Pipelining a Triggered Processing Element

Thomas J. Repetti
Dept. of Computer Science, Columbia University
trepetti@cs.columbia.edu

João P. Cerqueira
Dept. of Electrical Engineering, Columbia University
jd3137@columbia.edu

Martha A. Kim
Dept. of Computer Science, Columbia University
martha@cs.columbia.edu

Mingoo Seok
Dept. of Electrical Engineering, Columbia University
mgseok@ee.columbia.edu

ABSTRACT

Programmable spatial architectures composed of ensembles of autonomous fixed-ISA processing elements offer a compelling design point between the flexibility of an FPGA and the compute density of a GPU or shared-memory many-core. The design regularity of spatial architectures demands examination of the processing element microarchitecture early in the design process to optimize overall efficiency.

This paper considers the microarchitectural issues surrounding pipelining a spatial processing element with triggered-instruction control. We propose two new techniques to mitigate pipeline hazards particular to spatial accelerators and non-program-counter architectures, evaluating them using in-vivo performance counters from an FPGA prototype coupled with a rigorous VLSI power and timing estimation methodology. We consider the effect of modern, post-Dennard-scaling CMOS technology on the energy-delay tradeoffs and identify a set of microarchitectures optimal for both high-performance and low-power application settings. Our analysis reveals the effectiveness of our hazard mitigation techniques as well as the range of microarchitectures designers might consider when selecting a processing element for triggered spatial accelerators.

CCS CONCEPTS

• **Computer systems organization** → **Pipeline computing**; *Reduced instruction set computing*; *Multiple instruction, multiple data*; *Multicore architectures*; *Interconnection architectures*; • **Hardware** → *Power and energy*;

KEYWORDS

Spatial architectures, pipeline hazards, microarchitecture, design-space exploration, low-power design

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124551

ACM Reference format:
Thomas J. Repetti, João P. Cerqueira, Martha A. Kim, and Mingoo Seok. 2017. Pipelining a Triggered Processing Element. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124551

1 INTRODUCTION

Spatial accelerators support important workloads such as information retrieval [22], databases [33–35], string processing [26], and neural networks [8, 9]. A general purpose spatial array of programmable processing elements can serve these and other applications with spatial parallelism and direct inter-processing element communication. In contrast to a fixed-function accelerator, a programmable accelerator accommodates new workloads and optimization of existing ones. In such designs, triggered control [19, 20, 32] has demonstrable architectural benefits, reducing both dynamic and static instruction counts relative to program counter-based control. This is an important element of reducing energy and delay, but it is only part of the story.

Total energy consumption is a product of dynamic instruction count and the energy expended per instruction:

$$\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Energy}}{\text{Instruction}}.$$

While the architecture and workload determine $\frac{\text{Instructions}}{\text{Program}}$, the microarchitecture and circuit determine $\frac{\text{Energy}}{\text{Instruction}}$. Total delay is likewise determined by the archictectural constraint of dynamic instruction count but also by cycles per instruction (CPI) and cycles per unit time (frequency):

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}.$$

$\frac{\text{Cycles}}{\text{Instruction}}$ and $\frac{\text{Time}}{\text{Cycle}}$ are also properties of the microarchitecture and underlying circuit technology. Exploring these elements of energy and delay below the architectural level is the focus of this paper.

This work investigates the microarchitectural and circuit design space of triggered processing elements. The replication factor in a tiled architecture demands deep analysis and optimization of the central building block: the processing element (or PE). We focus on the interplay between the instruction pipeline and supply voltage scaling, as it has been long established that pipelining can improve instruction level parallelism, timing closure, and power efficiency through voltage scaling [2, 3, 16, 27]. Once a pipeline has reduced the critical path of a circuit, additional opportunity to trade

energy and delay appears. One could maintain nominal supply voltage and increase clock frequency, maintain the original clock frequency and reduce supply voltage, or apply some combination in the middle. Our results reveal the importance of such design choices. Having explored over 4,000 unique designs in this space, the energy-delay tradeoff curve spans 71x in energy – from 0.67 to 47.59 pJ / instruction – and 225x in delay – from 1.37 to 309.03 ns / instruction.

Triggered control poses unique sorts of hazards for an instruction pipeline. To launch an instruction, the front end must compare the predicate and communication queue state to a programmed set of trigger conditions, as opposed to just calculating the next address for the program counter. We present two new hazard mitigation techniques that help keep the pipeline full: speculation on upcoming predicate state and accurate queue status accounting given the current contents of the pipeline. We find that these techniques reduce the increases in CPI that otherwise accompany deep pipelines, together reducing CPI in a 4-stage pipeline by 35%. They incur some overheads – in the worst case 1.4% area, 8% power, 20% critical path – but ultimately improve the optimal design frontier by 20-25% in both energy and delay. While predicate prediction is applicable specifically to triggered instruction architectures, the method of determining effective queue status benefits any spatial architecture with pipelined processing elements and register queues.

We have released an open-source repository to support further investigation in this area. It includes SystemVerilog implementations of both the single cycle and pipelined microarchitectures presented here, which in turn can be used in synthesizable spatial arrays. It is supported by a toolchain that includes an assembler, functional ISA simulator, Linux driver and userspace library. All of these are governed by a single parameter file that configures the architecture (e.g., queue counts or instructions per processing element) and microarchitecture (e.g., turning on/off the aforementioned hazard mitigation techniques). Lastly, we include a set of ten triggered instruction microbenchmarks that exhibit a range of intra-PE behaviors.

In the following section we provide some more background on triggered architectures in general and the specific triggered ISA we have designed. Section 3 describes our FPGA-prototype and VLSI power and timing estimation methodology. Section 4 and Section 5 present and characterize the single cycle and pipelined designs respectively. Before closing, we discuss some limitations and possible extensions of this work in Section 6 and related work in Section 7.

2 TRIGGERED ARCHITECTURE

All of the microarchitectures we will examine are implementations of a triggered ISA of our own design. Here, we provide some background on triggered instruction control (Section 2.1), a description of our triggered ISA (Section 2.2), and the supporting toolchain we have released (Section 2.3).

2.1 Background

Triggered control was proposed by Parashar et al. in 2013 [19] as an alternative to program-counter-based control for spatial arrays of autonomous PEs. In the triggered scheme, each PE is programmed with a priority ordered list of guarded atomic actions. This list represents a finite, statically configured local pool of datapath instructions, whose eligibility for issue in any given cycle is determined by a corresponding "trigger" condition (i.e., the guard). Each cycle, all of the triggers are compared to designated architectural state – predicate and queue status, described shortly – to determine whether the corresponding instruction has been "triggered". Instructions are ordered by priority rather than sequence, with the highest priority triggered instruction issued for execution (Figure 2).

Each trigger-controlled PE is connected to neighboring PEs by a set of incoming and outgoing tagged data queues over an interconnect fabric. Tags encode programmable semantic information that accompanies the data communicated over these queues. For example, a tag might be used to indicate the datatype of the accompanying data word or a message to effect control flow like a termination condition. Tag values at the head of the input queues determine, in part, whether an instruction can be fired. The PE also contains a set of single-bit predicate registers, which can be updated immediately upon triggering an instruction, or as the result of a datapath operation. Each trigger's validity is determined by the state of the predicate registers, the availability of tagged input operands on the incoming queues, and capacity on the output queues for any instructions that write there. Comparison or logic instructions whose destination is a predicate register provide control flow equivalent to branching in program-counter-based ISAs.

By eliminating explicit branches, triggered control reduces the dynamic instruction count of a given task. Moreover, in a spatial context, it allows PEs to react quickly to incoming data. Together these two features help multiple PEs to work together in an efficient processing chain: each PE in the chain works on the current data item, and then efficiently hands it off to the next PE.

Parashar et al. evaluated a number of control idioms and laid out these architectural benefits in their original work [19]. The focus was primarily above the microarchitectural level, with mention of a two-stage pipeline not otherwise specified. The original paper examined the scheduler, placing it at 2% of PE area, but there was no other area or power breakdown or characterization. Subsequent work [32] mentions a CACTI and Verilog-based power model, suggesting it closed at 3.4 GHz in a unspecified commercial CMOS process, but that was the extent of the commentary. To date there has been little public study of the microarchitectural choices and challenges of implementing a triggered PE. Such study complements architectural ones, and, as this work shows, has a substantial impact on the ultimate energy efficiency performance that can be achieved.

Reducing the latency from when a PE receives a data token to when it emits a result token is an intra-PE optimization.

However, in a spatial context where PEs operate together as a pipeline, the throughput of the pipeline is limited by this single-PE latency. Improving the PE microarchitecture can thus have a system-level effect on the behavior of the entire fabric.

2.2 A Generic, Integer ISA

We have designed a triggered, general-purpose, RISC-style, integer ISA that supports a full complement of arithmetic and logical operations. In our assembly, an instruction looks like the following. Below, a merge sort worker looks for two available input queue operands with the tags 0 and performs an unsigned less than or equal to comparison on them, the result of which is written into seventh predicate register.

```
when %p == XXXX0000 with %i0.0, %i3.0:
    ult %p7, %i3, %i0; set %p = ZZZZ0001;
```

The first line is the guard and the second line is the datapath operation and predicate update. In this assembly, `%p` refers to the predicate register, which is part of the trigger condition and is optionally updatable by an assignment in the datapath operation segment, as shown in this example. Individual bits of the predicate can be pattern matched in the trigger condition and selectively assigned to with don't-care/high-impedance notation. The indexed `%r*`, `%i*`, `%o*` and `%p*` operands refer to general-purpose data registers, input channels, output channels and predicate bits, respectively. Tag information is referenced on input queues (in tag matching) and output queues (in datapath operations) with the `.` operator. In this snippet, the trigger is checking for tag values of `0` on input queues `%i0` and `%i3`. If any result is enqueued on an output queue, the tag must also be specified so downstream PEs can use the tag's semantic information.

In our ISA, we have restricted operations to fairly low-latency integer operations, with the lengthiest of these being two-word product integer multiplication. Although the current version of this architecture does not support floating point instructions, the need to pipeline such functional units would likely increase the need for techniques to fill deep triggered pipelines like those we present in Section 5. We do, however, offer a wide range of comparison operations and logical operators intended primarily for predicate writes to support expressive control flow. There is also a rich set of bit manipulation instructions, such as `clz` and `ctz` (count leading/trailing zeros) which support software implementation of routines such as integer division or floating point operations. The ISA also offers loads and stores from a small, PE-local scratchpad memory. Operations involving main memory are currently carried out explicitly via the queues using read and write ports as endpoints for designated channels as described in prior work [1].

The architecture exposes a number of parameters which govern the binary instruction encoding, listed in Table 1. These include standard ones such as register count and word size, as well as more unique ones such as the tag width for queue data and the maximum number of input queues that can be queried per trigger. Because our focus is on the microarchitecture, we have fixed all of the architectural parameters as listed in the table: a 32-bit data word size, 8 predicate registers, 8 data registers, 4 input channels, 4 output channels, a maximum of two input channel tag conditions per trigger, and 16 instructions per PE.

Parameter	Description	Value
$NRegs$	Number of registers	8
$NIQueues$	Number of input queues	4
$NOQueues$	Number of output queues	4
$MaxCheck$	Max queues checked per trigger	4
$MaxDeq$	Max dequeues allowed / ins	2
$NPreds$	Number of predicates	8
$Word$	Word width	32
$TagWidth$	Queue tag width	2
$NIns$	Number of instructions per PE	16
$NOps\star$	Number of operations	42
$NSrcs\star$	Number of source operands / ins	2
$NDsts\star$	Number of destinations / ins	1

Table 1: Architectural and microarchitectural parameters. All of them, except those marked with $\star$, are recognized by the toolchain described in Section 2.3.

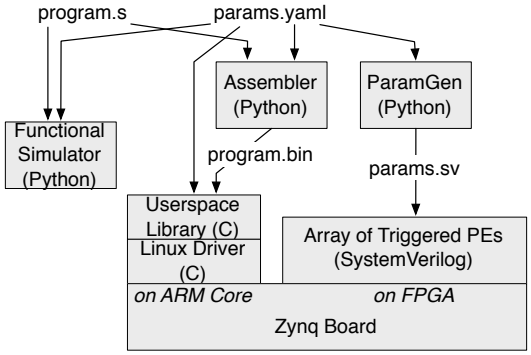

Figure 1: The toolchain is centered around a parameters file which can completely specify the target architecture and underlying microarchitecture to build a fully functional hardware/software system.

Table 2 describes each instruction field and its width, both in general and specifically as results from the parameter assignment in Table 1. These binary fields can be loosely divided between the trigger and datapath segment of the instruction.

Fields such as $PredMask$, which encode the requisite on-set and off-set of predicate register state necessary for an instruction to fire, clearly belong to the guard of the atomic action. While other fields like reference tags for comparison on input channels are also only needed for instruction scheduling, some others do not fit squarely into one half of the instruction or the other and must be available to the scheduler at all times.

For example, the destination information ($DstTypes$ and $DstIDs$) is required because if an operation is to write to an output queue, the instruction may trigger only if that queue has space. For instructions that do not enqueue information,

Field	Description	Width
Val	Valid bit	1
$PredMask$	Required on-set and off-set of predicates for trigger	$2 \times NPreds = 16$
$QueueIndices$	Input queues to check	$MaxCheck \times \lceil \log_2(NIQueues + 1) \rceil = 6$
$NotTags$	Which queues to check for *absence* of given tag	$MaxCheck = 2$
$TagVals$	Vector of tags to seek on input queues	$MaxCheck \times TagWidth = 4$
Op	Opcode	$\lceil \log_2(NOps) \rceil = 6$
$SrcTypes$	Source types (reg,input queue,immediate, or none)	$NSrcs \times 2 = 4$
$SrcIDs$	Source indices	$NSrcs \times \lceil \log_2(\max(Nregs, NIQueues)) \rceil = 6$
$DstTypes$	Destination types (register, output queue, or predicate)	$NDsts \times 2 = 2$
$DstIDs$	Destination indices	$NDsts \times \lceil \log_2(\max(NRegs, NOQueues, NPreds)) = 3$
$OutTag$	Tag with which to eneuque the result	$= TagWidth = 2$
$IQueueDeq$	Input queues to dequeue	$MaxDeq \times \lceil \log_2(NIQueues + 1) \rceil = 6$
$PredUpdate$	Masks of which predicates to force high or low	$2 \times NPreds = 16$
Imm	Immediate value	$WordWidth = 32$

Table 2: Instruction fields for our ISA encoding. The sizing of many fields in the machine code layout is dependent on the parametrization chosen in Table 1. Unsynthesized padding bits to round memory-mapped width are omitted.

these fields specify the destination data or predicate register are also required by the datapath. These two fields are then important to both the trigger resolution front-end of the processor and the datapath operation. A consequence of the parallel nature of trigger resolution is the need for all trigger fields to be exposed combinationally to the scheduler in order execution at a rate of one instruction per cycle, which has area and energy ramifications for instruction memory discussed at length in Section 4 and Section 5.4.

Another important component of the instruction is the *PredUpdate* field. These two bitvectors, with one entry per predicate, indicate which predicate bits to force high or low to update the predicate state. Updating the predicate state rapidly is important to keep the PE making forward progress. If any datapath instruction has a predicate as a destination, we assume that this predicate update mask will not conflict with it. This is guaranteed by the assembler provided in our toolchain. Since the predicate update is roughly equivalent to the default PC = PC + 4 update in an equivalent traditional machine, this field must update architectural state within a cycle of the instruction trigger in order to maintain an upperbound CPI of one.

We have opted to support full word-length immediate fields. Due to the small, fixed handful of instructions each PE can hold because of instruction storage medium constraints each instruction represents a scarce resource, and wasting two slots, and two cycles, to fill a register with a 32-bit physical address comprised of two ORed 16-bit immediates was undesirable. Moreover, the immediate is one of the few fields of the instruction that wholly belongs to the datapath, and therefore can reside in inexpensive RAM-based storage that is indexed after the triggers have been resolved.

2.3 Public Release

To foster research in this area, we have released a software toolchain for this instruction set, workloads, and the single-cycle and pipelined microarchitectures (Section 4 and Section 5). All of these will be available in a public repository at http://github.com/arcade-lab/tia-infrastructure.

Toolchain. The software toolchain is depicted in Figure 1. At its root is a single parameter file, recognized by the assembler, functional simulator, userspace library, and used to parametrize the microarchitecture itself. This flow recognizes all of the parameters listed in Table 1 with the exception of the starred ones. The assembler produces a binary that can be executed on either the functional simulator or an RTL prototype. The RTL prototypes – single-cycle PEs described in Section 4 and pipelined PEs as described in Section 5 implemented in SystemVerilog – are arranged in small-scale spatial arrays (maximum 4×4 to fit on a Zynq SoC-FPGA[11]) accompanied by a Linux-based driver and userspace library that accepts the same ISA parameter file as the assembler. The parameter file also supports a handful of on/off settings, making it easy to selectively enable features, such as wide multiplication and scratchpad use.

The userspace library is responsible for interacting with the Linux driver through a memory-mapped interface, through which it can manipulate control registers, program instruction memories, preload scratchpad memories, and read state from per-PE debug monitors and performance counters. In addition, it is responsible for performing all data I/O and setting up data buffers for program execution.

For simplicity of this interface we have padded each 106-bit instruction to a round 128 bits. This padding is never stored in the write-only instruction memory and exists solely to simplify manipulation of the instructions themselves by the host processor.

Workloads. The release includes a suite of hand written and optimized assembly programs designed to exhibit a range of behaviors within the PE. These programs – three of them running on a single PE, seven running on a two-by-two PE array – are summarized in Table 3. Some, such as the binary search tree are memory-access intensive, while others like the dot product computation are compute heavy. Others such as the merge worker and string search are branchy in data-dependent ways. Because we intentionally omitted a division instruction from the ISA, we have included a software division algorithm among the benchmarks, indicative of the software

support for otherwise omitted operations in a RISC-style ISA.

3 METHODOLOGY

The following sections describe and characterize the performance, area, and power consumption of a number of implementations of the ISA described in Section 2.2.

Using the toolchain described in Figure 1 and a Zynq SoC-FPGA test board, we gather event and cycle counts from performance counters embedded in each PE. All of the data in these runs is supplied from on-chip memory, which on this system has a load latency of four cycles. For multi-PE programs, performance counters are taken from the designated "worker" PE (described in the Table 3) which performs the representative part of the workload.

To assess the area, power, and clock frequency of each processing element, we synthesized netlists, minus performance counters, using Synopsys Design Compiler (2013.12-SP1) and TSMC 65 nm general-purpose CMOS standard cells. We used moderate synthesis optimization and allowed for retiming only within the multi-stage ALU and multiplier functional units in order to optimally pipeline arithmetic. We also allowed for clock-gating throughout the design since this is a simple dynamic power optimization that would be expected in an end-product. We characterized a subset of cells within the nominal 1.0 V standard cell library at a variety of supply voltages in the range of 0.4 V to 1.0 V to explore the interaction of pipelining and V_{DD} scaling.

For the design space exploration, we characterized this subset of the standard V_T standard cells at 0.6 V, 0.7 V, 0.8 V, 0.9 V, and 1.0 V, and target frequencies of 100 MHz to 1.5 GHz at 100 MHz granularity. We also selectively refined this frequency granularity in near-threshold regimes to a granularity of 50 MHz up through 500 MHz. For non-standard threshold voltage standard cells in the low- and high-V_T libraries, we performed characterization at 0.4 V, 0.6 V, 0.8 V, 1.0 V, and again used selective refinement in the near- and subthreshold voltage regimes. Here we additionally refined the subthreshold high V_T target frequency sweeps in increments of 10 MHz through 100 MHz. Eight pipeline microarchitectures with and without the two optional pipeline optimizations discussed in Section 5.2 and Section 5.3 came to a total of 32 different microarchitectures. The resulting design space spans over 4,000 different design points.

For each of them, we extracted gate-level activity factors from a run of the binary search tree program, whose behavior we compared against known FPGA performance figures for automated post-synthesis validation. Among the single-PE workloads, binary search tree had the most balanced combination of I/O channel use, computation and memory access delay. The resulting execution ranged from approximately 90,000 to 160,000 cycles depending on the microarchitecture being analyzed. The resulting fully annotated netlist-level VCD was used as a dynamic activity input into the Synopsys PrimeTime (2013.12-SP1) fine-grain power and timing modeling software. We report pre-extraction results, but use a uniform wire-load model of 2 fF for all local nets. Due to the size of the design space, it was impractical to perform place and route on all the designs, which is why we opted for this uniform capacitive wire load approximation, but we have also successfully pushed several of our designs through automatic place and route to demonstrate feasibility.

For the other microbenchmarks, dynamic instruction counts vary from 20,003 for `dot_product` to 411,540 for `gcd`. The total number of cycles executed for any one benchmark on any given microarchitecture maxes out at approximately 700,000 cycles.

4 SINGLE-CYCLE BASELINE

Figure 2 depicts our single-cycle baseline microarchitecture, which serves as a qualitative baseline on which we build the pipelined implementations. While the primary design objective was simplicity, it includes a handful of features aimed at efficiency. The front end of the processor centers around the scheduler, which as described in the original triggered architecture paper [19], is responsible for comparing the instruction triggers with the current predicate and queue state, to determine which instructions are eligible to execute. A priority encoder selects one, which is then fetched from instruction memory and executed on a fairly standard RISC datapath. Predicate updates from the triggered instruction are applied, with potentially additional conditional predicate updates originating from ALU comparison or logic operations.

As mentioned in our discussion of binary instruction encoding in Section 2.2, a peculiar aspect of triggered-instruction-based control is the need for the portion of the instruction that includes trigger information to be accessed in parallel. This precludes all synchronous RAM-based storage media for storing trigger information, making it possible for the instruction storage to become large and power hungry relative to other components. Since some instruction fields are not required by the Scheduler (e.g., Imm), it is possible to store those in a traditional indexed store such as SRAM, so long as the design is pipelined such that the stage in which instructions are triggered is separate from the stage in which those fields are decoded. We have validated that this mixed storage medium microarchitecture is possible on an FPGA prototype. In our preliminary CACTI-based [29] analysis of mixed register/latch-SRAM, we find that we can reduce instruction memory area and power usage by 16% and 24%, respectively, over register-only instruction memory and by 9% and 19%, respectively, over latch-only instruction memory. Because of the restrictions such a design would impose on the pipelines we wish to study, however, we have opted to forgo SRAM and place the entire instruction in the "trigger" storage.

Using clock-gated registers, the instruction storage accounts for 25% of PE area and 41% of PE power. The high area and power use are at least partially attributable to sized-up instruction register transistors, which are both timing critical and relatively high-fan-out. Latches reduce the area by just over 30% and power by 75% thanks to the removal of

`bst`	A single PE accesses memory to traverse a binary search tree with nodes generated with random numbers to increase branch (predicate datapath write) entropy. The PE then stores the Boolean result of this search in the same data memory.
`gcd`	A single PE reads two numbers for which to calculate the GCD (chosen intentionally for long runtime), and performs a register-register operation workload to calculate the GCD before storing it back to memory.
`mean`	A single PE reads an array of numbers from memory and accumulates them before calculating their average and storing it back to memory.
`arg_max`	One PE streams an array of integers from memory to another which determines the index of the highest of these values. The second PE (the worker) then stores the result back to data memory.
`dot_product`	Two PEs stream two integer arrays to a third PE (the worker) which calculates the dot product. Upon receiving end-of-program tags from both stream PEs, the multiply-accumulate PE saves its accumulator to memory before halting.
`filter`	One PE streams a list of integers to a second which determines whether they are above a threshold and in turn emits a zero of one accordingly to a third PE. This third PE (the worker) uses this Boolean input stream to determine whether to save the corresponding value from a second stream of integers to memory.
`merge`	Simulated the conditions for a PE in a high-radix spatial merge sort using a 2x2 array of PEs. Two PEs stream sorted lists to a merge PE (the worker), which must produce a sorted list combining them.
`stream`	One PE (the worker) generates a stream of data to store (increasing integers from zero to a maximum value) while a second produces an identical stream which is used as store indices. The goal of the benchmark is to determine the maximum throughput for a sequential loop within a PE program.
`string_search`	One PE reads four-byte words from memory and forwards them to a second PE, which breaks these words into bytes. This second PE forwards those bytes to a third PE (the worker) which interprets each as an ASCII character. This third string matching PE scans the stream for the string `"MICRO"` using a small DFA hard-coded in TI assembly. This PE emits zeros in all states except the match state in which it emits a one, resulting in an output array in memory which gives the indices of these occurrences of `"MICRO"`.
`udiv`	This benchmark implements an unsigned integer division TI assembly macro in a single PE (the worker) which is then fed numerators and denominators by another PE streaming them from memory before storing the resulting quotients in memory. The divider PE saves these quotients back to memory to validate the result.

Table 3: PE-centric benchmarks designed to display a range of anticipated behaviors within a PE. Publicly released. All reported performance counter figures from multi-PE workloads come from the designated "worker" PE.

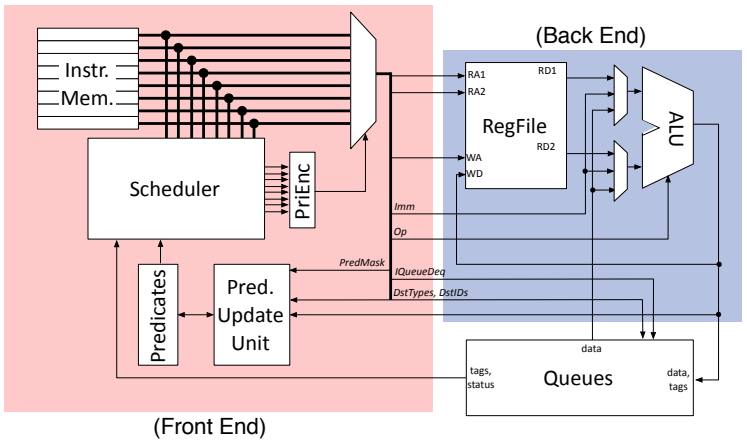

Figure 2: Block diagram of our single-cycle triggered PE. In order for programs to make forward progress using this control mechanism, predicate updates encoded in $PredMask$, any input channel dequeues in $IQueueDeq$ and datapath predicate writes must be atomic.

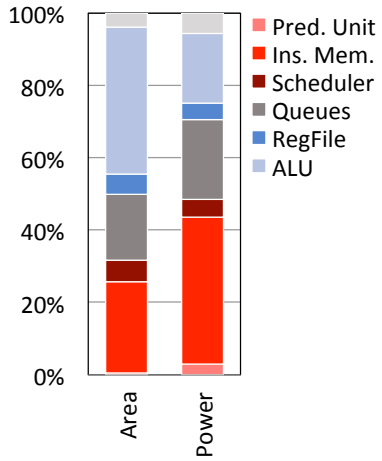

Figure 3: The baseline, single-cycle PE implementation consumes $64.435\,\mu m^2$ and $1.95\,mW$. The backend dominates area while the distribution is evenly split in terms of power.

clock tree capacitance and smaller cells. In the standard cell technology we use, however, we found that latches increased the critical path of the trigger resolver and the rate of failure in our gate-level post-synthesis validation. Thus, we settled on the clock-gated register-based instruction memory for both the baseline and all upcoming pipelined microarchitectures.

We use a detailed power and timing model through PrimeTime for sequential components and do not have an outside SRAM model for a scratchpad memory. Our benchmarks also

do not use local scratchpad memory, and so we omitted it from the single-cycle and later pipelined designs, although this feature is also functional in the FPGA prototype.

Area is dominated by ALU followed by instruction memory. Power is mostly proportionate to area, excepting the instruction memory which has relatively higher power and the ALU relatively lower power. This is at least partially due to the `bst` workload not exercising many of the larger functional components of the ALU like multipliers, but also due to the capacitance of the clock tree of the large sequential instruction memory. Our scheduler accounts for 6% area and 5% power, slightly higher than the 2% area consumption noted in Parashar's paper.

If you consider the queues, which account for 18% area and 22% power, to be neutral and break the remaining components down into front end (Predicate Unit, Instruction Memory, Scheduler) and back end (RegFile and ALU), the area split is 32% front v. 46% back end. For power this is reversed and slightly more skewed at 48% front end and 23% back end. This is a side effect of the simplicity of our datapath and instruction set. If it included more complicated operations and functional units, the relative power consumption of the front end would shrink. And the power would skew even further towards the backend if the alternative instruction stores described above were used and scratchpad memories were included.

5 PIPELINED MICROARCHITECTURE

The benefits to pipelining in terms of instruction-level parallelism, timing closure and increased power efficiency through voltage scaling are well understood and thus desirable in most microarchitectures. It is also well-known that pipelines present a wide variety of hazards necessitating control logic that can help mitigate them. Here we analyze the challenges that arise in triggered pipelines (Section 5.1), present two hazard mitigation strategies, predicate prediction (Section 5.2) and queue status accounting (Section 5.3), and evaluate them in the context of seven pipelines operating at a range of supply voltages (Section 5.4).

5.1 Control Challenges

A pipelined, triggered microarchitecture is susceptible to the same data hazards as a conventional pipeline, be they register operand dependences or dependences across stages of a pipelined functional unit. On top of this, triggered control introduces unconventional control hazards. With a PC, a processor must compute the next PC in order to fetch and begin executing the next instruction. The next PC is either the result of a predictable update function (e.g., PC = PC + 4) or a more complicated branch condition and target resolution calculation. By contrast, in a triggered architecture, program control advances via updates to the predicate and queue states.

To trigger the *next* instruction, one has two options: wait for the preceding instruction to complete its updates to the predicate and queue state, or speculate on the upcoming predicate and queue state and trigger a speculative instruction. The deeper the pipeline, the more critical it becomes to resolve or predict the upcoming states to keep the pipeline full. Interestingly, the dynamic rate of instructions that write to predicates in our workloads is 20%, almost exactly the rate of dynamic branches found in standard single-threaded workloads such as SPEC. In the next two sections, we present two techniques to anticipate the upcoming predicate and queue state to more quickly identify and begin executing the next instruction.

5.2 Predicate Prediction

The first technique is a speculative predicate unit, that is a drop in replacement for "Predicate Update Unit" depicted in Figure 2. The speculative version contains a two-bit saturating predictor for each predicate. For each instruction with a predicate destination, the speculative unit offers a predicted predicate value for that bit. When it does so, it also saves the original predicate state in the event of misspeculation and rollback.

Our scheme does not currently allow nested speculation, so predictions are made only if the system is not already speculating, or if the current speculation has been confirmed in the current cycle. During speculation, we restrict the instructions that can be issued. As in traditional processors, any instructions with side-effects prior to retirement cannot be issued speculatively as they could permanently alter architectural state. For this reason, dequeues are also forbidden since the dequeue of an input queue will take effect early during the execution of the associated instruction. Both of these restrictions lift, however, as soon as successful speculation is confirmed.

In the event of misspeculation, the pipeline is flushed and the predicate state visible outside of the speculative unit is returned to the fallback state. Because we have forbidden speculative dequeue operations, this simple predicate rollback is sufficient to restore the pre-speculation architectural state.

The term "predicate prediction" has been used in the context of out-of-order superscalar processors, describing a technique to improve the performance of `cmov`-like instructions [10]. However, in that context, predicated instructions are always fetched and issued, whereas the predicates in this context are central to the control mechanism, determining every instruction's execution eligibility.

5.3 Effective Queue Status

Queue hazards arise when an in-flight instruction might alter the state of an input or output queue, thus preventing the scheduler from identifying and launching the next instruction. Because queue state has a direct impact on both control flow and data movement, queue hazards straddle the traditional control/data hazard dichotomy. They resemble control hazards in that they effect instruction execution eligibility and data hazards in that the processor must prevent dequeues from empty inputs or enqueues to full outputs.

Such hazards can occur on any spatial architecture with operand queues, whether they have triggered control, PC-based control, or otherwise. Some designs such as MIT RAW have dealt with the queues going to and from the router in a binary full/empty fashion, conservatively treating all channels with pending enqueues as full or pending dequeues as empty [6, 30, 31]. Others like WaveScalar have padded the output queues with as many extra slots as the pipeline is deep, thereby guaranteeing queue capacity for in-flight instructions, calling this the "reject buffer" [23, 28].

We argue that such hazards may be dealt with more effectively and efficiently by pipeline register inspection and augmenting the instruction dispatch with some simple accounting. Accurate input queue status can be determined by exposing the current queue occupancy and calculating whether this value less the number of in-flight dequeues exceeds zero. Assuming that dequeues take effect within the first N pipe stages, this requires only a $\log_2(N)$-bit adder. In all of the pipelines we examine here N never exceeds 2 so this adder is small indeed. When queue tags are also scheduler inputs, it is further necessary to peek at the tag not just on the head of the input queue, but deeper in, again, according to the number of in-flight dequeues. More generally the first N tags on the input queue must be exposed, which for our pipelines is just the "head" and "neck".

Accurate output queues can be computed similarly and more simply, because the tag information of in-flight enqueues is irrelevant for scheduling. We still compute whether the current output queue occupancy, plus the number of in-flight enqueues for that queue, will fill the queue. For a pipeline of depth D, serving N output queues, this will require $N \times \log_2(D)$-bit adders plus a $\log_2(D)$-bit comparator. In contrast, padding the output queues, would require $D \times N$ additional queue entries, each one word wide plus tags.

Strictly speaking, this output accounting is still somewhat conservative as neighboring processing elements *may* dequeue from an output queue before the in-flight enqueues land. Even so, it is less conservative than the alternate approaches. Moreover, addressing the remaining conservative margin requires cross-PE coordination – between producer and consumer scheduler – which is an anathema in spatial design and would scale terribly as each scheduler must now coordinate with as many neighboring schedulers as there are output queues.

An efficient interface to determine the state of input and output queues is especially important here because of how spatial architectures encourage producer-consumer relationships between PEs. In the following section, our evaluation finds that the benefit of effective queue status accounting is substantial, even on our small-scale spatial workloads.

5.4 Evaluation

We have implemented these optimizations on seven different pipelines, ranging from 2 - 4 stages deep. Given the simplicity of the operations in our ISA, we found little value in deeper pipelines. Moreover, the critical path of these designs, ranging from 50 to 60 fan-out of 4 inverter delays (FO4) is in line with modern standards [12].

To describe the various pipelines, we have divided the work of a processing element into three conceptual stages:

- **trigger (T)** selects and fetches an instruction from memory,
- **decode (D)** retrieves any operands from the register files and input queues and performs the necessary operand forwarding, and
- **execute (X)**, optionally split into **X1** and **X2**, performs any arithmetic or logical operations and writes results to the register file or output queues.

We consider all possible pipelines that result from introducing pipeline registers between these stages: T|D|X, T|DX, etc. Including the single-cycle design, which we now refer to as TDX, this results in eight distinct microarchitectures.

Dequeueing from the inputs in the same cycle as the trigger resolution proved to be a long critical path, so we moved the dequeue operation to decode. In designs where the T and D stages are coalesced (TD*), this makes no difference, but it also opens up the possibility that they be split (T|D*). When they are split, and when the queue status optimization from Section 5.3 is enabled, the scheduler will inspect both the "head" and "neck" of the input queues.

Area and Power Overheads. The speculative predicate unit and effective queue status accounting are likely to incur their largest area and power overheads on deep pipelines. For these calculations we use the deepest pipeline (T|D|X1|X2) synthesized at nominal supply voltage using a standard V_t and a conservative target frequency of 500 MHz. While the pipeline can operate at higher frequency, the push for timing will inflate the resulting design. This baseline consumes 63 991.4 µm^2 and 2.852 mW. Adding a speculative predicate unit increases the area 0.5% to 64 278.4 µm^2 and the power by 7% to 3.048 mW. Adding the queue status accounting increases the baseline 2% to 64 131.8 µm^2 with no measurable difference in power consumption. For comparison, padding the output queues instead would have increased total device area 13% to 72 439.4 µm^2 and total power draw 12% to 3.194 mW. When combined, our two features increase the total PE area 1.4% to 64 895.4 µm^2 and total power consumption 8% to 3.077 mW.

As for the cost of pipelining itself, iso-frequency and iso-V_{DD} the power increases linearly with the addition of each pipeline register. Again at nominal voltage in a standard V_t standard cell technology at 500 MHz, we see an addition of 0.301 mW per pipeline register added. Due to the transistors sizing-up to meet timing in unpipelined (or under-pipelined) designs, any effect from the pipeline registers on area is negligible and lost in the noise.

Timing Overhead. We found that the simple circuitry to calculate effective queue status had no impact on timing closure. On the other hand, predicate speculation did slightly increase trigger stage delay. In a four-stage T|D|X1|X2 design at nominal voltage that closed at 1184 MHz without any

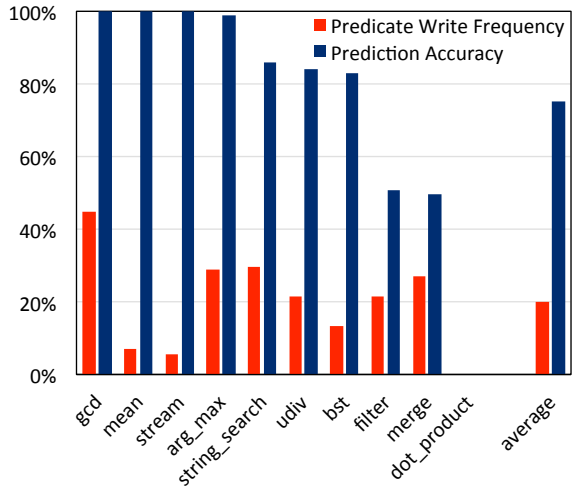

Figure 4: Datapath predicate write frequency and prediction accuracy by benchmark workload. Note that the worker PE in `dot_product` does not rely on predicates for control flow, just the semantic information encoded in operand tags.

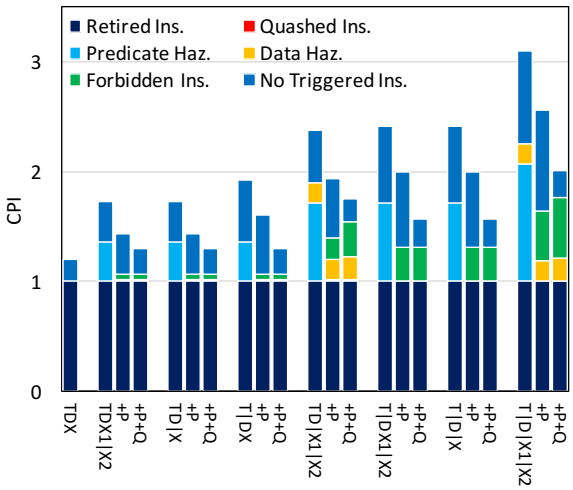

Figure 5: CPI stacks of various pipelined microarchitectures with the predicate prediction (+P) and effective queue status (+Q) pipeline optimizations selectively enabled. The stacks represent the average behavior over the ten workloads.

speculation, the critical path was the trigger stage with a delay of 53.6 FO4. Once speculation was enabled, this increased to 64.3 FO4.

This suggests the trigger stage largely sets the pipeline balance for any pipeline breakdown of this ISA, placing the balanced pipeline delay in the 50-60 FO4 range. This number is high relative to the optimal delays identified in the early 2000s ranging from 6 - 8 FO4 [17] to 18 FO4 [27]. However, CPUDB indicates a considerable upswing in average FO4 delay since then, with 2010-era delays ranging from 25 to 85 FO4 with an average of approximately 45 FO4 [12].

One consequence of this is that for simple integer functional units, as we shall see in our upcoming discussion of Pareto optimal designs, moderately deep pipelines may be sufficient in modern CMOS technology. This is a result that our early exploration of timing breakdown and pipeline balance using an FPGA failed to predict.

Predicate Prediction Accuracy. With the exception of the `dot_product` benchmark, all of our test programs have datapath predicate writes that activate the predictor. `filter` and `merge` both have data dependent control flow based on high-entropy input data generated prior to the test with a PRNG. These unpredictable control flow patterns represent our worst-case branch predictor accuracy of around 50%, seen in Figure 4. On the other hand, others such as `gcd`, `stream`, and `mean` all feature long-running and thus predictable loops as their main control structure and represent the best-case scenario of near perfect accuracy.

Still other benchmarks such as `bst` and `udiv` have unpredictable branches nested within predictable loop-like branches. In the case of `bst`, the predictable loop is the `while (next != NULL)` {} loop which is by definition always taken until the program exits with a result, and the unpredictable predicate write is from the result of the less-than comparison that determines which child to dereference. Similarly, in `udiv` the predictable predicate write is an iteration shifting through all the bits of the dividend, while the less predictable branch is whether the bit in question is one or zero.

A unique aspect of this scheme is that when a program assigns certain semantic significance to particular predicates, writing to them only to represent certain binary decisions, this bank of predictors becomes a per-branch predictor without the traditional overhead of indexing a bank of predictors via the instruction pointer. Our benchmarks were hand written with an awareness of this and generally assign a unique predicate for each different datapath predicate write.

Impact on CPI. Turning now to the microarchitectural dynamics of the pipelines, Figure 5 plots the CPI for each of the seven designs with no optimization, with predicate prediction turned on (+P), and with both predicate prediction and queue accounting turned on (+P+Q).

In the base pipelines, we observe that, rather intuitively, the rate of predicate hazards increases as pipeline depth increases. In fact, the impact of predicate hazards is the same for all pipelines of a specific depth. Moreover, their impact is superlinear in the length of the pipe, adding 0.18 CPI at 2-stage pipes, 0.24 CPI at 3-stages, and 0.27 at 4-stages.

When the speculative predicate unit is introduced, the predicate hazards are eliminated almost entirely, with virtually no quashed instructions. Considering the accuracies we found in Figure 4 for the overwhelming majority of the benchmarks averaged in the CPI stacks, this is unsurprising.

However, with speculative execution, we also see an uptick in forbidden instructions, namely those with pre-retirement side effects described in Section 5.2 that are forbidden during speculation. In this situation, the scheduler recognizes them as ready to execute, but they are prevented from issue because the current speculation has not yet been confirmed. As with the predicate hazards, we observe that the rate of forbidden instructions increases with pipeline depth and the correspondingly lengthened speculative windows. This is exacerbated by the fact that many of our benchmarks feature nested loop control and dequeues embedded within loops.

When the queue status accounting is added, we see a drop in cycles with no triggered instruction. This is the expected effect of a mechanism that is intended to make the scheduler trigger instructions less conservatively. Interestingly however, the cycles with no triggered instructions drops to a pipeline-depth-agnostic constant equal to that found in the single-cycle design. Whereas this component of CPI had been increasing with pipeline depth, the queue accounting erases this unpleasant side effect of pipelining entirely.

Energy Delay Analysis. To achieve optimal results in the face of energy-delay tradeoffs, we explored supply voltage as a first-class design parameter. As opposed to post-synthesis exploration looking at a design's behavior under a DVFS scheme, here we can take advantage of having a specific target frequency and voltage in mind when pushing our design through the VLSI flow, which results in optimally sized cells for the intended use case.

We scaled voltage and frequency considerably, and further considered the tradeoff between leakage power dissipation and gate switching delay that can be tuned by selecting an appropriate threshold voltage for the underlying circuit technology. Many of the optimal points we see on the tradeoff curve in Figure 6 can only be realized through this form of exploration conducted at this scale. As expected, the upper-end of the performance spectrum is dominated by low V_T standard-cell designs, the middle by standard V_T, and the low-power and ultra-low-power domains by high V_T designs. Despite all of this, the energy-delay span of a single architectural design point was remarkably broad, stretching 71x in energy and 225x in delay.

It is over this range that we see the benefits of our proposed pipeline optimizations. As indicated in Figure 7, any negative effects on timing closure brought about by using the speculative approach are largely erased by improvements to CPI. Near the balanced region closest to the origin of the Pareto curve, the addition of both the effective queue status accounting and predicate speculation improves the frontier by 20-25% in both energy and delay.

The benefit of combining speculation with queue accounting is not uniform, however. Though it has a particularly strong benefit to pipelined designs and dominates virtually all other pipelines in the balanced to moderately low-power regions of the frontier, towards the high performance extreme of the Pareto set, it appears that determining effective queue status alone is optimal. It may be that in those conditions, the critical path overhead incurred by speculating in the trigger stage is not worth the cost of losing on timing closure in high performance designs. That said, the two microarchitectural knobs offer clear benefits – together in ultra low power and moderate cases and in queue status alone in high performance.

Pareto Optimal Designs. Figure 8 details the designs on the Pareto frontier. Through many of the low power regimes into the energy-delay balanced region of the frontier, the single-cycle TDX baseline remains surprisingly competitive due at least partially to its superior CPI. The curve traced out by T|DX designs with prediction and queue status accounting enabled, however, narrowly dominates most of them.

Another surprising observation is that the exact same microarchitecture provides the best global performance and best absolute energy efficiency. At the high performance extreme, a two stage pipeline with a split-stage ALU, TDX1|X2, synthesized in a low V_T standard cell library using the effective channel status optimization and running at 1157 MHz achieves the highest throughput with an instruction latency of 1.37 ns, but does so at a cost of 21.42 pJ/instruction. At the low-power extreme this exact same microarchitecture synthesized in a high V_T standard cell library also proves globally optimal achieving 0.89 pJ/instruction. It seems that this correlation between high performance and low-power microarchitectures is not without precedent, though, as it has been reported in other literature that high performance designs can counter-intuitively share characteristics with ultra-low-power microarchitectures [2].

Only a single three-stage pipeline appears on the frontier, here as the second most performant design. It in fact has nearly identical throughput compared to the highest performing design, with an instruction latency of 1.43 ns, but consumes nearly half the energy per instruction of the highest-performance design with 11.91 pJ/instruction. This T|DX1|X2 pipeline also uses both of the proposed optimizations with prediction and queue status accounting enabled in order to achieve this result.

Power Density. Power density is of central importance in the design of massively parallel architectures. The density of power dissipation and thus the thermal design power (TDP) of GPUs is one of the primary factors that keeps their clock frequencies often twice as slow as CPUs in comparable technologies. When selecting a processing element design, power density will likely be one factor that limits the scalability of the spatial architecture.

Our Pareto-optimal designs show remarkably little variance in area, but as discussed above show considerable variance in power characteristics. With this in mind, it is possible to see where these designs fit in the context of conventional mainstream architectures like CPUs and GPUs in terms of their power density.

In the 65 nm era, CPUDB shows a mean power density of approximately 500 mW/mm^2 for CPUs with a maximum of 1000 mW/mm^2 and a minimum of 50 mW/mm^2 [12]. At

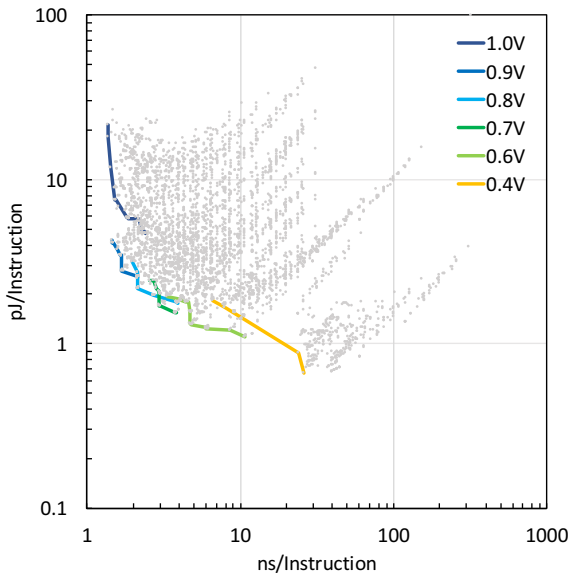

Figure 6: Frontiers for each supply voltage considered within the design space.

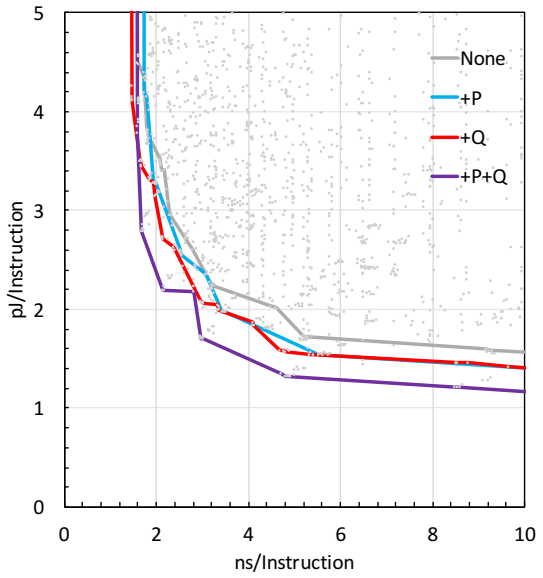

Figure 7: The benefit of adding predicate prediction (+P) and queue status accounting (+Q) in balanced designs at the Pareto frontier of the main energy delay tradeoff.

the same node, GPUs had a maximum power density of approximately 300 mW/mm^2 [7].

All of the PEs on the Pareto frontier fall below these CPU and GPU densities with the highest power density design consuming 167.6 mW/mm^2, although that is likely to increase substantially post-extraction, despite our uniform wire load model. It would also likely increase had the test workload post synthesis been one that used more power-hungry arithmetic like multiplication instead of primarily comparisons and additions.

6 LIMITATIONS AND EXTENSIONS

Since this is a study of pipelined microarchitectures, it would have been interesting to add functional units like floating point that demand deeper pipelines, as much of our timing analysis indicates that for simple integer functional units like the one examined here, moderate pipelines in the range of two to three stages are optimal. An analysis of such a system, we believe, may have also exposed limitations in our speculation scheme. The fact that we do not support nested speculation – which already showed evidence of hurting CPI in Figure 5 – would have likely hurt even more in deeper pipelines with more instructions forbidden from entering the pipeline due to the nesting restriction. Our initial exploration suggests that it would not be terribly expensive to support nested speculation, and we would like to examine the effect of this addition on decreasing the number of forbidden instructions in deep pipelines.

This study focused squarely on intra-PE microarchitecture and therefore did not consider the effect of memory access in

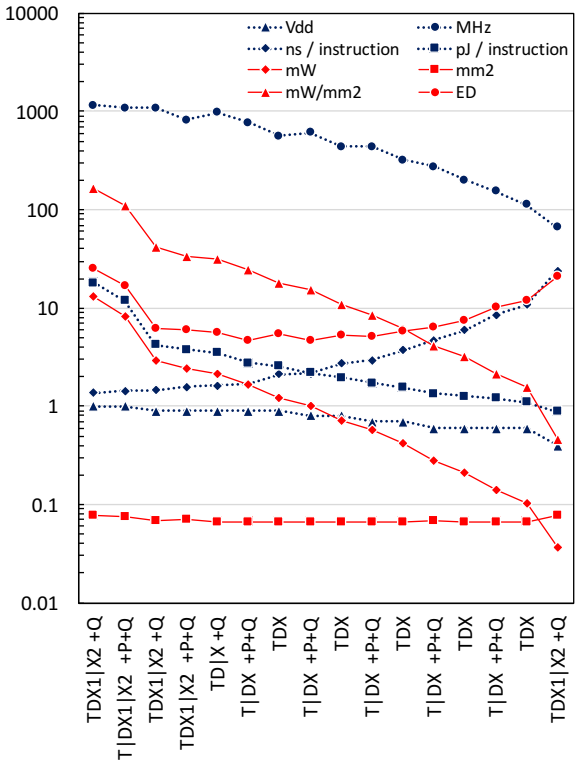

Figure 8: Parametric analysis of Pareto optimal designs.

great detail. Since the FPGA testbed and VLSI testbench with fixed memory access latency via an on-chip local memory effectively emulates perfect caching, the interplay of a spatial architecture that can benefit from memory-level parallelism and a memory hierarchy was not considered. It remains to be seen if performing loads and stores over the mesh interconnect as in prior art [1] is optimal. We plan a future version of the ISA and system, not considered here, that will enable main memory access through per-PE load-store queues using the decoupled load access paradigm [18], as opposed to generating interconnect traffic. The effects of this addition on the PE pipelines are yet unknown.

As mentioned in Section 4, we did not use an SRAM library in our dynamic power analysis, and therefore have omitted the intended feature of per-PE scratchpad memories. The remaining control and datapath structures were all small and would have been placed in registers even had we used SRAMs elsewhere. An extended version of this work would include an analysis of using synchronous indexed storage for immediates or other datapath portions of the instructions. While it is unclear what the performance impact would be, our hypothesis is that area and power would increase due to the new SRAM scratchpad, but that power density would decrease overall due to its relatively lower activity level.

Lastly, our study of area and power shows that instruction memory is significant in the overall design budgeting, and not only can this be addressed through the storage used for the instruction memory itself but also by better encoding techniques. We believe that it should be possible to arrive at more efficient trigger and instruction encodings at the architectural level, alleviating some of the area and power pressure imposed by the unique parallel instruction memory access paradigm in the architecture.

7 RELATED WORK

General purpose programmable spatial architectures have been an area of active research for decades. Only a subset of these, however, have processing elements with sufficient execution control to fall into the category of locally autonomous spatial architectures. The MIT RAW architecture takes an in-order MIPS pipeline and extends it input and output register queues and a set of programmable routers in an on-chip network for spatial coordination between autonomous processing elements, and the microarchitectural implications of a MIPS pipeline in a spatial context is explored [30, 31]. It however uses a single Boolean signal for backpressure propagation on output ports and does not inspect its pipeline to determine effective queue occupancy. As a PC-based system with operand queues, it is one such architecture where our queue management approach would also apply.

The UCD AsAP DSP system [4] and more recent KiloCore chip [5] also are both examples of large-scale PC-based spatial architectures. Due to the deep SRAM I/O queues in AsAP, it may be using a padding approach to dealing with the in-flight enqueue problem for its PE pipelines. The GreenArrays GA144 chip [14, 15, 21] features a set of small PC-based MISC processing elements between which data is only moved from one immediate neighbor to another. It is notable among these because, while the AsAP chip is a globally asynchronous locally synchronous device, GreenArrays is an entirely asynchronous system. In addition to communicating data via these spatial ports, the GA144 can also moved instructions as well. While existing documentation discusses the uses for these input and output ports, the mechanisms by which backpressure is exerted and operand availability are established at the microarchitecture level is not detailed.

The WaveScalar and TRIPS architectures both feature processing elements with limited degrees of local autonomy and predication. TRIPS instruction memories are filled with windows of operand-availability triggered instructions that are in turn fetched via a global program counter [13, 25]. WaveScalar uses dataflow, rather than a PC, for control. While it explicitly describes its five-stage pipeline in detail as well as its queue management system as discussed in Section 5.3 [23], the TRIPS pipeline is not laid out in detail [25]. Although both WaveScalar and TRIPS use predication within their processing elements neither considered the possibility of predicting predicates as a pipeline optimization. A derivative of the TRIPS project, however, did use predicates for a different purpose: as a form of branch history when predicting the next hyperblock [24].

8 CONCLUSION

We have examined several microarchitectural concerns unique to pipelines within the processing elements of a triggered-instruction-based spatial architecture. Our two proposed hazard mitigation techniques prove successful across a variety of workloads in controlling CPI and show marked improvements to the energy-delay curve of a processing element. This enables either higher operating frequencies at iso-V_{DD}, lower supply voltages at iso-frequency, or deeply pipelined operations such as floating point or custom CISC-like ISA extensions.

We demonstrate the vast design space for combined circuit and microarchitectural techniques, examining the subtle interplay between microarchitecture, circuit optimization and application domain in choosing optimal PE designs. We find that the most optimal designs for our integer system are two-stage pipelines with our two pipeline optimization features enabled. Taken together, this work demonstrates the flexibility and criticality of intra-PE microarchitecture for spatial architectures.

9 ACKNOWLEDGEMENTS

This work is supported by C-FAR, one of the six SRC STARnet Centers sponsored by MARCO and DARPA, a grant from the National Science Foundation under CCF-1065338, and through funding from the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES) under grant 13289-13-6.

REFERENCES

[1] Bushra Ahsan, Michael C. Adler, Neal C. Crago, Joel S. Emer, Aamer Jaleel, Angshuman Parashar, and Michael I. Pellauer. 2013. Distributed Memory Operations. (Sept. 26 2013). reference: United States Patent App. 14/037,468.

[2] Massimo Alioto. 2012. Ultra-Low Power VLSI Circuit Design Demystified and Explained: A Tutorial. *IEEE Transactions on Circuits and Systems I: Regular Papers* 59, 1 (2012), 3–29.

[3] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. 2010. Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 26–36.

[4] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, et al. 2007. AsAP: A Fine-Grained Many-Core Platform for DSP Applications. *IEEE Micro* 27, 2 (2007).

[5] Brent Bohnenstiehl, Aaron Stillmaker, Jon J Pimentel, Timothy Andreas, Bin Liu, Anh T Tran, Emmanuel Adeagbo, and Bevan M Baas. 2017. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits* (2017).

[6] 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 (2004), 44–55.

[7] John Y. Chen. 2009. GPU Technology Trends and Future Requirements. In *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 1–6.

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 367–379.

[9] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[10] Weihaw Chuang and Brad Calder. 2003. Predicate Prediction for Efficient Out-of-Order Execution. In *Proceedings of the 17th Annual International Conference on Supercomputing*. ACM, 183–192.

[11] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. 2014. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media.

[12] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. 2012. CPU DB: Recording Microprocessor History. *Commun. ACM* 55, 4 (2012), 55–63.

[13] Paul Gratz, Changkyu Kim, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Stephen W Keckler, and Doug Burger. 2007. On-chip Interconnection Networks of the TRIPS Chip. *IEEE Micro* 27, 5 (2007), 41–50.

[14] GreenArrays. 2010. GreenArrays F18A 18-bit Computer. (2010).

[15] GreenArrays. 2010. Product Brief: GreenArrays. (2010).

[16] Seongmoo Heo and Krste Asanovic. 2004. Power-Optimal Pipelining in Deep Submicron Technology. In *Proceedings of the 2004 international symposium on Low power electronics and design*. ACM, 218–223.

[17] M.S. Hrishikesh, Doug Burger, Norman P Jouppi, Stephen W. Keckler, Keith I. Farkas, and Premkishore Shivakumar. 2002. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *ACM SIGARCH Computer Architecture News*, Vol. 30. IEEE Computer Society, 14–24.

[18] Ziqiang Huang, Andrew D. Hilton, and Benjamin C. Lee. 2016. Decoupling Loads for Nano-Instruction Set Computers. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 406–417.

[19] Angshuman Parashar, Micahel Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 142–153.

[20] Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, Daniel Lustig, Stephen Maresh, Vladimir Pavlov, Rachid Rayess, Antonia Zhai, and Joel Emer. 2015. Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 10.

[21] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Total, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 396–407.

[22] Andrw Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 13–24.

[23] Andrew Putnam, Steve Swanson, Martha Mercaldi, Ken Michelson, Andrew Petersen, Andrew Schwerin, Mark Oskin, and Susan Eggers. 2005. The Microarchitecture of a Pipelined WaveScalar Processor: An RTL-Based Study. *Tech. Rep. TR-2005–11–02* (2005).

[24] Behnam Robatmili, Dong Li, Hadi Esmaeilzadeh, Sibi Govindan, Aaron Smith, Andrew Putnam, Doug Burger, and Stephen W Keckler. 2013. How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 460–471.

[25] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *ACM SIGARCH Computer Architecture News*, Vol. 31. ACM, 422–433.

[26] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast Regular Expression Matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 227–238.

[27] Viji Srinivasan, David Brooks, Michael Gschwind, Pradip Bose, Victor Zyuban, Philip N. Strenski, and Philip G. Emma. 2002. Optimizing Pipelines for Power and Performance. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*. IEEE, 333–344.

[28] 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 (TOCS)* 25, 2 (2007), 4.

[29] David Tarjan, Shyamkumar Thoziyoor, and Norman P Jouppi. 2006. *CACTI 4.0*. Technical Report. Technical Report HPL-2006-86, HP Laboratories Palo Alto.

[30] Michael Bedford Taylor. 1999. *Design Decisions in the Implementation of a RAW Architecture Workstation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[31] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2002. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE micro* 22, 2 (2002), 25–35.

[32] Jesmin Jahan Tithi, Neal C. Crago, and Joel S. Emer. 2014. Exploiting Spatial Architectures for Edit Distance Algorithms. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 23–34.

[33] Louis Woods, Gustavo Alonso, and Jens Teubner. 2013. Parallel Computation of Skyline Queries. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 1–8.

[34] Louis Woods, Gustavo Alonso, and Jens Teubner. 2015. Parallelizing Data Processing on FPGAs with Shifter Lists. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8, 2 (2015), 7.

[35] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. *ACM SIGPLAN Notices* 49, 4 (2014), 255–268.