Michael Pellauer, Intel, 77 Reed Road, MS HD2-330, Hudson, MA 01749; michael.i.pellauer@intel.com.