Chronos: Efficient Speculative Parallelism for Accelerators

Maleen Abeydeera
maleen@csail.mit.edu
Massachusetts Institute of Technology

Daniel Sanchez
sanchez@csail.mit.edu
Massachusetts Institute of Technology

Abstract

We present *Chronos*, a framework to build accelerators for applications with *speculative parallelism*. These applications consist of atomic tasks, sometimes with order constraints, and need speculative execution to extract parallelism. Prior work extended conventional multicores to support speculative parallelism, but these prior architectures are a poor match for accelerators because they rely on cache coherence and add non-trivial hardware to detect conflicts among tasks.

Chronos instead relies on a novel execution model, *Spatially Located Ordered Tasks (SLOT)*, that uses *order* as the only synchronization mechanism and limits task accesses to a single read-write object. This simplification avoids the need for cache coherence and makes speculative execution cheap and distributed. Chronos abstracts the complexities of speculative parallelism, making accelerator design easy.

We develop an FPGA implementation of Chronos and use it to build accelerators for four challenging applications. When run on commodity AWS FPGA instances, these accelerators outperform state-of-the-art software versions running on a higher-priced multicore instance by 3.5× to 15.3×.

CCS Concepts • **Computer systems organization → Multicore architectures**.

Keywords speculative parallelism; fine-grain parallelism; accelerators; specialization; FPGA.

ACM Reference Format:
Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators . In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378454

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378454

1 Introduction

The impending end of Moore's Law is forcing architectures to rely on application- or domain-specific accelerators to improve performance. Accelerators require large amounts of parallelism. Consequently, prior accelerators have focused on domains where parallelism is easy to exploit, such as deep learning [12, 13, 37], and rely on conventional parallelization techniques, such as data-parallel or dataflow execution [48]. However, many applications do not have such easy-to-extract parallelism, and have remained off-limits to accelerators.

In this paper, we focus on building accelerators for applications that need *speculative execution* to extract parallelism. These applications consist of tasks that are created dynamically and operate on shared data, and where operations on shared data must happen in a certain *order* for execution to be correct. Order constraints may arise from the need to preserve atomicity (e.g., operations across tasks must be ordered to not interleave with each other), or from the need to order tasks due to application semantics (e.g., tasks dequeued from a priority queue). Enforcing these order constraints a priori, before running each task, is often too costly and/or limits parallelism. Thus, it is preferable to run tasks *speculatively* and check that they followed a correct order a posteriori.

For instance, consider discrete event simulation, which has wide applicability in simulating digital circuits, networked systems, and physical processes. Discrete event simulation consists of dynamically created tasks that may operate on the same simulated object and must run in the correct simulated time order. Running these tasks non-speculatively requires excessive synchronization and limits parallelism [10, 28]. Running tasks speculatively is far more profitable [32, 34].

To make speculation efficient, prior work has proposed hardware support for speculation, including Thread-Level Speculation [21, 34, 53, 55, 57], and Hardware Transactional Memory [1, 6, 9, 20, 26, 29, 30, 46]. Unfortunately, prior speculative architectures are hard to apply to accelerators, because they all *rely on coherent cache hierarchies* to perform speculative execution, modifying the coherence protocol to detect conflicts among tasks. This is a natural match for multicores, which already have a coherence protocol. But such a solution would be onerous and complex for an accelerator: it would require implementing coherent caches and speculation-tracking structures that, while a minor overhead for general-purpose cores, are too expensive for small, specialized ones.

To address this challenge, in this paper we present a hardware system that implements speculative execution *without using coherence*. Instead, this system follows a *data-centric approach*, where shared data is mapped across the system; work is divided into small tasks that access at most one shared object each; and tasks are always sent to run at the place where their data is mapped. To enforce atomicity across task groups, or other order constraints, tasks are ordered through timestamps (these are program-specified logical timestamps completely decoupled from physical time).

We formalize these semantics through the *Spatially Located Ordered Tasks* (SLOT) execution model. In SLOT, all work happens through tasks that are ordered using timestamps. A task may create children tasks ordered after them, and parent tasks communicate input values to children directly. Each task *must operate on a single read-write object*, which must be declared when the task is created (besides this restriction, tasks may access an arbitrary amount of read-only data).

We leverage SLOT to implement *Chronos*, a novel acceleration framework for speculative algorithms. Each Chronos instance consists of spatially distributed tiles. Each tile has multiple *processing elements* (PEs) that execute tasks, and a local cache. Each tile also implements hardware to queue tasks, dispatch them to PEs, track their speculative state, and abort or commit them in timestamp order. Chronos maps read-write objects across tiles, and sends each newly created task to the tile where its read-write object is mapped. This enables completely distributed operation without a cache coherence protocol.

Chronos provides a common framework to accelerate speculative algorithms, abstracting away the complexities of task management and speculative execution. Developers need only express their application as SLOT tasks coded against a high-level API. To achieve high performance, Chronos supports two types of customization. First, applications can customize the PEs, which can be specified in RTL or described using High-Level Synthesis (HLS). PEs can also be general-purpose cores, so developers can start with a software implementation and specialize tasks as needed to achieve high performance. Second, Chronos lets applications turn off unneeded features. For example, if the algorithm is naturally resilient to out-of-order writes (e.g., if updates are monotonic), applications can disable rollback on misspeculation.

We evaluate Chronos by implementing it on an FPGA and use it to implement accelerators for several graph analytics and simulation applications. We use four hard-to-parallelize applications with speculative parallelism. We deploy these accelerators on commodity AWS FPGA instances. We compare these accelerators with state-of-the-art software implementations of these applications running on a higher-priced 40-thread multicore instance. Chronos achieves speedups of up to 15.3× and gmean 5.4× over the software versions. Chronos outperforms the multicore baseline *despite running at a 19× slower frequency*, because it exploits orders of magnitude more parallelism. These results show that FPGAs are a practical and cost-effective way to accelerate applications with speculative parallelism.

```
PrioQueue<Time, GateInput> eventQueue;

void simToggle(Time time, GateInput input) {
 Gate gate = input.gate;
 bool outToggled = gate.simulateToggle(input);
 if (outToggled) {
  // Toggle all inputs connected to this gate
  for (GateInput i : gate.connectedInputs()) {
   Time nextTime = time + gate.delay(input, i);
   eventQueue.enqueue(nextTime, i);
}}}

... // Enqueue initial events (input waveforms)
// Main loop
while (!eventQueue.empty()) {
 (time, input) = eventQueue.dequeue();
 simToggle(time, input);
}
```

Listing 1. Sequential implementation of `des`.

In summary, this paper contributes:

- SLOT, the first execution model that supports speculative parallelism without cache coherence (Sec. 3).
- Chronos, a customizable framework that implements the SLOT execution model and makes it easy to accelerate applications with speculative parallelism (Sec. 4).
- A detailed evaluation of Chronos using commodity FPGAs in the cloud that demonstrates significant speedups for several challenging applications, analyzes system efficiency, and quantifies the benefits of customization (Sec. 6).

Our Chronos implementation is open-source and available at https://chronos-arch.csail.mit.edu.

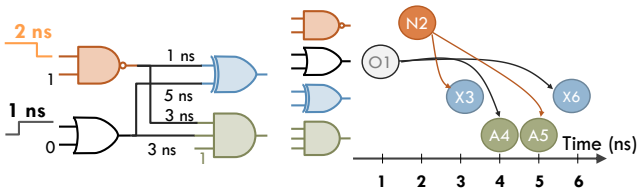
2 Motivation and Background

In this section we first present a case for speculative parallelism through a simple application, discrete event simulation (`des`). We then review the types of parallelism exploited by prior accelerators, and see that most do not exploit speculative parallelism. Finally, we review prior speculative architectures, and use `des` to identify a key simplification that these architectures have missed: support for task order avoids the need for coherence-based conflict detection, motivating SLOT.

2.1 A case for speculative parallelism

We illustrate the utility of speculative parallelism through `des`, a discrete event simulator for digital circuits [28]. Listing 1 shows code for a sequential implementation of `des`. Each `des` task processes a gate input toggling at a particular time. If this input toggle causes the gate's output to toggle, the task enqueues events for all inputs connected to that output at the appropriate times. The sequential implementation processes one task at a time in simulated time order, and maintains the set of tasks to process in a priority queue.

Fig. 1a shows a circuit with input waveforms and propagation delays, and Fig. 1b shows the task diagram of an execution of `des` on this circuit. Arrows between tasks show



(a) Example circuit (b) Ordered tasks produced by *des*

Figure 1. Example execution of `des`: (a) Input circuit and (b) Ordered tasks produced by the execution of `des` on this circuit. Tasks O1 and N2 correspond to inputs of the NAND and OR gates; both tasks toggle their gates' outputs, producing tasks X3, X6 and A4, A5.

parent-child dependences (e.g., task O1 creates tasks A4 and X6). The x-axis shows task order, and the task's location in the y-axis represents the gate it operates on.

Parallelism exists despite order constraints because independent tasks may run out of order. In our `des` example, only tasks that operate on the same gate have a data dependence; others (e.g., O1 and N2) may run out of order without violating correctness. But tasks and dependences are not known, so running tasks out of order is not straightforward.

In `des`, each task operates on a single object (a gate), and this object is known in advance. But this is not sufficient to find which tasks are safe to run, because *a task may have a dependence with another task that comes earlier in program order but does not yet exist*. Suppose that task O1 is executed first, producing task X6. At this point X6 is the earliest task in the system that operates on the XOR gate. But executing X6 would produce incorrect results, because X6 must follow the earlier data-dependent task X3, which does not yet exist (as N2 has not been run).

A natural way to parallelize `des` is to run tasks in parallel, speculating that, for each task, no earlier data-dependent tasks will exist. If speculation is correct, the task can commit and the system has successfully elided order constraints; but on an order violation, the misspeculating task and its descendants need to be aborted and re-executed in the right order.

This execution strategy, known as Time Warp in the case of `des` [31], shows that speculation arises from the need to preserve order constraints, *even though the data that each task accesses is known in advance*. As we will see later, advance knowledge of task data accesses enables a simple implementation of speculative execution.

2.2 Types of parallelism in prior accelerators

We classify prior accelerators by the type of parallelism they target. We can establish a taxonomy of parallelism types based on two key questions. First, are tasks known in advance or are they created dynamically? Second, if tasks operate on shared data, how should they synchronize to respect the algorithm's data dependences and produce the right result?

Static parallelism: If tasks and their data dependences are known in advance, scheduling can be done statically and requires no or very simple runtime mechanisms. Static parallelism arises when operating on regular data structures, such as dense matrices. Most prior accelerators focus on static parallelism, e.g., by building deep pipelines and data-parallel hardware such as in DaDianNao [12] and Google's TPU [37].

Dynamic parallelism with independent tasks: Some algorithms, such as those that operate on trees or graphs, must create tasks dynamically, as they find more work to do. In the simplest case, tasks operate on disjoint data and need no synchronization for shared data accesses. They fit the *fork-join* model pioneered by NESL [5] and Cilk [16]. The ParallelXL [11] and TAPAS [43] accelerators target this dynamic parallelism. Their key ingredient is hardware support for task creation and load-balancing, e.g., through work-stealing [16].

Non-speculative synchronization of dependent tasks: Prior work has demonstrated accelerators where tasks operate on shared data, but most synchronize by *stalling rather than speculating*. Graphicionado [23] and Li et al. [40] are accelerators for graph algorithms that support atomic operations through pipelining: they stall a later dependent task until the earlier task finishes its update.

Speculative synchronization of dependent tasks: Finally, Ma et al. [42] build an accelerator for graph analytics applications on FPGA. They support atomic tasks. Each task can access multiple addresses, and conflict detection is implemented using a globally shared address-tracking structure, similar to a coherence directory. This approach is thus analogous to coherence-based conflict detection, which we review next, and suffers from additional overheads (as all accesses, instead of only cache misses, access the global directory). By contrast, SLOT avoids the need for coherence by restricting each task to operate on a single object, and supports ordered tasks to enable multi-object atomicity.

2.3 Prior speculative architectures rely on cache coherence

Prior architectures for speculative parallelization, such as Thread-Level Speculation (TLS) [24, 34, 53, 55, 57, 65] and Hardware Transactional Memory (HTM) [25, 26, 29, 46, 52], extend a cache-coherent multicore. These systems *reuse existing mechanisms* to implement speculative execution. Specifically, they adapt the cache coherence protocol for conflict detection.

Coherence-based conflict detection works by leveraging invalidation and downgrade messages to detect conflicts. Each task runs in a single core. The core acquires coherence permissions for each read and write as usual, but keeps permissions for these lines throughout the execution of the task (either by keeping the task's data in the private cache [1, 9, 55], or by tracking these permissions in the shared directory [34, 46, 64]). Thus, the core will receive an invalidation or downgrade request on every possible conflict (i.e., if another task issues a read to a line in the task's write-set, or any access to a line in the task's read- or write- set).

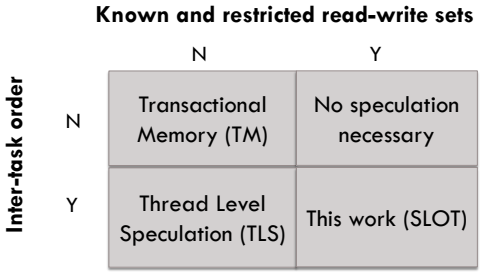


Figure 2. A categorization of the reasons for speculative execution. Cache coherence is only required if read-write sets are unknown or unrestricted. We will show that inter-task order is sufficient for all aplications.

While relying on coherence is reasonable for multicores, it would be expensive for accelerators. Accelerators in general and reconfigurable hardware in particular do not have an coherent cache hierarchy that supports invalidation-based conflict detection [54]. Implementing coherence would add complexity, latency, and significant on-chip SRAM to implement a directory that tracks sharers. Beyond coherence, performing conflict detection on tasks with arbitrary read and write sets would add further overheads, e.g., several kilobytes per core worth of Bloom filters [34], which would be too onerous for specialized processing cores.

Indeed, while there have been FPGA implementations of cache coherence protocols [45, 63] and HTMs [8, 62], these systems were not designed as accelerators and these features added significant overheads.

2.4 Reasons for speculative execution

In general, speculation is needed when tasks have *either* unknown read- and write- sets *or* inter-task order constraints.

By relying on coherence, prior speculative systems support tasks with unknown read- and write-sets. Speculation allows HTM systems to preserve atomicity among unordered tasks (transactions), and TLS systems to enforce both atomicity and order among tasks. But as we saw in Sec. 2.1, `des` needs speculation to elide order constraints among tasks even though each task's read/write-set is known in advance.

Fig. 2 presents systems according to their reasons for speculative execution. As we can see, prior architectures all support tasks with unknown read- and write-sets, which forces complex conflict detection. In this paper, we focus on the remaining quadrant: supporting *inter-task order* only, but where tasks have *known and restricted read- and write-sets*. This simplification is sufficient for `des`; in the next section, we will see that inter-task order is in fact sufficient to support unknown read- and write-sets, because *inter-task order allows breaking work into tasks with known read- and write-sets*.

3 The SLOT Execution Model

We now present the *Spatially Located Ordered Tasks* (SLOT) execution model. SLOT restricts each task to access a single read-write *object*, which must be known when the task is created (Sec. 3.1). Though this restriction may seem limiting, *inter-task order* enables atomicity among computations that access multiple objects and where objects are not known in advance (Sec. 3.2).

```
void simToggle(Time time, GateInput input) {
 Gate gate = input.gate;
 bool outToggled = gate.simulateToggle(input);
 if (outToggled) {
  // Toggle all inputs connected to this gate
  for (GateInput i : gate.connectedInputs()) {
   Time nextTime = time + gate.delay(input, i);
   slot::enqueue( simToggle,   // task type
                  nextTime,    // timestamp
                  i.gate.ID,   // object id
                  i  );        // args
}}}
```

Listing 2. SLOT implementation of `des` task.

3.1 Spatially Located Ordered Tasks

SLOT applications consist of ordered, dynamically created tasks. Each task can be implemented in software or hardware. We describe the execution model independently of the implementation, and illustrate it using the software API.

Each task is given two attributes when it is created: a *timestamp* and an *object id*. Timestamps specify order constraints: the system guarantees that tasks appear to execute in timestamp order. Tasks with equal timestamps may run in any order, but are atomic (i.e., they do not interleave).

Object ids are integers that specify the data dependences among tasks: two tasks are treated as data-dependent *if and only if they have the same object id*. Object ids restrict each task to accessing at most one read-write object in shared memory. Note that this restriction only applies to read-write data. A task may access any amount of read-only data.

A SLOT task can create children tasks as it finds more work to do, by specifying the type of the child task, as well as its timestamp, object id, and any input data values it may need. Each child task may have any timestamp that is greater than or equal to its parent's.

In SLOT, parent-child relations are *unidirectional*: a parent task can create and pass values to its children, but parents are ordered before their children and thus appear to complete before children execute. Child tasks cannot return values or communicate with their parents. This is different from fork-join execution models like Cilk [16], where parents wait for their children to complete.

API: Listing 2 illustrates the SLOT software API by showing the implementation of a `des` task. In software, each task is implemented by a function. The implementation is almost the same as the sequential one in Listing 1: each task simulates an input toggle at a particular gate. Instead of enqueuing tasks to a priority queue, this code creates new tasks by calling `slot::enqueue`, which specifies the child task's type (its function pointer since it's a software task), timestamp, object id, and any additional arguments (the gate input in this case).

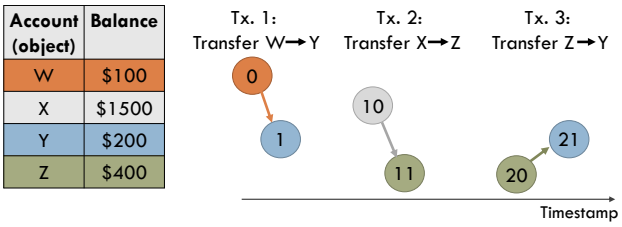


Figure 3. Example showing how to leverage order to implement atomic accesses to multiple read-write objects. Each transaction is broken down to multiple tasks that access one object each. Atomicity is maintained by assigning a disjoint timestamp range to each transaction.

SLOT enables coherence-free conflict detection: By restricting each task to access at most one read-write object, implementations of SLOT can perform distributed conflict detection without complex tracking structures. If the implementation maps object ids across cores or tiles, and sends each task to where its object id is mapped, then finding conflicts becomes a local operation.

For example, if Fig. 1 was run on a four-core system, the NAND, OR, XOR, and AND gates could be mapped to cores 1–4. Then, if task X3 arrives in core 3 after X6 has already run, core 3 can determine X3's conflicts (tasks for the same gate and a higher timestamp, {X6} in this case) locally, by comparing X3's object id with those of still-speculative tasks.

3.2 Mapping multi-object computations to SLOT

While SLOT's single-object restriction is natural for applications like `des`, many applications must perform atomic accesses to multiple read-write objects, and may not know all these objects in advance.

Nonetheless, SLOT's *support for order* enables a trivial, systematic mapping of these computations to SLOT. Specifically, multi-object transactions can be expressed as SLOT tasks by breaking each transaction into multiple single-object tasks, each accessing a single object, and *giving each transaction a disjoint range of timestamps*. This way, tasks within a transaction do not overlap with those in other transactions.

For example, consider a banking application where transactions transfer money between accounts. Each transaction must atomically decrement the source account's balance and increment the destination account's balance. To scale, each account should be a different object; but since account balances are read-write data, a single task cannot access two accounts.

Fig. 3 shows how task order makes this possible. We implement each transaction using two SLOT tasks, each of which manipulates a single account: the first decrements the source's balance and creates a second task to increase the destination's balance. Each transaction has a disjoint range of timestamps, so tasks from different transactions do not interleave.

This technique generalizes to arbitrary combinations of read-write operations. For example, our implementation of `maxflow` (Sec. 5) uses it to perform complex atomic operations on the neighborhood of a graph vertex.

While breaking each transaction into many small tasks could add significant overheads to a software runtime, small tasks are a natural match for an accelerator, as hardware performs task management and small tasks need simple processing elements.

3.3 Discussion

Benefits of SLOT's fine-grained tasks: SLOT's key advantage over prior work is to enable coherence-free conflict detection. In addition, prior work [33, 59] has shown that, even in systems that support tasks with arbitrary read/write-sets, this division is often desirable, for three key reasons:

1. *Increased parallelism:* Breaking a long serial transaction into short tasks allows these tasks to run in parallel.
2. *Reduced impact of aborts:* On misspeculation, only the tasks that conflict are aborted, rather than the entire transaction.
3. *Increased data reuse:* Rather than bringing shared data across the system where the transaction is running, tasks are sent to run close to their data, avoiding cache line ping-ponging. Since each task message is much smaller than a cache line, this reduces traffic; and tasks are sent and executed asynchronously, so their latency is easier to hide than that of synchronous memory accesses.

SLOT limitations: While breaking programs into short single-object tasks is generally beneficial, there is one case where coherence-based conflict detection would outperform SLOT: if the application is dominated by *rarely modified* read-write data that has substantial reuse, coherence-based conflict detection would allow caching this data across the system, making reads between the sporadic writes local, whereas SLOT needs to isolate each access to these data in a separate task and send them to a single place.

We do not find this behavior in the applications we target, so we have not optimized SLOT for this case. A simple extension of SLOT could address this by letting tasks write into addresses not covered by its object id. The system could then treat rarely modified data as read-only and allow them to be cached privately. Upon a write, which should be rare, a simple implementation could flush all caches and abort all future tasks; more complex implementations may perform more precise flushes and conflict detection. We leave a detailed study of these implementation choices to future work.

Relationship with prior work: Spatial hints [33] and Espresso [35] also propose to tag tasks with an identifier similar to an object id, but with different goals. Spatial hints are used to distribute speculative tasks so that tasks *likely* to access the same data run at the same place. But spatial hints are optional and advisory, and the system must still use coherence-based speculation. Espresso uses locales to additionally provide mutual exclusion among *non-speculative* tasks. By contrast,

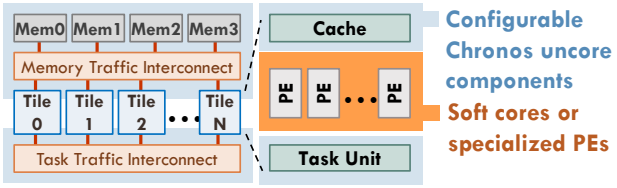


Figure 4. Chronos system overview.

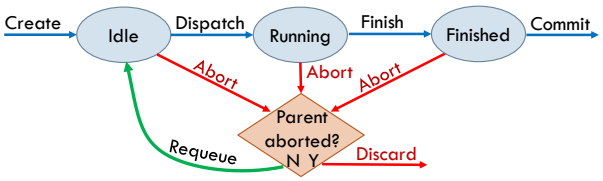


Figure 5. Task life cycle.

SLOT requires all tasks to specify correct object ids, i.e., to identify the read-write object they will access. This enables using object ids to implement conflict detection among speculative tasks.

4 Chronos System

Chronos is an architectural framework that makes it easy to design accelerators for applications with ordered parallelism. Chronos achieves this by providing an *architecture template* that implements the SLOT execution model efficiently. Accelerators can then specialize this template by defining their own task processing engines or configuring Chronos's uncore components. With this division, creating a Chronos accelerator is as simple as specifying the processing engines; the framework takes care of the intricacies of ordered task management and speculative execution.

Fig. 4 shows Chronos's organization. Chronos is a tiled design with fully distributed task management and speculation mechanisms. Each tile has several Processing Elements (PEs) that execute tasks, a local (non-coherent) cache, and a *task unit* that queues, dispatches, and commits ordered tasks.

4.1 Design requirements and techniques

Chronos must run short ordered tasks efficiently. This requires achieving *high-throughput task management* and a *large speculation window*:

1. High-throughput task management: Short tasks place high throughput demands on the system. For example, if each task takes 20 cycles to execute, a Chronos system with 200 PEs must create, dispatch, conflict-check, and commit 10 tasks per cycle to keep the PEs busy. This forces a design *without centralized components*: all task management and speculation mechanisms must be fully distributed. Chronos's tiled design achieves this. Moreover, each tile's task unit needs to maintain a high throughput as well.

2. Large speculation window: To prevent order from limiting parallelism, the system must be able to speculate far ahead of the earliest unfinished task. More specifically, due to order restrictions, tasks may stay speculative for a long time before they can commit—far longer than the time they take to execute. Therefore, the system should be able to track many more speculative tasks than running tasks. For example, as we will see in Sec. 6, some applications require about 10 speculative tasks per running task.

These requirements force *fully distributed, deep out-of-order execution*. To achieve these requirements, several of Chronos's techniques are adapted from Swarm [34]. Specifically, Chronos borrows Swarm's *task management* and *ordered commits* techniques. However, Chronos implements speculative execution differently, by leveraging the SLOT execution model instead of relying on a coherent cache hierarchy. We first describe how Chronos performs speculative execution, then detail its task management structures.

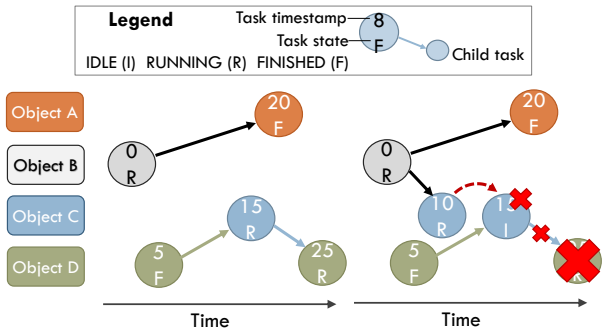
4.2 Distributed ordered speculation

Chronos uses speculative execution to elide order constraints. Chronos can run any task as soon as it is created, even if its ancestors are still speculative. Fig. 5 shows the execution flow of each task. Top horizontal arrows denote correct speculation. When a task is created, it is sent to a tile where it stays *idle*, queued until it is ready to dispatch. The tile dispatches idle tasks to PEs in timestamp order. After a *running* task finishes execution, it stays speculative (in the *finished* state) until the system determines it is safe to commit.

Fig. 5 shows that tasks may be aborted at any point before commit. Because tasks may run while their ancestors are still speculative, aborting a task requires aborting *and discarding* all its descendants. These cascading aborts are necessary to uncover parallelism, and are *selective*: aborts undo the effects of the aborting task, its descendants, and any data-dependent tasks that come later in program order. As shown in Fig. 5, if a task is aborted because its parent has aborted, then it is discarded; otherwise, the abort is due to a data dependence, then the task is requeued for execution.

Fig. 6 shows an example of speculative execution in Chronos. Tasks are created and run out of order: in Fig. 6a, task 20 has run and finished even though earlier tasks are still running; in particular, task 0, 20's parent, is still running. In Fig. 6b, task 0 creates a child with timestamp 10, which conflicts with task 15. This causes 15 to be aborted, along with its child task 25. Though aborts may affect multiple tasks, they are selective: independent tasks such as 20 are not aborted.

Task mapping and conflict detection: To perform speculative execution cheaply, Chronos uses the task mapping and conflict detection strategy outlined in Sec. 3: Chronos maps object ids across tiles, then sends each created task to the tile where its object id is mapped. Our current Chronos implementation uses a static object-to-tile mapping: the object id is simply hashed to produce the tile id. We find this achieves good load balance in our workloads; Chronos could also adopt more sophisticated load balancing based on dynamic remapping of objects among tiles [33].



(a) Tasks can run out of timestamp order

(b) Selective aborts: Task 10 aborts 15 and 25, but not 20

Figure 6. Chronos uncovers parallelism by running tasks out-of-order (a). On a conflict, Chronos aborts are selective: the only affect the mispeculating tasks (b).

Task dispatch: The task unit dispatches tasks to PEs in timestamp order to prioritize earlier tasks. To avoid conflicts, the task unit serializes the execution of tasks with the same object id. Therefore, conflicts among running tasks never arise; only a task that arrives to a tile out of order can create a conflict.

Speculative value management: Chronos adopts eager version management: speculative writes update memory in-place, and old values are written to a separate *undo log*. Commits are fast, as the undo log is simply discarded; aborts require restoring the old values from the undo log.

Eager version management facilitates running chains of data-dependent tasks without waiting for them to commit: if task *A* writes a value that is later read by (same-object) task *B*, *B* will naturally use *A*'s value even when *A* has not yet committed. This process, known as *speculative forwarding*, is important for ordered speculation [35], but would be hard to do with lazy version management.

High-throughput commits: To determine when a task can commit, Chronos borrows the Global Virtual Time (GVT) protocol from prior work [32, 34]. Tiles communicate periodically (e.g., every 32 cycles) to find the timestamp of the earliest unfinished task, then commit all earlier tasks. This process leverages large commit queues to commit many tasks at once, achieving commit throughput of multiple tasks per cycle with little communication.

4.3 Task unit design

Chronos's task unit consists of two main structures: a *task queue* (TQ) holds all tasks in the tile and dispatches *idle* tasks to PEs, and a *commit queue* (CQ) that holds the speculative state of *running* or *finished* tasks, and commits or aborts them as required. In addition, a small *task send buffer* (TSB) receives newly created tasks from PEs and sends them to the right tile. Fig. 7 details the microarchitecture of each tile and shows these structures, which, together, are similar to a task-level reorder buffer.

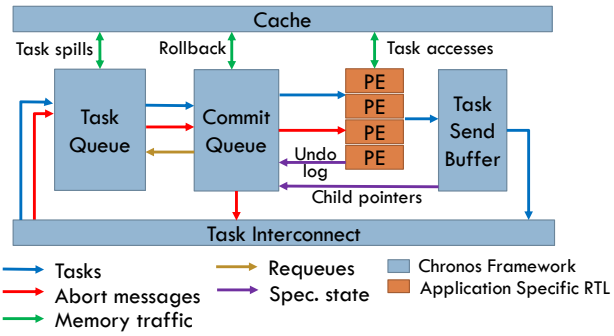


Figure 7. Chronos tile microarchitecture.

4.3.1 Task queue

The task queue consists of two main structures: a *task array* and an *order queue*. The task array is a simple memory that stores the *task descriptor* of every task in the tile. Each task descriptor contains all data needed to run the task: its type, timestamp, object id, and arguments. The order queue holds *idle* tasks and dispatches them to PEs in timestamp order.

Tasks allocate entries in the task array and order queue when they arrive to the tile. They hold their order queue entry until they are dispatched to a PE, but hold their task array entry throughout their lifetime (i.e., until they commit or are discarded). This is so that, if the task is aborted, the task array has the information needed to reexecute it. When a task needs to reexecute, it is reinserted into the order queue.

Task spilling: Task queues have limited capacity, but SLOT programs may create an unbounded number of tasks. Chronos provides the illusion of unbounded task queues by spilling tasks to main memory when a task queue is nearly full.

4.3.2 Commit queue

The commit queue holds the speculative state of all tasks that are either running or finished. In Chronos, this speculative state consists of the task's *undo log*, which allows rolling back the task's memory writes, and *child pointers*, which allow aborting the task's descendants.

Each child pointer tracks the tile and task array entry id of a child task. When a child is created, it is sent to the tile specified by its object id. When the receiving tile queues it, it replies with the child task's pointer.

Aborts: Every abort event needs to abort the current task and its data-dependent tasks, and discard all their descendants. Chronos handles all data dependences by aborting the task and all the same-object tasks that come later in program order. Chronos rolls back this sequence of tasks by applying their undo logs in reverse execution order and sending abort notifications to all child tasks, which initiate their own aborts.

Commits: Chronos implements the GVT protocol to perform high-throughput commits, as noted above. Periodically (e.g., every 32 cycles), each tile finds the timestamp of its earliest unfinished task, called the *local virtual time* (this is simply the minimum of the timestamps in the order queue, PEs, and

TSB). Tiles send their local virtual times to an arbiter, which finds the minimum, i.e., the timestamp of the earliest unfinished task in the system, called the *global virtual time* (GVT). Finally, the arbiter broadcasts the GVT. The commit queue in each tile scans its entries in the background, and frees (i.e., commits) those whose timestamp is lower than the GVT.

Deadlock avoidance: Chronos prevents deadlock by *never blocking the lowest-timestamp task*. By induction, this strategy ensures that all tasks ultimately commit. If the lowest-timestamp (i.e, earliest) task cannot be dispatched because the commit queue is full, the commit queue aborts one of its entries (the highest-timestamp entry) and lets the earliest task proceed. To prevent the earliest task from being stalled because the TSB is full, the TSB reserves one of its entries for the earliest task.

Commit entry implementation: Since SLOT tasks are short, the commit queue implements a simple storage format with a fixed number of child pointers and undo log entries. These values are configurable per application. We find that having eight child pointers per task (as in prior work [34]) and eight address-value pairs per undo log suffices for all tasks. Tasks that exceed these limits could be split to use multiple commit queue entries [58]. Alternatively, more sophisticated implementations could support variable entry sizes, or unbounded sizes by spilling to memory [46].

4.3.3 Task send buffer

The task send buffer (TSB) buffers child tasks created by PEs and sends them to their destination tiles. Each entry stays in the buffer until the destination tile acknowledges its receipt. The TSB decouples PEs from task enqueue latency: it lets PEs continue execution and even move to other tasks before the current task's child enqueues have been acknowledged.

4.3.4 Processing Element interface

The PEs within a tile execute the functionality of each task. If the application program consist of more than one task type, the PEs could either be homogeneous (any PE can execute any task) or heterogeneous (a different PE for each task type). Chronos admits multiple styles of PE, from programmable cores to fully specialized engines, and only requires that they implement a simple interface.

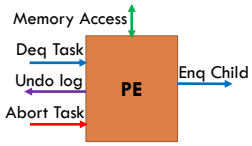


Figure 8. PE interface.

Fig. 8 details the PE interface, which consists of five ports. All ports use a simple valid/ready handshake mechanism and support pipelining. The PE signals it can accept new tasks and receives them through the *Deq Task* port. It sends children to the TSB through the *Enq Child* port, and issues memory accesses through the *Memory Access* port. The task unit may abort a running task through the *Abort Task* port. Finally, when the PE finishes or aborts a task, it outputs its undo log through the *Undo Log* port.

Enforcing SLOT restrictions: If a SLOT task tries to access a different read-write object than the task's object id provided at task creation time, the expected behavior is undefined. To ease debugging, Chronos can be configured with a debug aid that detects all memory accesses whose task has an incorrect object id, and stops execution on any such violation.

4.4 Chronos customization

Chronos is fully customizable, including the number of tiles, number and type of PEs, and cache geometry. Moreover, Chronos can relax its operational features to take advantage of application characteristics. Some applications can be made resilient to out-of-order writes (e.g., applications that perform directed graph searches, as we will see in Sec. 6). In this case, Chronos can be configured to not perform rollback on aborts. This simplification removes the undo log and commit queue. As we will see in Sec. 6, this saves about 30% of area, enabling designs with more tiles and thus more parallelism.

Even with no-rollback execution, respecting task order is still important. The algorithm may be resilient to order violations, but frequent violations make these algorithms work-inefficient. Therefore, task queues still dispatch tasks speculatively in timestamp order.

Finally, Chronos can also be used for non-speculative applications. In this case, in addition to disabling rollback, the task queue dispatches tasks in FIFO order, removing the order queue and further reducing area.

5 Methodology

Chronos FPGA implementation: We implement the Chronos framework in SystemVerilog. We use a pipelined heap [4] to implement the order queue, and a TCAM adapted from [41] to find conflicting tasks in the commit queue.

Our FPGA implementation is fully configurable in terms of number of tiles, number of cores and their types, and task and commit queue sizes. We use the Amazon AWS FPGA framework [3], and develop Chronos as a CL (Custom Logic) module. This CL module interacts with the AWS Shell, which provides I/O services such as memory and PCI controllers.

We synthesize our CL module using Vivado 2018.1. We target AWS f1.2xlarge instances, which have the Xilinx UltraScale+ VU9P FPGA. This FPGA is fairly large, featuring 1.2M LUTs, 76Mb of Block RAM, and 270Mb of Ultra RAM. We use URAM for the caches and BRAM for the task queues. These resources were sufficient to fit systems of up to 16 tiles while meeting a target frequency of 125 MHz. Table 1 details the Chronos configurations used, and Fig. 9 shows the layout of a 16-tile Chronos system on FPGA.

Applications: We build Chronos accelerators for four challenging applications. We compare their performance against highly optimized software-parallel implementations:

1. `des` performs gate-level simulation of logic circuits. We use the CMB implementation in Galois [50] as our baseline.

Task Queue	4096-entry task-array 8192-entry order queue
Commit Queue	128 entries default; 256 for `maxflow`
Task Send Buffer	16 entries
Cache	2 MB/tile default; 1 MB for `sssp`; 4-way set associative, 64B cache lines, 5-cycle hit latency from a PE
Data Widths	32-bit timestamp and object id
Clock Frequency	125 MHz

Table 1. Chronos tile configuration parameters used.

Application	Baseline	Input	N. Tasks
`des`	Galois [50]	csa32 [50]	3.1M
`maxflow`	Galois[50]	rmf-wide[2]	7.8M
`sssp`	Galois[50]	USA-roads [15]	58.0M
`astar`	In-house	germany-roads [22],	4.1M
`color`	[27]	com-youtube [39]	5.8M

Table 2. Applications accelerated using Chronos, their baselines, inputs, and total number of tasks executed.

Figure 9. FPGA layout of the 16-tile Chronos sssp accelerator. Each Chronos tile is shown in a different color.

We do not compare against the software speculative version of the algorithm (TimeWarp) since software speculation adds thousands of cycles per task [7, 28], a huge overhead since each `des` task takes tens of cycles.

2. `maxflow` finds the maximum amount of flow that can be pushed from a source to a destination node through a network. We use the Galois [50] implementation of the push-relabel algorithm as our baseline. In the baseline, each task operates on a vertex's neighborhood, involving atomic accesses to both the vertex and its neighbors. Our Chronos implementation uses more fine-grained tasks where each neighbor access is a separate task and the atomicity of the original task is preserved using order, as described in Sec. 3.2.

3. `sssp` finds the shortest distance between a given source node and all other nodes in a directed graph. This benchmark admits the no-rollback optimization since the Dijkstra's algorithm can be made resilient to order violations by allowing a node's distance to be set to a non-final value. We compare `sssp` with the Galois implementation, which uses Delta Stepping [44]. We pre-tune Delta to the graph used to put the software version in the best possible light.

4. `astar` performs a heuristic-directed search to find the least cost path towards a goal node. We evaluate astar using a road graph, where the distance function is the great-circle distance between two points given their (latitude, longitude) coordinates. Like `sssp`, `astar` admits the no-rollback optimization. We could not find a high-performance parallel implementation of `astar`. Therefore, we implement our own, using Galois and its `obim` scheduler, and use it as the baseline.

`color`*:* We also implement a non-speculative graph coloring algorithm to show that Chronos can be used with non-speculative tasks. `color` assigns a color to all graph nodes such that no two adjacent nodes share the same color. We use the Jones-Plassmann [36] algorithm with the largest-degree-first heuristic. We compare against a baseline implementation from Hasenplaugh et al. [27]. All `color` tasks are unordered, and rely on object ids to serialize tasks for the same node.

Table 2 details the baselines, input sets used, and the number of tasks executed for these applications.

Baseline system: We run the software baselines on a 20-core/40-thread m4.10xlarge AWS instance. These instances use a 2.4 GHz Intel Xeon E5-2676v3 (Haswell) CPU. We chose this instance specifically because its price is comparable with the FPGA one ($2/hour vs. $1.6/hour for the FPGA instance), which we believe is a fair metric when comparing application performance on different hardware substrates.

5.1 PE implementations

Specialized PEs: We write specialized PEs for each application in SystemVerilog. Our PEs are pipelined, with 4-30 pipeline stages per PE. Each stage performs some computation and may issue a single memory access. Pipelining is flexible: tasks stalled on a memory access do not block other tasks, which can overtake them and proceed to later stages. Each PE has sufficient storage for 32 in-flight tasks. However, we find that a single PE often saturates task and cache bandwidth with fewer in-flight tasks.

PEs can also be generated from a C-like description using High-Level Synthesis (HLS). Specifically, `astar`, which has trigonometric computations that are tedious to write in SystemVerilog, uses HLS to generate most of the pipeline.

RISC-V cores: In addition to application-specific cores, we also built a Chronos version with RISC-V cores. We use the Spinal HDL RISC-V core generator [56] to generate a 32-bit core that has a performance of 1.2 DMIPS/MHz. We extend these cores to implement Chronos `enqueue` and `dequeue` operations, and use the interrupt logic to abort running tasks.

We write SLOT C implementations of `des`, `maxflow`, `sssp` and `color`, compile them to the RISC-V ISA, and run them on a 48-core Chronos system (4 tiles with 12 cores each). We later compare this system with those using application-specific cores to quantify the benefits of PE specialization.

For all timing measurements, we ignore the benchmark setup time, including data transfer time between the host CPU

and FPGA, and only consider the runtime of the algorithm. This is reasonable since the time spent transferring data can be amortized over multiple queries on the same graph. We instrument the FPGA design to count cycles, profile efficiency, and measure detailed component utilization.

6 Evaluation

We first compare the performance and scalability of the Chronos FPGA accelerators with application-specific PEs against the software-parallel versions on the Xeon CPU.

Fig. 10 shows the speedup of each of the four applications as the number of threads or tiles grows until they fill each system. Because each design has different numbers of tiles, the x-axis is percentage of system utilization. For the software-parallel versions, we sweep the number of threads from 1 to 40. For FPGA versions, 100% system utilization corresponds for the maximum system size we can fit on the FPGA (6-16 tiles, depending on the application). Table 3 reports this maximum size. To obtain results at lower utilization, we first disable tiles, and then disable the number of concurrent tasks on a single-tiled system. `sssp` and `astar` admits the no-rollback optimization. Table 3 also summarizes the performance of the FPGA and baseline implementations at the best-performing system sizes.

For all four ordered applications, Chronos performs better than the best CPU baseline, achieving a gmean speedup of 5.4×. Chronos is 15.3× faster than the CPU version on `des`.

Table 3 also shows the progression of speedups relative to a serial CPU implementation as the FPGA system is scaled from a single concurrent task, a single tile, and the full system.

Note that the FPGA runs at a 19× slower frequency than the CPU. Hence, except in `des`, the FPGA version with a single concurrent task is substantially slower than the serial CPU version. This slowdown is typically 1.7–9×, better than the 19× difference in frequencies, because the FPGA PEs are customized to each application.

Nonetheless, Chronos more than makes up for this handicap by *scaling to many concurrent tasks*: Chronos accelerators all scale well, sometimes beyond 100×, because they exploit abundant parallelism that is not available to CPU versions.

6.1 Application analysis

We now look at each application in detail.

des: Even when running a single task at a time on one PE, Chronos is actually 2.45× faster than the baseline (left column of Table 3). This happens even though Chronos is running at a 19× lower frequency. This is primarily because the baseline maintains a priority queue in software, whereas the Chronos framework provides a much higher throughput implementation in hardware. As the number of concurrent tasks increases, the Chronos implementation scales to a self-relative speedup of 44.9× at 8 tiles, corresponding to a 15.3× speedup over the best CPU implementation.

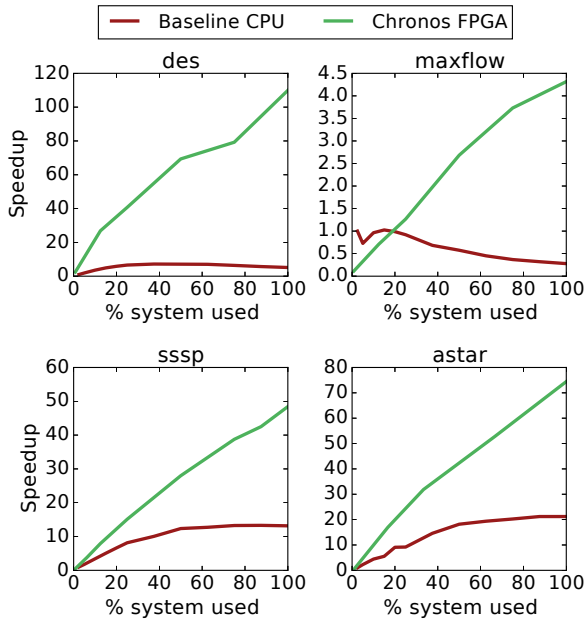


Figure 10. Scalability of Baseline CPU and Chronos implementations of the ordered algorithms. For CPU, 40 threads = 100% system. For Chronos, 100% corresponds to the number of tiles in Table 3. Speedups are normalized to the serial CPU version.

App	Tiles	FPGA vs serial CPU			CPU	Overall
		1 task[1]	1 tile	all tiles	scaling	speedup
`des`	8	2.45×	26.8×	109.9×	7.2×	15.3×
`maxflow`	8	0.11×	0.7×	4.3×	1.0×	4.3×
`sssp`	16	0.24×	3.8×	48.4×	13.3×	3.6×
`astar`	6	0.58×	16.9×	74.4×	21.2×	3.5×

[1]A single concurrent task running on a single PE.

Table 3. Performance and scalability of Chronos accelerators with specialized PEs vs. the best CPU performance.

maxflow: The Chronos instance that runs a single task at a time is 9× slower than CPU (2.1× faster after accounting for the frequency difference), mainly because the baseline `maxflow` does not use priority queues and hence is not as hampered as in `des`. However, in the baseline implementation, the tasks are large and therefore parallelism is scarce, achieving a maximum speedup of 3% at 6 threads, before crashing down to a 3.7× slowdown at 40 threads, due to overwhelming synchronization costs at higher thread counts.

The Chronos implementation uses more fine grained tasks, which uncovers huge parallelism, with a self-relative scalability of 39.9×, resulting in an absolute speedup of 4.3×.

`maxflow` thus shows the benefits of dividing large unordered transactions into small ordered tasks, as Sec. 3.3 outlined.

sssp: Similar to `des`, baseline `sssp` also uses a priority queue to schedule tasks, which Chronos provides in hardware. Hence, performance with a single concurrent task is 4.5× larger than

the frequency-adjusted 0.05×. But unlike `des`, `sssp` tasks are resilient to order violations. The baseline uses this insight to obtain a 13.3× speedup at 40 threads. However, Chronos scales even further, upto a 202× self-relative speedup, to give a 3.6× performance advantage over the baseline.

astar: `astar` follows a similar pattern to `sssp`. The performance with a single concurrent task is 1.7× slower than the CPU, and both CPU and FPGA versions scale near linearly. But the FPGA can fit significantly more concurrent tasks than the CPU, achieving a speedup of 3.5×.

In conclusion, Chronos FPGA accelerators uncover significantly more parallelism than their baseline CPU implementations, enough to consistently outperform the CPU despite their much lower frequency. Thus, we expect that high-frequency ASIC versions would achieve even higher speedups.

Comparison with Swarm: We do not compare against Swarm because Swarm was evaluated only in simulation and no actual hardware exists. However, the self-relative scalabilities reported in Swarm papers [33, 34, 59] are similar to the ones we report here. Therefore, given a similar sized Swarm system for each application as in Table 1 and Table 3, we would expect performance to be similar.

6.2 Chronos on non-speculative algorithms

Fig. 11 shows the scalability of non-speculative `color`. Chronos achieves a self-relative scalability of 45×. But the baseline also scales to 9.1×. Though Chronos uncovers more parallelism, it is not sufficient to make up for the 19× penalty in frequency, so the FPGA version is 2.9× slower than the CPU overall. This result shows that Chronos is not necessarily profitable when the software version has easy parallelism (i.e., simple synchronization and sufficient scalability).

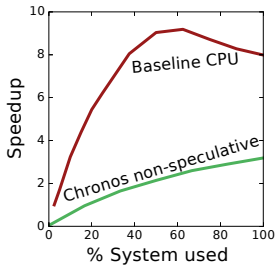


Figure 11. Scalability of unordered `color`.

6.3 Analysis of system efficiency

To analyze the efficiency of speculative execution, we look at how each PE spends its cycles for the four speculative applications. Fig. 12 breaks down PE cycles into those spent running *(i)* tasks that are ultimately committed or *(ii)* tasks that later aborted or, for the no-rollback applications, tasks that performed useless work; and cycles where the PE was stalled due to *(iii)* a full commit queue or *(iv)* the task queue not having any task to dispatch.

Fig. 12 reveals two insights. First Chronos spends most cycles on tasks that ultimately commit. Only 11% of cycles are spent on aborted or useless work overall. Second, the commit queue size, i.e., the number of tasks that can be speculated ahead, moderately limits performance on applications without the no-rollback optimization.

Impact of the no-rollback optimization: We have also generated Chronos accelerators for `sssp` and `astar` without the no-rollback optimization. Due to the higher area requirement of enforcing rollback (Sec. 4.4), we were only able to fit 8 tiles for `sssp` (compared to 16). As a result, the performance of with-rollback versions are 2.3× slower for `sssp` and 4.1× slower for `astar`. The slowdown for `astar` is because `astar` with rollback suffers from large commit queue stalls.

Queue utilization: Fig. 13 shows the average number of task and commit queue entries used across the system by each speculative application. Each tile has a 4K-entry task queue, with the number of tiles specified in Table 3. Large task queues are important for `sssp` and `astar`, which use more than 4K entries on average.

Fig. 13 also shows the commit queue utilization. Since `sssp` and `astar` no-rollback versions do not use the commit queue, this graph shows the results for versions with rollback. All applications use 700–1500 commit queue slots across all tiles on average, showing that applications need a large window of speculation to uncover enough parallelism.

Benefits of specialization: Fig. 14 quantifies the benefits of using application-specific PEs over general-purpose RISC-V soft-cores. Overall, these results show that application-specific PEs have a 4–11× performance advantage.

Estimated ASIC performance: Finally, we use our FPGA prototype to evaluate the benefits of an ASIC Chronos implementation. We estimate that an ASIC RISC-V Chronos implementation could run at 2 GHz, a 16× higher frequency than the FPGA prototype's 125 MHz. To emulate this higher

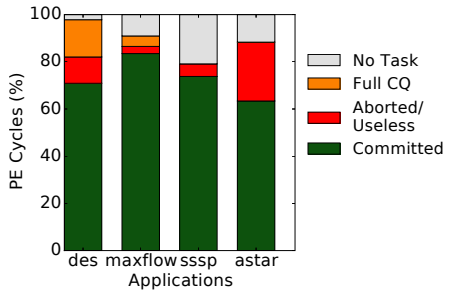


Figure 12. Breakdown of aggregate PE cycles in speculative Chronos variants.

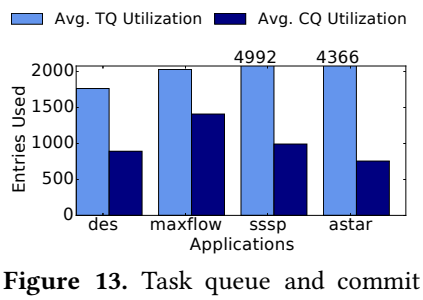


Figure 13. Task queue and commit queue utilization (over all tiles) of each application.

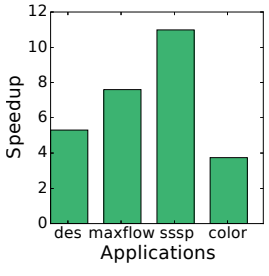


Figure 14. Performance advantage of using application-specific PEs over RISC-V soft-cores.

	Available	TQ	CQ	TSB	Cache	des	maxflow	sssp	astar	color
LUTs (K)	895	17	12	0.5	12	7	11	4	10	7
FFs (K)	1790	6	8	0.3	12	7	6	4	10	8
BRAM	1680	38	5	-	72	-	-	-	-	-
URAM	800	-	-	-	64	-	-	-	-	-

Table 4. Per-tile FPGA resource consumption for each of the framework components and application-specific PEs

frequency, we throttle DDR memory bandwidth by 1/16th, since off-chip bandwidth would not change with frequency.

We find all applications except `color` are not bandwidth-bound and the 2 GHz ASIC achieves a 16× performance improvement over the 125 MHz FPGA (the FPGA prototype has a memory bandwidth of about 50 GB/s). For `color`, the improvement is limited to 13.7×. Thus, compared to the CPU baseline, an ASIC RISC-V Chronos would achieve speedups ranging from 4.7× (`color`) to 244.8× (`des`). Compared to having specialized PEs on an FPGA, speedups would range from 1.5× (`sssp`) to 3.7× (`color`).

6.4 Analysis of implementation costs

Lines of code: Chronos makes it simple to design custom accelerators to extract speculative parallelism. The Chronos framework components take over 20000 lines of SystemVerilog. By contrast, each application is much simpler: `sssp` takes just 100 lines, `des`, `maxflow`, and `color` around 300 lines, and `astar` is around 600 lines.

FPGA utilization: Table 4 shows the FPGA resource consumption of each framework component and PE. Overall, we observe that, while the framework components consume substantial resources, they are comparable to those of PEs, which are very simple.

7 Additional Related Work

Transactional memory on accelerators: Prior work has demonstrated HTM systems on FPGAs [8, 47]. However, they do not target application acceleration using FPGAs, and instead focus on implementing a prototype with soft cores where conflict detection is achieved by augmenting a coherence protocol. Unfortunately, for high-throughput FPGA accelerators, the overheads of a coherence protocol are not desirable.

Ma et al. [42] is the only system that targets FPGA acceleration using TM. However, they do not use an on-chip cache, and hence suffers from reduced performance. Further, while they use priority scheduling to reduce useless work, they do not support strict order constraints among tasks, only unordered transactions.

Kilo TM [18, 19] proposes to implement HTM on GPUs without using cache coherence. Instead, it uses value-based conflict detection, relying on a post-completion validation phase where read values are re-read to detect conflicts. This technique is expensive (e.g., requiring logging of read values) and is restricted to lazy version management, which makes it hard to support speculative forwarding, a key feature for Chronos.

Accelerators for graph algorithms: Numerous other work have also proposed accelerators for graph algorithms, both for FPGA [14, 40] and ASIC [23, 49]. However, none of them support strict task ordering, and as a result resort to less work-efficient algorithms like Bellman-Ford for `sssp`.

Simulation accelerators: Prior work in parallel discrete event simulation has proposed accelerators for different aspects of the Time Warp protocol. The Rollback chip [17] accelerates the speculative versioning and rollback process, but leaves other aspects such as conflict detection to software. Rahman et al. [51] implement a discrete event simulation accelerator on an FPGA. This uses a centralized design that shows why Chronos's distributed, high-throughput approach is crucial: its single event queue saturates around 0.15 events per cycle, a 50× lower task throughput than a 16-tile Chronos system. Moreover, Rahman et al. evaluated their design using a microbenchmark with long tasks and do not explore how to accelerate actual applications. Hence, they do not consider subtle issues that arise when doing so, such as dealing with limited on-chip queue capacity.

FPGAs have also been used to accelerate architectural simulation. RAMP [60, 61] simulates multicore systems, and FireSim [38] simulates large, scale-out clusters. These systems use non-speculative CMB-style simulation, which may limit parallelism, and could benefit from Chronos's techniques.

8 Conclusion

We have presented Chronos, the first framework to build accelerators for applications with ordered speculative parallelism. Chronos makes speculative execution cheap by relying on SLOT, a new execution model that limits tasks to access a single read-write object, avoiding the need for cache coherence.

We implement Chronos on an FPGA and use it to accelerate several challenging applications in graph analytics and simulation. We deploy these accelerators on commodity AWS FPGAs, where we demonstrate 5.4× gmean speedup for the same applications over their software-parallel versions.

Acknowledgments

We sincerely thank Mark Jeffrey, Victor Ying, Joel Emer, Po-An Tsai, Anurag Mukkara, Guowei Zhang, Quan Nguyen, Hyun Ryong Lee, Keiko Yamaguchi, Shuichi Konami, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CAREER-1452994 and SHF-1814969, NSF/SRC grant E2CDA-1640012, and by a Sony research grant.

A Artifact Appendix

A.1 Abstract

Our artifact consists of the source code for the Chronos FPGA acceleration framework; pre-compiled FPGA images for our evaluated configurations (to facilitate a quick evaluation); and scripts to set up the development environment, compile the images from source code, run the experiments in the paper, and regenerate the graphs.

This appendix describes how to use Chronos to reproduce the paper's results, and explains how to set up and run other Chronos configurations and experiments. All experiments are run on the Amazon AWS f1.2xlarge instance, configured using the Amazon-provided FPGA Developer AMI.

A.2 Artifact check-list (meta-information)

- **Compilation:** Xilinx Vivado, GNU RISC-V embedded GCC compiler.
- **Run-time environment:** Amazon AWS FPGA instance.
- **Hardware:** Xilinx UltraScale VU9P.
- **How much disk space required (approximately)?:** 2GB.
- **How much time is needed to prepare workflow (approximately)?:** Approx. 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 2 weeks to reproduce the full results from scratch, or 2 hours if using the precompiled images. The tutorials (Sec. A.7) take about 2 days each, or 2 hours if using precompiled images.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GPL v2.
- **Archived (provide DOI)?:** 10.5281/zenodo.3558760

A.3 Description

A.3.1 How delivered

Our artifact can be downloaded from https://doi.org/10.5281/zenodo.3558760 as a .zip file.

A.3.2 Hardware dependencies

Chronos is designed to run on an Amazon AWS f1.2xlarge instance configured with the Amazon FPGA Developer AMI.

A.3.3 Software dependencies

The main dependence is Xilinx Vivado 2018.2, which comes with the FPGA Developer AMI. The RISC-V Chronos variant relies on the GNU RISC-V embedded GCC compiler.

A.3.4 Data sets

For small, testing runs, we include scripts to generate synthetic datasets. The experiments in the paper use large, publicly available datasets from other projects. Since datasets are large and publicly available, they are not included directly in the artifact code. Instead, the artifact includes scripts to download these datasets. These datasets are also archived, with the DOI 10.5281/zenodo.3563178.

A.4 Installation

1. Launch an AWS f1.2xlarge instance using the Amazon FPGA Developer AMI. Log into the instance.
2. Extract the Chronos artifact .zip file, and navigate to its base directory.
3. Run `source install.sh`. This will clone the Amazon FPGA SDK repository and install the necessary drivers.
4. Run `aws configure` to set up the instance with your AWS credentials.
5. (Optional) Install the GNU RISC-V embedded GCC compiler within the instance (https://xpack.github.io/riscv-none-embed-gcc/). This step is optional because the distribution already includes pre-compiled RISC-V binaries necessary for the workflow.

A.5 Experiment workflow

We provide an automated workflow to validate the main results in the paper from scratch. Note that this process involves synthesizing multiple Chronos instances for each application, a process that takes about two weeks to complete.

To facilitate a quick evaluation, we also provide precompiled FPGA images of the Chronos instances; when using these images, reproducing the results takes about two hours.

The `cl_chronos/validation/scripts/` directory contains the necessary scripts to validate the results from the paper. The full process is explained in comments in the master script `run_validation.py`.

To run all experiments from scratch, run:

```
python run_validation.py
```

To run all experiments with precompiled images, run:

```
python run_validation.py --precompiled
```

This will download a list of precompiled image IDs from a shared S3 bucket and run the rest of the workflow.

Sec. A.7 includes two smaller tutorials using Chronos, which can be completed in about 2 hours.

A.6 Evaluation and expected result

Running `run_validation.py` would generate all evaluation plots (Figures 10-14).

A.7 Experiment customization

This section provides two smaller tutorials on using Chronos. First, we illustrate the SLOT programming model using a sample application running on a Chronos instance with RISC-V soft cores. Second, we describe how to generate Chronos instances with specialized cores.

Before starting either tutorial, run `source aws_setup.sh` to configure the necessary environment variables and to define the `$CL_DIR` environment variable to point to the `cl_chronos` subdirectory. Please see `README.txt` here for more detailed information, including topics not covered in this workflow, such as how to simulate Chronos RTL and how to debug Chronos.

A.7.1 Tutorial 1: Chronos using RISC-V soft cores

Step 1: Generate a test graph.

The graph_gen tool can be used to generate test graphs to test our implementation of `sssp`.

```
cd $CL_DIR/tools/graph_gen
make
./graph_gen sssp grid 20
```

This generates a 20x20 grid graph with random weights.

Step 2: Synthesize a Chronos image with RISC-V soft cores.

The output of this step is an Amazon FPGA Image ID (AGFI-ID) that can loaded into the FPGA. This step will take about 8 hours to complete. If you'd like to skip this step, you can instead use the pre-synthesized FPGA image with the AGFI-ID (`agfi-02159d0614fb731a9`).

1. Configure Chronos to use RISC-V cores.
   ```
   cd $CL_DIR/design/
   ./scripts/gen_cores.py riscv
   ```
2. Run synthesis
   ```
   cd $CL_DIR/build/scripts
   ./aws_build_dcp_from_cl.sh
   ```
   This script launches a Vivado synthesis/place-and-route job. The output of this process is a placed-and-routed design, produced at:
   ```
   $CL_DIR/build/checkpoints/to_aws/
      <timestamp>.Developer_CL.tar
   ```
3. Create an FPGA image. (The commands below follow the standard instructions on how to generate a runnable FPGA image from the placed-and-routed design, at https://github.com/aws/aws-fpga/blob/master/hdk/README.md#step3.)
   First, copy the design file to a location in Amazon S3:
   ```
   aws s3 cp $CL_DIR/build/checkpoints/to_aws/
     <timestamp>.Developer_CL.tar <s3_location>.tar
   ```
   Then, create the FPGA image
   ```
   aws ec2 create-fpga-image --name <name>
     --input-storage-location Bucket=<s3_bucket>,
      Key=<location_in_s3> --logs-storage-location
      Bucket=<s3_bucket_name>, Key=<location_in_s3>
   ```
   Running this command generates an AGFI-ID that can be used to load the image into the FPGA.

Step 3: Compile `sssp` RISC-V code.

This step requires the RISC-V embedded GCC compiler. You can skip this step by using the precompiled binaries from `$CL_DIR/riscv-code/binaries` in the next step.

To build `sssp` from source, run:

```
cd $CL_DIR/riscv-code/sssp
make
```

Step 4: Run `sssp` on the FPGA.

First load the generated image into the FPGA (This command may have to be run twice the first time it is loaded).

```
sudo fpga-load-local-image -S 0 -I <agfi-id>
```

Next, compile and run the `test_chronos` program that transfers the input graph to the FPGA, collects results, and analyzes performance.

```
cd $CL_DIR/software/runtime
make
./test_chronos --n_tiles=1 sssp <sssp_input_file>
   <sssp_riscv_binary>
```

A.7.2 Tutorial 2: Chronos with specialized cores

The RTL code for specialized applications can be found in `$CL_DIR/design/apps/`. For this example, we will again use `sssp`; other applications are similar.

To generate a Chronos instance with these cores, run:

```
./scripts/gen_cores.py sssp
```

The rest of the steps are same as in Tutorial 1, except that the `test_chronos` script does not take a `<sssp_riscv_binary>` argument.

A precompiled `sssp` Chronos instance is also available with the AGFI-ID = `agfi-0d3750b6360762108`.

A.7.3 Customized configurations and applications

Customizing Chronos parameters: The file `config.sv` contains the configuration parameters of Chronos. These include the number of tiles, the sizes for various queues and cache parameters.

Porting new applications: The first step in porting a new application is to break the application down into SLOT tasks (single-object tasks ordered using timestamps). Initially, these tasks can be expressed as software functions and run on a Chronos instance with RISC-V cores.

Once the SLOT implementation is verified, a specialized core can be designed for each task. Please refer to the script `$CL_DIR/design/scripts/gen_cores.py` on how to integrate new specialized cores into the Chronos workflow.

References

[1] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded transactional memory. In *Proc. of the 11th IEEE intl. symp. on High Performance Computer Architecture (HPCA-11).*

[2] Richard J. Anderson and João C. Setubal. 1992. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. of the 4th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA).*

[3] AWS FPGA Hardware and Software Development Kit. 2017. https://github.com/aws/aws-fpga.

[4] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. of the IEEE Infocom 2000.*

[5] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP).*

[6] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. Performance pathologies in hardware transactional memory. In *Proc. of the 34th annual Intl. Symp. on Computer Architecture (ISCA-34).*

[7] Christopher D. Carothers, David Bauer, and Shawn Pearce. 2000. ROSS: A High-performance, Low Memory, Modular Time Warp System. In *Proc. of the 14th Workshop on Parallel and Distributed Simulation (PADS)*.

[8] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. 2011. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proc. of the 16th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*.

[9] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. 2007. A scalable, non-blocking approach to transactional memory. In *Proc. of the 13th IEEE intl. symp. on High Performance Computer Architecture (HPCA-13)*.

[10] K. Mani Chandy and Jayadev Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM* 24, 4 (1981).

[11] Tao Chen, Shreesha Srinath, and G. Edward Batten, Christopher Suh. 2018. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51)*.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proc. of the 47th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-47)*.

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (ISCA-43)*.

[14] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proc. of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[15] C. Demetrescu, A. Goldberg, and D. Johnson. 2006. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9

[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[17] R. M. Fujimoto, J.-J. Tsai, and G. C. Gopalakrishnan. 1992. Design and evaluation of the rollback chip: special purpose hardware for Time Warp. *IEEE Trans. Comput.* 41, 1 (1992).

[18] Wilson W. L. Fung and Tor M. Aamodt. 2013. Energy Efficient GPU Transactional Memory via Space-Time Optimizations. In *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-46)*.

[19] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware Transactional Memory for GPU Architectures. In *Proc. of the 44th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-44)*.

[20] Epifanio Gaona-Ramirez, Rubén Titos-Gil, Juan Fernandez, and Manuel E. Acacio. 2010. Characterizing energy consumption in hardware transactional memory systems. In *Proc. of the 22nd symp. on Computer Architecture and High Performance Computing (SBAC-PAD 22)*.

[21] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. 2003. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. In *Proc. of the 9th IEEE intl. symp. on High Performance Computer Architecture (HPCA-9)*.

[22] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing* 7, 4 (2008).

[23] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics. In *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-49)*.

[24] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proc. of the 8th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*.

[25] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture (ISCA-31)*.

[26] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional memory. *Synthesis Lectures on Computer Architecture* (2010).

[27] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*.

[28] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*.

[29] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture (ISCA-20)*.

[30] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2013. Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies. In *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*.

[31] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. 1987. Time Warp Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*.

[32] David R. Jefferson. 1985. Virtual time. *ACM TOPLAS* 7, 3 (1985).

[33] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-centric execution of speculative parallel programs. In *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-49)*.

[34] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48)*.

[35] Mark C. Jeffrey, Victor A. Ying, Suvinay Subramanian, Hyun Ryong Lee, Joel Emer, and Daniel Sanchez. 2018. Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51)*.

[36] Mark T. Jones and Paul E. Plassmann. 1993. A Parallel Graph Coloring Heuristic. *SIAM J. Sci. Comput.* 14, 3 (1993).

[37] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance

Analysis of a Tensor Processing Unit. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[38] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proc. of the 45th annual Intl. Symp. on Computer Architecture (ISCA-45)*.

[39] Jure Leskovec and Andrej Krevl. 2014. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[40] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. 2017. Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[41] Kyle Locke. 2011. Parameterizable Content-Addressable Memory. *Xilinx Application Note* (2011).

[42] Xiaoyu Ma, Dan Zhang, and Derek Chiou. 2017. FPGA-Accelerated Transactional Execution of Graph Workloads. In *Proc. of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[43] Steve Margerm, Amirali Sharifian, Apala Guha Guha, and Gilles Shriraman, Arrvindh Shriraman Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51)*.

[44] Ulrich Meyer and Peter Sanders. 1998. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proc. of the 6th Annual European Symposium on Algorithms (ESA)*.

[45] Vincent Mirian and Paul Chow. 2012. FCache: A System for Cache Coherent Processing on FPGAs. In *Proc. of the 2012 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

[46] Kevin Moore, Jayaram Bobba, Michelle Moravan, Mark D. Hill, and David Wood. 2006. LogTM: Log-based transactional memory. In *Proc. of the 12th IEEE intl. symp. on High Performance Computer Architecture (HPCA-12)*.

[47] Njuguna Njoroge, Jared Casper, Sewook Wee, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. 2007. ATLAS: A Chip-multiprocessor with Transactional Memory Support. In *Proc. of the conf. on Design, Automation and Test in Europe (DATE)*.

[48] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[49] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (ISCA-43)*.

[50] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[51] Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. 2017. PDES-A: A Parallel Discrete Event Simulation Accelerator for FPGAs. In *Proc. of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*.

[52] Ravi Rajwar and James R Goodman. 2002. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*.

[53] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proc. of the Intl. Conf. on Supercomputing (ICS'05)*.

[54] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48)*.

[55] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture (ISCA-22)*.

[56] SpinalHDL. 2018. A FPGA friendly 32 bit RISC-V CPU implementation. https://github.com/SpinalHDL/VexRiscv.

[57] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture (ISCA-27)*.

[58] Suvinay Subramanian. 2018. *Architectural Techniques to Unlock Ordered and Nested Speculative Parallelism.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[59] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An execution model for fine-grain nested speculative parallelism. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[60] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. 2010. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *Proc. of the 47th Design Automation Conf. (DAC-47)*.

[61] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C Hoe, Derek Chiou, and Krste Asanovic. 2007. RAMP: Research accelerator for multiple processors. *IEEE Micro* 27, 2 (2007).

[62] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. 2007. A Practical FPGA-based Framework for Novel CMP Research. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA)*.

[63] H. J. Yang, K. Fleming, M. Adler, and J. Emer. 2014. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. In *Proc. of the Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[64] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th IEEE intl. symp. on High Performance Computer Architecture (HPCA-13)*.

[65] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. 1998. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proc. of the 4th IEEE intl. symp. on High Performance Computer Architecture (HPCA-4)*.