**QUEEN'S
UNIVERSITY
BELFAST**

Hyperqueues: Design and Implementation of Deterministic Concurrent Queues

Vandierendonck, H., & Nikolopoulos, D. S. (2019). Hyperqueues: Design and Implementation of Deterministic Concurrent Queues. *ACM Transactions on Parallel Computing (TOPC)*, 6(4), [23].
<https://doi.org/10.1145/3365660>

Published in:
ACM Transactions on Parallel Computing (TOPC)

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
[Link to publication record in Queen's University Belfast Research Portal](#)

Publisher rights
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Hyperqueues: Design and Implementation of Deterministic Concurrent Queues (Postprint)

HANS VANDIERENDONCK, Queen's University Belfast
DIMITRIOS S. NIKOLOPOULOS, Queen's University Belfast

The hyperqueue is a programming abstraction for queues that results in deterministic and scale-free parallel programs. Hyperqueues extend the concept of Cilk++ hyperobjects to provide thread-local views on a shared data structure. While hyperobjects are organized around private local views, hyperqueues provide a shared view on a queue data structure. Hereby, hyperqueues guarantee determinism for programs using concurrent queues. We define the programming API and semantics of two instances of the hyperqueue concept. These hyperqueues differ in their API and the degree of concurrency that is extracted. We describe the implementation of the hyperqueues in a work-stealing scheduler and demonstrate scalable performance on pipeline-parallel benchmarks from PARSEC and StreamIt.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; **Concurrent algorithms**; • **Software and its engineering** → **Software performance**; • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: hyperqueue

ACM Reference Format:

Hans Vandierendonck and Dimitrios S. Nikolopoulos, 2019. Hyperqueues: Design and Implementation of Deterministic Concurrent Queues. *ACM Trans. Parallel Comput.* 6, 4, Article 23 (November 2019), 36 pages. DOI: 10.1145/3365660

1. INTRODUCTION

Ubiquitous parallel computing aims to make parallelism accessible to a wide variety of programming areas without putting quality of software at risk. It is understood that a *task abstraction*, where a task is a unit of computation, is a key element as it allows programmers to focus on the “What?” instead of the “How?”. Task parallel programming languages use a task abstraction to simplify the construction of parallel programs. They support common parallel patterns such as parallel DOALL loops and divide-and-conquer parallelism using the addition of a small number of keywords or annotations to the program. Moreover, task parallelism supports the definition of scale-free programs, where the specific dimensions of the hardware, such as the number of threads, are not hard-coded in the source code.

The task abstraction moreover helps to provide determinism [Bocchino et al. 2011; Jenista et al. 2011; Cavé et al. 2011; Bauer et al. 2012], which describes that each execution of a parallel program provides the same or equivalent outcomes. Several programming patterns require dedicated language support to achieve determinism. One such common pattern is a *reduction*, which is the application of an associative op-

This work is partly supported by the European Community's Seventh Framework Programme (FP7/2003-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744), under the TEXT project (grant agreement 261580) and by the United Kingdom EPSRC GEMSCCLAIM project (grant agreement EP/K017594/1). Kallia Chronaki contributed to the design, implementation and evaluation of the reducing hyperqueue [Vandierendonck et al. 2013].

Author's addresses: H. Vandierendonck and Dimitrios S. Nikolopoulos, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2019 Copyright held by the owner/author(s). 1539-9087/2019/11-ART23 \$15.00

DOI: 10.1145/3365660

erator to reduce a set of values into one. Various languages support reductions through thread-local storage [OpenMP 2013]. This provides deterministic behavior on the condition that the reduction operator is also commutative. Hyperobjects [Frigo et al. 2009] provide a more general solution to handling shared data structures in a deterministic manner. There are three types of hyperobjects: reducers, which support reductions, holders, which support thread-local storage, and splitters, which support backtracking patterns.

Other program idioms build on data structures for which it is generally hard to define deterministic behavior. Pipeline parallelism assumes that data items are sent through a succession of *stages*. A common implementation of pipeline parallelism assumes concurrent queues and an explicit threading model. Such designs, however, are not deterministic [Payer et al. 2011]. Task parallel models may provide support for pipeline parallelism by encoding stages as tasks. These models, however, restrict the pipeline structure. Task parallel programming languages that encode dataflow dependencies among tasks may express pipelines [Vandierendonck et al. 2011a], but stages must produce a fixed number of items. Cilk-P extends Cilk to pipelines but provides no support for buffering or chunking data elements [Lee et al. 2013; ?]. Moreover, these models restrict producer-consumer dependencies among “sibling” tasks, whereas explicit threading allows queue access from any location in a program.

This paper investigates extensions to task parallel programming languages that support concurrent queues while providing determinism guarantees. We call these extensions *hyperqueues*. Hyperqueues provide *internal determinism* [Blelloch et al. 2012] for calls to the queue methods (push, pop, peek and empty check). Internal determinism states that, for every two executions of the program, corresponding calls to the queue methods have identical arguments and return values. An immediate consequence is that a serial execution of the program is sufficient to analyze correctness of a program [Blelloch et al. 2012]. As such, a parallel execution pops values from the hyperqueue in the same order as sequential execution. Hyperqueues thus implement concurrent queues that are semantically equivalent to a single-producer, single-consumer queue, yet admit out-of-order execution.

This paper presents two implementations of hyperqueues that differ in their API and in the concurrency that can be derived from the API. Depending on the properties of programs, either one or the other may be used. A first implementation, published previously [Vandierendonck et al. 2013], supports concurrent execution of producers but executes consumers serially. It uses polling to delay the consumer until its data is available in the queue. We henceforth call it a *reducing hyperqueue* because it builds on Cilk++ reducers [Frigo et al. 2009] to enable concurrent execution of producers.

A second implementation, the *counted hyperqueue*, is described in this article for the first time. It exposes a higher degree of concurrency as it allows concurrent execution of producers and consumers. It moreover starts execution of consumer tasks only when the input data is available. This makes polling unnecessary. The added concurrency makes an important contribution to parallel scalability for a wide class of streaming applications. It can, however, be exposed only by providing additional application knowledge through the API. The counted hyperqueue requires that tasks are annotated with the precise number of elements that will be pushed or popped by each task. As such, we can assign sequence numbers to every element passed through the queue and we can map these sequence numbers to tasks. This allows the runtime system to match producers with consumers by looking at overlap in the sequence numbers of queued elements.

Using hyperqueues, we parallelize several benchmarks with less programming effort than using POSIX threads or Threading Building Blocks (TBB). The hyperqueue, moreover, obtains the same or better performance.

The remainder of this paper is organized as follows. Section 2 discusses the programming model. Section 3 discusses the implementation of a list-based polling hyperqueue. Section 4 discusses the implementation of a counted hyperqueue supporting concurrent consumers. Section 5 discusses various properties of the hyperqueues. Then, Section 6 presents programming techniques. We present an experimental evaluation in Section 7. Finally, Section 8 discusses related work and Section 9 concludes this paper.

2. PROGRAMMING MODEL

We define and implement hyperqueues in the Swan programming model [Vandierendonck et al. 2011b], which extends Cilk with dataflow concepts. We briefly describe Swan to provide context.

Swan inherits the **spawn** and **sync** keywords from Cilk. The **spawn** keyword indicates that calling a task may occur in parallel with the continuation of the calling procedure. The **sync** keyword blocks a procedure until all tasks that were spawned from the same procedure have finished execution.

The parallelism indicated by **spawn** can be altered by expressing dataflow dependencies. These dataflow dependencies describe the inputs and outputs to tasks, which essentially describes the task's side effects. The runtime system collects these side effects as tasks are spawned and computes the task dependency graph on the fly. Variables that express dataflow dependencies are defined with the **versioned** keyword which attaches facilities to them for tracking inter-task dependencies. Also, automatic memory management is applied to versioned objects to break write-after-read dependencies. Versioned variables may be used as procedure arguments provided they are cast to type **indep**, **outdep** or **inoutdep**, which describes side effects of reading, writing or both. These side effects restrict the parallelism in programs. Consider a program where first a task is spawned where a versioned variable is supplied as an **outdep** argument, and a second spawned task takes the same variable as an **indep** argument. The dataflow declares that the second task requires the outcome of the first task. The Swan runtime system recognizes this dependency and enforces during execution that these two tasks never execute concurrently.

Dataflow dependencies allow to encode complex dependency patterns with relatively simple constructs. Pipeline parallelism is just one of the many patterns that the dependency graph may take in dataflow models [Vandierendonck et al. 2011b]. Moreover, task dataflow systems provide memory management that greatly simplifies writing pipeline parallel programs [Vandierendonck et al. 2011a].

2.1. Hyperqueue Programming Model

Hyperqueues are a programming abstraction for queues. A queue is an ordered sequence of values. Values are added to the tail of the sequence using a *push* method. Values are removed from the head of the sequence using a *pop* method. The *push* method always succeeds (we assume that the queue's buffer space can grow as needed). The *pop* and *peek* methods may only be called on a non-empty queue. The *empty* method returns a boolean indicating whether the queue is empty.

We define a hyperqueue as a special object in our programming language that models a single-producer, single-consumer queue. Its implementation allows tasks to concurrently push and pop values without breaking the semantics of a single-producer, single-consumer queue, and without breaking the serializability of the parallel program. We propose two distinct implementations. The first implementation, the **reducing hyperqueue**, allows concurrency between producers, but only a single consumer can be active at a time. The **counted hyperqueue** increases parallelism by supporting multiple concurrent consumers. To enable this concurrency, it is necessary

Table I. Three ways to construct pipeline-parallel programs in the Swan language.

Operation	Dataflow objects	Reducing hyperqueue	Counted hyperqueue
Type	versioned object;	hyperqueue queue;	counted.hyperqueue queue;
Producer annotation	outdep	pushdep	suffixdep (with length)
Produce “v”	object = v;	queue.push(v)	queue.push(v)
Consumer annotation	indep	popdep	prefixdep (with length)
Consume value	v = object;	v = queue.pop();	v = queue.pop();
Peek value	v = object;	v = queue.peek();	v = queue.peek();
Empty check	One item passed in	queue.empty()	queue.empty()

to inform the runtime of how many elements will be popped by a consumer. The annotations and methods supported by our programming model for implementing pipeline parallelism using dataflow objects, reducing hyperqueues and counted hyperqueues are summarized in Table I.

Reducing hyperqueues are defined as variables of type **hyperqueue**, which takes a type parameter to describe the type of the values stored in the queue. Hyperqueues may be passed to procedures provided they are cast to a type that describes the access mode of the procedure. This type can be **pushdep**, **popdep** or **pushpopdep**, to indicate that the spawned procedure may only push values on the queue, that it may only pop values from the queue, or that it may do both. A task with **push** access mode is not required to push any values, nor is a task with **pop** access mode required to pop all values from the queue. A hyperqueue may be destroyed with values still inside.

The reducing hyperqueue allows tasks to both produce and consume elements on the same queue (through the **pushpopdep** access mode). Concurrent execution of consumers is not supported for the reducing hyperqueue as it is impossible to guarantee deadlock freedom, or to adhere to the serial semantics. Indeed, if two tasks pop elements from the same queue, it is impossible to tell which elements the second task will pop without knowing the number of elements that will be popped by the first task. The counted hyperqueue lifts this restriction.

Counted hyperqueues are defined as variables of type **counted.hyperqueue**. A type parameter describes the type of the values stored in the queue. Counted hyperqueues support two access modes: **prefixdep** and **suffixdep**. The **prefixdep** access mode assigns a fixed-length prefix of the queue to the spawned task. It is assumed that the spawned task will consume all values in the prefix. The **suffixdep** access mode assigns a suffix of given length to the spawned task, allowing it to produce values. It is assumed that the task will produce as many values as given by the suffix length.

It is an error if a **prefixdep** is created that consumes more elements than what has been promised by producers. This cap is required to avoid deadlock in serial execution and also to assure internal determinism in parallel execution.

Contrary to the reducing hyperqueue, counted hyperqueues make strong assumptions on the number of values pushed or popped by tasks. This enables the runtime implementation to execute multiple consumers concurrently. Moreover, the counted hyperqueue ensures that all required values have been pushed to the queue before initiating a consuming task. In contrast, the hyperqueue will perform busy waiting (polling) if the queue is temporarily empty.

Besides a type parameter, the **prefixdep** type takes a second parameter that indicates whether the spawned task executes in *blocking* or *non-blocking* mode. Blocking mode implies that the task can execute only when all of its input data is available. In non-blocking mode, tasks may be initiated before all data is available. These tasks may however not consume elements from the queue themselves. Instead, they must spawn other tasks in blocking mode. This construct is useful to launch consumers in parallel.

<pre> 1 struct data { ... }; 2 void consumer(popdep<data> queue) { 3 while(!queue.empty()) { 4 data d = queue.pop(); 5 // ... operate on data ... 6 } 7 } 8 void producer(pushdep<data> queue, 9 int start, int end) { 10 if (end-start <= 10) { 11 for(int n=start; n < end; ++n) { 12 data d = f(n); 13 queue.push(d); 14 } 15 } else { 16 for(int n=start; n < end; n += 10) { 17 int to = min(n+10,end); 18 spawn producer(queue, n, to); 19 } 20 sync; 21 } 22 } 23 void pipeline(int total) { 24 hyperqueue<data> queue; 25 spawn producer(pushdep<data>queue, 26 0, total); 27 spawn consumer(popdep<data>queue); 28 sync; 29 } </pre>	<pre> 1 struct data { ... }; 2 void consumer(prefixdep<data,block> queue) { 3 while(!queue.empty()) { 4 data d = queue.pop(); 5 // ... operate on data ... 6 } 7 } 8 void producer(suffixdep<data> queue, 9 int start, int end) { 10 if (end-start <= 10) { 11 for(int n=start; n < end; ++n) { 12 data d = f(n); 13 queue.push(d); 14 } 15 } else { 16 for(int n=start; n < end; n += 10) { 17 int to = min(n+10,end); 18 spawn producer(queue.suffix(to-n), n, to); 19 } 20 sync; 21 } 22 } 23 void pipeline(int total) { 24 counted.hyperqueue<data> queue; 25 spawn producer(queue.suffix(total), 0, total); 26 spawn consumer(queue.prefix(total/2)); 27 spawn consumer(queue.prefix(total-total/2)); 28 sync; 29 } </pre>
(a) Reducing hyperqueue	(b) Counted hyperqueue

Fig. 1. A simple pipeline-parallel program expressed with hyperqueues.

A simple 2-stage pipeline using the reducing hyperqueue is shown in Figure 1 (a). The procedure *pipeline* at line 23 creates a hyperqueue object where elements of the queue are of type *struct data*. It then spawns a procedure *producer* with **pushdep** access mode which will produce data in the queue using the *push* method. The procedure *consumer* is spawned with **popdep** access mode and will consume the data. It may utilize the method *empty* to check whether any data on the queue is pending and the method *pop* to remove data from the head of the queue.

The *empty* method checks if more values are pending in the queue. It is designed such that it mimics the result of sequential execution: the *empty* method returns false only if it is certain that no more values will be added to the queue. If there is a possibility that values will be added that are visible to the task executing the *empty* method, then the *empty* call will block until a definite decision can be made.

Pop must only be called on non-empty queues, as popping elements from an empty queue is an error.

Figure 1 (b) shows the same problem implemented with the counted hyperqueue. As in the case of the reducing hyperqueue, producers may be spawned recursively. In this simple example, there is an immediate mapping between **popdep** and **prefixdep** and between **pushdep** and **suffixdep**. The *empty* method can be used similarly as in the hyperqueue. It differs, however, as it indicates whether the prefix assigned to the calling task has been fully consumed. This does not imply that the queue is empty as a whole.

2.2. Dependencies and Task Scheduling

The Swan runtime system utilizes the queue access modes (**pushdep**, **popdep**, **pushpopdep**, **suffixdep** and **prefixdep**) to decide when a spawned procedure may start execution. This process is similar to how **indep**, **outdep** and **inoutdep** define an execution order between tasks operating on versioned objects [Vandierendonck et al. 2011b]. We separately discuss task scheduling for hyperqueues and counted hyperqueues.

2.2.1. Task Scheduling for Reducing Hyperqueues. The task scheduler enforces the following constraints due to queue access modes (the access modes on all arguments are taken into account when scheduling tasks, including the versioned object access modes):

- (1) Tasks with **pushdep** access mode on the same queue may execute concurrently. The runtime system will use the concept of reduction to manage concurrent pushes and expose the pushed values in serial program order to any consumer.
- (2) A task with **popdep** access mode may execute concurrently with the preceding tasks with **pushdep** access mode on the same queue. This enables concurrent pushes and pops on the queue. The runtime system ensures that pops do not run ahead of pushes.
- (3) A task with **popdep** access mode may initiate execution only when all older tasks with **popdep** access mode on the same queue have completed execution. The rationale is that values are exposed in program order, so the oldest task must perform all its pops before a younger task may perform its pops.
- (4) A task P with **pushdep** access mode may execute concurrently with an older task C with **popdep** access mode. The rationale is that P will create a sequence of values, but this sequence of values is independent of the actual pops performed by C . Moreover, C is not allowed to see any of the values pushed by P because this would violate the serializability of the program. The runtime system will ensure that any values left in the queue when C completes execution will be merged with the values produced by P in program order.

Tasks with **pushpopdep** access mode are scheduled by taking restrictions of both **pushdep** and **popdep** modes into account. The Swan runtime system uses the same machinery to enforce the execution order of tasks with queue dependencies as it does for versioned objects [Vandierendonck et al. 2011b].

In recursive programs, tasks can only spawn child tasks with a subset of the privileges that they hold, i.e., tasks with **pushpopdep** access on a reducing hyperqueue can pass both privileges on that hyperqueue, while tasks with either **pushdep** or **popdep** access mode can pass only the named privilege on the corresponding hyperqueue. This restriction makes it safe to apply the above rules for task scheduling separately to each procedure instance [Pratikakis et al. 2011].

Consider the following program to illustrate these rules:

```

1 hyperqueue<T> queue;
2 spawn A( (pushdep<T>)queue );
3 spawn B( (pushdep<T>)queue );
4 spawn C( (popdep<T>)queue );
5 spawn D( (pushpopdep<T>)queue );
6 spawn E( (pushdep<T>)queue );
7 spawn F( (popdep<T>)queue );
8 sync;
```

Procedure A is the oldest procedure and is immediately ready to execute. B may execute concurrently with A due to case 1. C may execute concurrently with A and B due to case 2. D must wait until C completes due to case 3. E may execute concurrently with A, B, C and D following case 4. Finally, procedure F must wait until D completes

Fig. 2. Key ideas behind the internal representation of the reducing hyperqueue.

due to case 3. F will never start execution prior to E due to the work-first principle (spawned tasks are executed immediately by the spawning thread).

2.2.2. Task Scheduling for Counted Hyperqueues. The counted hyperqueue allows all tasks to execute out of order, except for tasks with **blocking prefixdep** access mode, which may execute only if their input data has been produced.

We recognize whether a consuming task's input data has been produced by tracking the lengths of prefixes and suffixes. Considering that the queue represents a sequence of elements, we assign a *unique sequence number* to each element in the queue. These sequence numbers correspond to the order in which the elements are pushed in the queue assuming the execution order of the serial elision. As such, the sequence number of an element identifies the producing task, where the first task with suffix length p_0 pushes elements in the range $0, \dots, p_0 - 1$, the second task with suffix length p_1 pushes elements in the range $p_0, \dots, p_0 + p_1 - 1$, etc. Similarly, elements are consumed in the same order in serial execution and the link between consuming and producing tasks can be made by matching ranges of sequence numbers.

Note that, while it is also possible to assign sequence numbers to the elements in a reducing hyperqueue, this association can only be made for a producing task when all producing tasks that precede it in serial execution order have completed. The reducing hyperqueue does not specify how many elements the older tasks will push. As such, one must wait until these prior tasks complete to know how many elements they have pushed and to know the next sequence number for the later tasks.

3. REDUCING HYPERQUEUES

The reducing hyperqueue is a concurrent queue designed to provide internal determinism for the arguments and return values of its methods. An example illustrates what we aim to achieve (Figure 2). A queue consists of a linked list of segments that hold the queue contents. Tasks in the program are given access mode to the queue. In this example, Task 1 and Task 2 may push values in the queue, while Task 3 may pop values. We aim to allow parallelism among these tasks:

- (1) Task 3 should be able to execute in parallel to Tasks 1 and 2, restricted only by the availability of values in the queue.
- (2) Tasks 1 and 2 should be able to execute in parallel.
- (3) At any moment in time, only one task should be in flight that is allowed to pop values from the queue.

Importantly, during such parallel execution, the queue must be *internally deterministic*, i.e., it must record the pushed values in the same order as they would be produced

by the serial elision of the program, and the *pop* operation should return the values in the same order as the serial elision. Moreover, calls to the *empty* function should return an empty condition only when the queue would be empty in the serial elision of the program. If the correct state of the queue is uncertain during parallel execution, then the empty call should block until that uncertainty is resolved. Under these conditions, the *empty* function becomes blocking and *pop* remains non-blocking. We require that every call to *pop* is preceded by a call to *empty*.¹

The semantics of the reducing hyperqueue can be realized by extending two prior techniques: dataflow scheduling and Cilk++ hyperobjects. Dataflow scheduling [Vandierendonck et al. 2011b] tracks producer-consumer dependencies between tasks and executes tasks only when the dependencies are resolved. This technique can be used to ensure that no two tasks that aim to consume values from the queue execute in parallel.

The second technique are Cilk++ hyperobjects [Frigo et al. 2009], a linguistic mechanism to provide local views to non-local objects. Non-local objects are accessible from multiple tasks. A particular type of hyperobjects are *reducers*, which implement reduction operations. Reducers are relevant to the hyperqueue due to the equivalence between pushing an element to the end of a queue and list concatenation. The latter is a reduction operation [Frigo et al. 2009].

3.1. Background: Cilk++ Reducers

A reducer [Frigo et al. 2009] is specified as a triple $(R, \otimes, e)$ where R is a set of values, $\otimes$ is an associative binary operator over R and e is an identity element of $\otimes$. In this paper, we are interested in $R = S^*$, the set of all lists consisting of elements of S , $\otimes$ is the list concatenation operator $++$, and $e = \epsilon$, the empty list. This operation describes the push operation on queues as follows: assume $Q \in S^*$, then the operation $push(x)$ transforms Q to $Q ++ (x)$ where (x) signifies the single-element list containing x .

Reducers do not require that the operator $\otimes$ is commutative, i.e., swapping its left and right operands need not compute the same value. This ensures reducers are applicable to our use case.

To explain reducers, we must analyze programs at their smallest granularity of parallel execution: the *strand*. A strand is a maximal sequence of instructions between any pair of parallel statements (**cilk_spawn** and **cilk_sync**). The strands in a Cilk or Swan program form a directed acyclic graph. The strands in any task form a sequence (each task is executed sequentially).

The Cilk++ scheduler performs random work stealing to distribute tasks over processors. In this process, it distinguishes between two types of function call frames: full frames and stack frames. A full frame is a frame that may be executing in parallel to its siblings. A stack frame is frame that is not executing in parallel to its siblings. Stack frames are smaller and cheaper to construct, in part because they execute sequentially and thus do not require locking and heap memory allocation upon construction and destruction. Each processor maintains its own call stack. By construction, the oldest frame on this call stack is a full frame and all other frames are stack frames. In a Cilk++ program where the available parallelism exceeds the number of processors, the majority of invoked functions are executed sequentially and work stealing is a relatively rare event.

Reducers provide each strand with its own view of the reduction variable. Each view is owned by one strand at any time. Views are created and destroyed by the runtime system at key events during the execution of the program. The “special optimiza-

¹An alternative interface would make *pop* blocking, in which case it would not be compulsory to call *empty* before each call to *pop*. This can be achieved with minor changes.

tion” [Frigo et al. 2009] leverages the presence of exactly one full frame on a processor’s call stack and associates views to full frames only. As the majority of the invoked functions are executed sequentially, they can access the view of the closest ancestral full frame.

The implementation of reducers retains three views for each full frame. This is necessary to support non-commutative reductions as the reduction operations need to be applied strictly in the order that they would appear in the serial elision. The main view is the *user* view, which is visible to the user code executed by the current strand on the call stack. The *child* view holds the reductions of completed child strands that were full frames and are “to the left” of the current strand. The *right* view holds the reductions of the completed right siblings. The operations performed on the *left* view must be applied before those on the current strand, and those on the *right* view must be applied after those on the current strand to reflect the order of the serial elision.

We refer to [Frigo et al. 2009] for details on the view management process.

3.2. Reducing Hyperqueue Semantics

The reducing hyperqueue provides two types of view: a view to push and a view to pop (we will relate these views to the children, user and right views of Cilk++ reducers later). The view to push supports parallel push operations by multiple threads. The view to pop allows a thread to pop elements. The key insight behind this distinction is that a push operation only needs to know the tail of the queue and not about the head, while a pop operation requires to know the head, but not the tail. The views of a strand are created in correspondence to the access mode provided to the call of the containing task: a view to pop is created for a queue passed in with **popdep** or **pushpopdep** access; and a view to push is created for a queue passed in with **pushdep** or **pushpopdep** access. If a task has no rights to push or pop, then the corresponding view is missing. A pop operation accesses the queue through the view to pop. The pop operation fails if the view to pop is missing. Push access rights are enforced similarly through the view to push.

3.3. Local and Shared Views

If we consider that a queue is an ordered list, then reducers already enable parallelism between push operations. Moreover, reducers will make the push operations visible in the same order as they would be executed by the serial elision of the program. In the following, we extend reducers to support concurrent pop operations.

A key difference with reducers is that the queue must be shared between two strands in order to enable concurrent push and pop operations. As such, the queue is a shared data structure and a view on the queue contains control variables that govern its correct operation.

It is important to make a distinction between the queue, which is a shared data structure, and the views on the queue, which consist of pointers to the underlying queue. The runtime primarily operates on the views in response to task creation, completion and work stealing, while the application uses the views as a means to access the underlying queue when performing push and pop operations.

We consider a queue implemented as a singly-linked list of segments. Each queue segment is a fixed-size array and holds head and tail indices for this segment. This design supports efficient concatenation of queues in $O(1)$ steps, as well as efficient iteration over the queue elements and an economic storage format. A queue segment contains following variables:

- `head_index`, `tail_index`: head and tail indices in circular buffer
- `size`: the size of the buffer in this segment

- *buffer*: the buffer in this segment
- *next*: a pointer to the next segment in the linked list
- *producing*: a boolean that indicates if additional values may be pushed on the queue segment

A view on a hyperqueue contains two pointers to queue segments, namely to the head and tail of the linked list that is accessible from the view. Thus, in the view (h, t) , h points to the head of a linked list of queue segments, and t points to the last segment in the list.

We distinguish three classes of pointers: *null* pointers are used in empty views; *local* pointers point to an actual queue segment; and *non-local* pointers are symbolic entities that indicate logical linkage between two views, but do not point to any actual segments. Given these types of pointers, we define a *view to pop* as a view where the head is a local pointer. The view to pop thus provides access to a segment of the queue from which elements may be popped. If the head is null or a non-local pointer, then there is no backing storage available to perform a pop operation and it is impossible to pop through this view. A *view to push* is a view where the tail is null, or a local pointer. Thus, either a segment exists, or one can be created. If the tail is a non-local pointer, then access to the queue is prohibited. This may be a temporary condition as the views are modified in response to the creation and completion of tasks.

A view may have both push and pop access rights if it holds two local pointers.

A view where the head and tail pointers are local pointers points to queue segments that can only be accessed by the task holding that view. We call such a view a *local view*. A view where at least one of the head or tail pointers is a non-local pointer is a *shared view*.

Views are created to match the access mode annotations: a view with a local head pointer is created when a task executes with the **popdep** or **pushpopdep** annotations, and a view with a local or null tail pointer is created with a task executes with the **pushdep** or **pushpopdep** annotations.

3.4. View Operations

Four operations are supported on views: creation, move, split, and reduce. Views are created as empty views (holding two null pointers), or local views initialized with a head and tail pointer pointing to the same queue segment. Views may be moved from one location to another simply by copying the pair of pointers to the destination and setting the source to a pair of null pointers (empty view).

The split operation splits a view in two views, one which exposes the head end of the view and another that exposes the tail end of the view. It also creates a unique non-local pointer to indicate that these two views are logically linked (in fact, non-local pointers can only be created by the split operation).

```

1 (view, view) split ( view v ) {
2   let  $p_{NL}$  = unique non-local pointer;
3   return ( (v.head,  $p_{NL}$ ), ( $p_{NL}$ , v.tail) );
4 }
```

Figure 3(a) shows an example of splitting a local view in a pair of shared views. The shared views are linked logically by the non-local pointer, and physically by the fact that $v.tail$ is reachable from $v.head$ when walking the linked list of queue segments that originates in $v.head$. Non-local pointers always appear in pairs.

The split operation plays a central role for hyperqueues as it allows to create two views on the same underlying queue. One view is to be passed to the consumer task (head view) and the other view is to be passed to the producer task (tail view). This way, two tasks having each a distinct view can communicate in a single-producer/single-

Fig. 3. Splitting and reducing views. The *split* operation creates two views on the same queue segment that, when passed on to distinct tasks, allow a concurrent producer/consumer relationship between them.

consumer relationship. The challenge for the runtime system is to manage views in such a way that determinism is retained.

The reduction of views involves two parts: physically linking the linked list of queue segments in the two views, and destroying matching non-local pointers, if present:

```

1 view reduce( view left , view right ) {
2   if ( left is an empty view ) {
3     return right ;
4   } else if ( right is an empty view ) {
5     return left ;
6   } else if ( left . tail is a local pointer ) {
7     // invariant : right . head is a local pointer
8     // invariant : left . tail -> next == null
9     left . tail -> next = right . head ;
10  } else {
11    // invariant : left . tail == right . head
12    destroy non-local pointer left . tail ;
13  }
14  return ( left . head , right . tail ) ;
15 }

```

The two main cases are:

- (1) The pointers `left.tail` and `right.head` are local pointers. Reduce will set the next pointer in `left.tail` to point to `right.head`. This situation is illustrated in Figure 3(b).
- (2) The pointers `left.tail` and `right.head` are non-local pointers. These non-local pointers must match: `left.tail == right.head`, a condition that is guaranteed true in our system. This case is the inverse of *split* and results in the destruction of the non-local pointers (see Figure 3(a)). Note that non-local pointers appear in exactly two views during their existence as the only operations performed on views are *split*, *reduce* and *copy*.

The cases above express constraints on `left.tail` and `right.head`. The pointers `left.tail` and `right.head` may be local or non-local. E.g., if `left.head` equals q_{NL} , a non-local pointer distinct from p_{NL} , then the reduced view becomes $(q_{NL}, \text{right.tail})$, again a shared view. If `right.tail` is also a non-local pointer, say r_{NL} , then the result is the view (q_{NL}, r_{NL}) , again holding non-local pointers. Note that such a shared view is distinct from the empty view.

3.5. Queue Operations

The application can perform *push*, *pop* and *empty* operations on the reducing hyperqueue. Access is controlled through the view that the currently executing task holds

on the hyperqueue. We will describe in the next section how the views are created and destroyed. For now, assume that a task has a view to push only if it is allowed to push on a queue, and a view to pop only if it is allowed to pop.

The push and pop operations reflect common operation of a queue that is built as a linked list of segments. Two extensions are, however, required to maximize concurrency while ensuring determinism: turning local views into shared views; and tracking the existence of older producers.

The push operation retrieves the tail segment that it needs to operate on. If the buffer in the tail segment is full, then a new segment is appended to the list. This also updates the pointer in the view. Otherwise, the element is inserted at the tail end of the buffer in the segment. Note that the single-producer/single-consumer setup of the reducing hyperqueue implies that simple, lock-free algorithms can be used for the underlying queue.

```

1 void push( view & v, T value ) {
2     segment * seg = v.tail;
3     if( seg == null ) { // empty view
4         seg = new segment();
5         (view tmp, v) = split( (seg, seg) );
6         link_predecessor( tmp );
7     } else if( seg is full ) {
8         seg->next = new segment();
9         seg->producing = false;
10        v.tail = seg = seg->next;
11    }
12    seg->buffer[seg->tail_index] = value;
13    seg->tail_index = ( seg->tail_index + 1 ) mod seg->size;
14 }

```

It can happen that a task attempts to push a value on an empty view. This may happen, for instance, in Task 2 in Figure 2. Tasks 1 and 2 may produce values concurrently. In order to retain values in program order in the hyperqueue, a similar approach to Cilk++ reducers is followed: these tasks will initially push their values on distinct queue segments. Once the tasks have completed, their queue segments will be reduced (concatenated) and a correct, deterministic view on the queue will emerge. A critical issue is, however, that with Cilk++ reducers the data produced by Task 2 is visible only by itself until it completes. Until then, the consumer cannot access Task 2's data, which limits the degree of concurrency. To avoid this, we need to make Task 2's queue segment accessible by linking it up logically to Task 1.

In the scenario described, when Task 1 is created, it will receive a view on the tail end of the hyperqueue and Task 2 initially receives an empty view. On the first push operation, Task 2 will create a new segment and split it in a view to pop and a view to push. The view to push replaces its current view, while the view to pop is linked to the segments held in the predecessor task. At most one queue segment needs to be linked to the predecessor per executed task. The ensuing queue segments are also accessible as they are linked to the first segment. We will detail the `link_predecessor` function below.

The pop operation accesses the queue through a view to pop and retrieves the head segment from the view. It pops an element from the head index of the buffer in the segment.

```

1 T pop( view & v ) {
2     // Application must check emptiness of queue before
3     // calling pop(), guaranteeing segment is not empty
4     segment * seg = v.head;
5     T value = seg->buffer[seg->head_index];
6     seg->head_index = ( seg->head_index + 1 ) mod seg->size;

```

```

7   return value;
8 }

```

The pop operation is conditioned on the requirement to call the empty function before each call to pop. The empty method performs necessary actions such as destroying empty segments. At its heart, the empty method checks whether the head segment in the queue is empty or not. Such a check is sufficient for queues in general. However, this would be insufficient to maintain determinism in the hyperqueue. The goal of the hyperqueue is to answer every *pop* and *empty* call with the same return value as would be observed during a sequential, depth-first execution of the program. As such, the empty method may be called on a queue that is temporarily empty and one of two situations may occur:

- All tasks that appear earlier in the sequential execution order have completed. As such, no additional data can be produced that should be visible to the task executing *empty* and the method should return *true*.
- Some task with push access rights is still executing and may possibly produce additional data on the queue, and for this reason *empty* cannot yet determine whether it should return *true* or *false*.

These two cases are distinguished by adding a flag called *producing* to each queue segment. The flag is set to true upon creation of the segment and is set to false when a queue segment is linked in behind it. Moreover, the flag is modified in response to task creation and completion to reflect the possibility of additional data being produced on the segment. The empty method distinguishes the two cases above using the producing flag: if the flag is false, it returns true (queue is empty); otherwise, it needs to wait until data appears in the queue or the flag is set to false.

```

1 bool empty( view & v ) {
2     segment * seg = v.head;
3     // Spin until a task produces data in the segment 'seg' or
4     // indicates no data will arrive by clearing the producing flag
5     while( true ) {
6         if ( seg->head_index != seg->tail_index ) { // Segment not empty
7             return false;
8         }
9         if ( !seg->producing ) {
10            // No more data will be produced into 'seg'
11            if ( seg->next ) { // Segment is no longer needed, destroy it
12                segment * next = seg->next;
13                delete seg;
14                v.head = seg = next;
15            } else {
16                // Queue is empty
17                return true;
18            }
19        }
20    }
21 }

```

3.6. Runtime Actions

The runtime system maintains up to 4 views on the queue per strand. Every strand has the views *user* and *right*. Strands with push privileges also have the view *children* in order to collect queue segments created by the strand's children, while strands with pop privileges have the view *queue*. The top-level strand always has both push and pop privileges and thus maintains 4 views. The *user* view corresponds to the view to push; the *queue* view corresponds to the view to pop.

The *user*, *children* and *right* views operate similarly to Cilk++ reducers in terms of passing views to children, creating empty views and reducing views. The *queue* view provides access to the head of the queue is passed down to every task with pop privileges.

Spawn with push privileges. The *user* view is passed from the parent frame to the child frame. If *user.tail* points to a segment, then the producing flag in *user.tail* is set to true to indicate to *empty* calls that additional values may be produced. The parent's *user* view is cleared. This behavior is the common path when executing code sequentially.

Return from spawn with push privileges. Let us assume that a child frame *C* has finished execution, and that *C* was originally spawned by its parent frame *P*. The *right* view of *C* is reduced with its *user* view, linking it to the data produced by *C*'s right sibling $C.user \leftarrow \text{reduce}(C.user, C.right)$. If the tail of the updated *C.user* points to a segment, then that segment's producing flag is set to *false* when *C* has no right siblings, or when *C*'s right sibling has pop access rights. In the latter case, the producing flag must be turned off to ensure that a call to *empty* by the right sibling recognizes that the producer has completed.

If *C* has a left sibling *L*, then *C*'s values are reduced with *L*: $L.right \leftarrow \text{reduce}(L.right, C.user)$. If *C* has no left sibling, then it must be the oldest child of *P*. Thus, we perform: $P.children \leftarrow \text{reduce}(P.children, C.user)$.

Call and return from call with push privileges. For reasons of simplicity, we treat calls in the same way as spawns for the purpose of hyperqueues. We do not anticipate that call statements would be a common idiom on hyperqueues because calls forego concurrency with consumers.

Spawn with pop privileges. When a parent frame *P* spawns a child frame *C* with pop privileges, then *P*'s *user* view is moved to *C*'s *user* view ($C.user = P.user$; $P.user = \text{empty}$). The *user* view is passed to *C* to hide it from subsequent strands with push privileges. The *user* view will be reduced in correct program order when the current strand completes. Similarly, *P*'s *queue* view is passed over to *C*.

Return from spawn with pop privileges. When returning from a frame that was spawned with pop privileges, it is not necessarily the case that all elements have been consumed from the queue. The remaining elements must be passed on to the parent procedure. First, the same actions are taken as in the case of "return from spawn with push privileges". Next, the resulting *queue* view, which is a head-only view, is returned to the parent strand.

Sync. A frame *P* that executes a sync statement waits until all children have completed execution. As such, all spawned children have completed and they have reduced their local views with *P*'s *children* view. *P*'s *user* view is updated with the reduction of *P*'s *children* and *user* views.

Push on empty user view. A push operation that finds an empty user view executes the *link-predecessor* operation to accelerate the attachment of new queue segments to the segments that precede them in program order. This happens in response to parallel execution of tasks with push access mode. First a view (*seg*, *seg*) is created that contains the newly created segment '*seg*'. The method *split* splits this view in a head view and a tail view. The head view is then reduced with the right view in the immediate logically preceding strand. If the strand performing *link-predecessor* has a left sibling in the spawn tree, then the temporary view is reduced with the left sibling's *right* view:

```
1 left.right = reduce( left.right, tmp )
```

Sibling frames are recorded in a doubly-linked list to facilitate lookup of left and right sibling.

Fig. 4. Illustration of the concurrent execution of producing and consuming tasks and their effect on the construction and destruction of the hyperqueue. Legend: Q is queue view, C is children view, U is user view and R is right view. Arrows with a numeric ID represent non-local pointers, which occur in pairs.

If the creating strand does not have a left sibling, then the head pointer is reduced with the parent strand's *children* view. If this view is empty before the reduction, then the algorithm to share the queue head is executed recursively from the parent, until the top-level strand is encountered, where it is reduced with the *children* view.

Random Work Stealing. The design of Cilk++ reducers is tightly coupled with the operation of the Cilk++ scheduler, as explained in Section 3.1. Hyperqueues present a different use case. Hyperqueues are accessed within a context that describes the access privileges to the hyperqueue, which may be restricted to push or pop. This context is passed through specific argument types to a function through the **pushdep**, **popdep** and **pushpopdep** argument types. By consequence, hyperqueues are accessed as local variables. Moreover, hyperqueue views are small (two pointers per view), can be manipulated in an application-agnostic way (no indirect function calls to application code are required for memory allocation, initialization or reduction) and can be allocated on the call stack. As such, it is most efficient to manipulate views for hyperqueues at the time of spawn, call and return operations.

3.7. Example

Figure 4 presents an example of view creation and reduction. The top-level task (Task 0) spawns Task 1 with push privileges, followed by Task 4 with pop privileges, followed

by Task 6 with push privileges. Determinism requires that the effects of the tasks must be observed in this order. Task 1 in turn spawns Tasks 2 and 3 with push privileges. Task 2 pushes values 0–3 on the queue, while Task 3 pushes 4–7. Task 4 spawns Task 5 which pops values from the queue. Finally, Task 6 pushes the value 8 on the queue, which should not be observed by Tasks 4 and 5 in light of determinism.

Tasks 2 and 3 are spawned first and generate a partial list of values (Figure 4(a)). Task 2 inherits access to the initial queue segment through its *user* view and pushes values on that segment. Task 3 creates a new queue segment which it splits and then reduces the head with Task 2's *right* view. The split creates a new non-local pointer with unique ID (1). As discussed above, it is too early to link this segment to the segment operated on by Task 2 as Task 2 may perform more pushes and may require additional segments.

Tasks 4 and 5 are created concurrently while Tasks 1 and 2 execute (Figure 4(b)). Task 5 inherits the *queue* view through Task 4. Task 2 and Task 5 are now in a producer-consumer relation. They can make progress concurrently. In the mean time, Task 2 has pushed values 1, 2, 3 and Task 5 has popped value 0.

When Task 2 completes, its *user* and *right* views are reduced, together with Task 1's *children* view (Figure 4(c)). The *user* view is a tail-only view (due to the push), while the *right* view is a head-only view (due to the *split* and propagation of the head performed by Task 3's push). These views are reduced, leaving Task 1's *children* view with non-local pointers. This case shows the utility of splitting the view on new segments and reducing the head-only view ahead of the reduction of the tail. Even though Task 3 is still executing, the consumer is able to pop values produced by Task 3.

Finally, Task 6 is created and pushes values onto a new queue segment (Figure 4 (c)). Similar to Task 3, Task 6 shares the head of this queue segment with its left sibling (Task 4). By consequence, this segment is not linked with its predecessor and remains inaccessible to Tasks 4 and 5. This is, again, a requirement for deterministic execution. While this is going on, Task 5 pops the value 1.

3.8. Queue Segments

The queue segment at the head of the queue may also act as a queue in its own right. It is utilized as a circular buffer FIFO queue under those circumstances. This allows a concurrent producer and consumer to continuously reuse a queue segment, realizing a queue implementation with zero allocation cost in steady state. We use Lamport's lock-free algorithm for fixed-size array queues in this situation [Lamport 1983]. When the queue segment fills up, the producer will append a new segment to hold newly produced values. Note that only the queue segment at the head of the queue may be used in this mode because only one consumer can be executing at any time.

The hyperqueue is represented as a queue at two levels. At the inner level it is a fixed-size circular buffer based queue [Lamport 1983]. At the outer level, it is a lock-free linked-list based queue based on Michael and Scott's queue [Michael and Scott 1996]. We have however omitted the counters that protect against the ABA problem. These counters are unnecessary in case of a single-producer queue. The algorithm is further simplified by ensuring there is always at least one queue segment. Empty queues are problematic to ensure correct concurrent updates [Valois 1994; Fatourou and Kallimanis 2011]. Michael and Scott [1996] discuss this issue in detail. They avoid these issues by inserting a dummy node that does not hold elements. In our case, we guarantee that the queue always holds at least one queue segment, even if it is an empty segment, and we remove the dummy node.

3.9. Reducing Hyperqueue Invariants

Reducing hyperqueues respect the following invariants which we state without proof:

1. Every hyperqueue holds **at least one segment**. An initial segment is created when the hyperqueue is constructed. The last segment is not deleted when it is empty.
2. At any one time, for a given hyperqueue, there is **exactly one queue view with a local head pointer**. This view is accessible by the single task with **pop** privileges that is allowed to consume data.
3. The tail pointer in the *queue* view and the head pointer in the *user* view are **always** non-local unless if these views are empty. Space may be saved by not storing these pointers.
4. Every segment in a hyperqueue is pointed to by either **one next-segment** pointer, or by **one** view's *head* pointer.
5. Every segment stored in a hyperqueue is pointed to by **at most one** view's *tail* pointer. Every segment stored in a hyperqueue is pointed to by **exactly one** view's *tail* pointer if and only if the segment's *next-segment* pointer is null.
6. A consequence of invariants 4 and 5 is that **any segment may be shared by at most two tasks**, of which one is a consumer and one is a producer, as a consumer requires access through the head pointer and a producer requires access through the tail pointer.

Assume a total order $<$ of views that reflects the program order (following serial elision) in which the data stored in those views has been produced. We say that for views v_1 and v_2 , $v_1 < v_2$ when the following holds:

- (i) For a task T , $T.queue < T.children < T.user < T.right$. If a task does not have a particular view, the relation for that view is irrelevant.
- (ii) For sibling tasks T_1 and T_2 where T_2 is later in program order, all views of T_1 are ordered before T_2 's views.
- (iii) For tasks P and C where P is the parent of C , and for any view v of C , $P.children < C.v < P.user$.

7. If a linked list of segments is pointed to by the *head* pointer of view $T_1.v_1$ and by the *tail* pointer of view $T_2.v_2$, then $T_1.v_1 < T_2.v_2$ provided that v_1 is not a queue view. An interpretation of this invariant is that values are stored in an order that corresponds to program order.

8. For views $T_1.v_1$ and $T_2.v_2$ as in invariant 7, it holds that $T_2.v_2 < T_1.v_1$ provided that v_1 is a queue view and T_1 does not have both push and pop privileges. This invariant shows that a consumer task can only observe values that have been pushed by tasks preceding it in program order.

9. For views $T_1.v_1$ and $T_2.v_2$ as in invariant 7, if v_1 is not a queue view, then for any non-queue view v held by any task T , if $T_1.v_1 < T.v < T_2.v_2$, then v is a non-local view or ϵ .

The reducing hyperqueue achieves determinism by retaining all pushed values in program order, a functionality it inherits from Cilk++ reducers, and by ensuring all pop operations execute in program order. The latter is achieved by ensuring there is only one view in the program that allows popping elements from the queue and by passing this view to tasks in depth-first, left-child first order. Moreover, the implementation ensures that a value pushed by a strand can not be popped by any strand earlier in program order. This is achieved by ensuring that the segments on which the later strand can push are not reachable by following linked list edges starting from the view on the head of the queue. This is the case in the example for Task 6, which pushed a value 8 on the queue, but that value is not reachable from the head of the queue.

4. COUNTED HYPERQUEUES

The counted hyperqueue enables concurrent execution of consumers by tracking the sequence numbers of the elements stored in the queue. It assigns ranges of sequence numbers to producers and consumers and uses an efficient structure to match producers and consumers with overlapping ranges.

4.1. Queue Representation

The counted hyperqueue is constructed as a sequence of queue segments. Like the hyperqueue, every queue segment is a fixed-size array with metadata fields that track how many elements have been produced and how many have been consumed. Contrary to the hyperqueue, the counted hyperqueue does not build a linked list of queue segments. Instead, it stores queue segments in a lookup structure using the sequence number range of its elements as a key. Queue segments are stored in the index only when they have been fully constructed. After that moment, queue segments become read-only. The full life cycle of a queue segment consists of the following phases:

- (1) **Allocation:** Queue segments are allocated on demand whenever a producing task requires a queue segment.
- (2) **Write-only phase:** A producer pushes elements on a queue segment during a write-only phase. Only one producer has access to the queue segment at a time. Multiple producers can add elements sequentially when the queue segment is passed between tasks, e.g., during sequential execution. Consumers cannot access the queue segment during this time.
- (3) **Matching:** When producers are finished with a queue segment, the segment is matched against the index of queue segments and consumers. The index enables the transfer of data from producers to consumers.
- (4) **Read-only phase:** After matching, queue segments are used exclusively in read-only mode. A queue segment can be accessed by multiple consumers concurrently as each consumer knows exactly what values to use.
- (5) **Deallocation:** Queue segments are deallocated when all consumers have read their elements.

Besides the stored data, queue segments store the sequence number of their first element and a counter that tracks the number of elements yet to be consumed. The latter counter triggers deallocation of the segment when it drops to zero.

The counter hyperqueue derives its deterministic properties through assigning strictly ordered sequence numbers to all data pushed on the queue. It furthermore ensures that at each task scheduling point these sequence numbers are again assigned to tasks that may pop from the queue and it assigns those numbers in program order.

4.2. Queue Sequence Numbers

The programming model requires that tasks specify only the number of elements that will be pushed or popped. The runtime deduces the exact range of sequence numbers touched by a task by accumulating and propagating sequence numbers through the spawn tree. It associates following properties to every variable of type **counted_hyperqueue**, **suffixdep** or **prefixdep**:

- `start_{push,pop}`: the next sequence number accessed by the task
- `length`: the number of pushes or pops that must be made by the task, i.e., the length of the task's range
- `volume_{push,pop}`: the number of pushes or pops already performed, or promised to be performed by a child task

- `head`, `tail`: a pointer to the current queue segment at the head or tail, possibly a null pointer.
- `index`: a pointer to the counted hyperqueue's index structure where producers are matched up with consumers

Only the relevant properties are used, e.g., variables of type **prefixdep** do not require the variables relevant to push operations and vice versa.

These properties are updated by spawn statements, i.e., when tasks start. They do not need to be updated when a task finishes due to the predictive nature of the task argument annotations. Assume a task P spawns a task C with **suffixdep** annotation on a queue Q . Let `queue` be the variable accessible to P . This is either the Q variable itself, or a suffix taken from Q and passed to P as a **suffixdep** argument. Let `arg` be the **suffixdep** argument of C , which is derived from `queue`. Furthermore, let `arg.length` denote the length of the suffix `arg`. The properties of `arg` and `queue` are updated as follows:

```

1 arg.start_push = queue.start_push;   queue.start_push += arg.length;
2 arg.volume_push = 0;                 queue.volume_push += arg.length;
3 arg.tail = queue.tail;               queue.tail = null;
```

Similarly, for a **prefixdep** argument:

```

1 arg.start_pop = queue.start_pop;     queue.start_pop += arg.length;
2 arg.volume_pop = 0;                 queue.volume_pop += arg.length;
3 arg.head = queue.head;              queue.head = null;
4 signal error if queue.start_pop > queue.start_push;
```

Note that it is an error for the next sequence number to pop (`start_pop`) to progress beyond the next push index (`start_push`). Note that `queue.start_push` indicates the number of elements for which producer tasks have been spawned. This restriction is necessary to guarantee internal determinism. It is checked only when `queue` equals the main Q variable.

For simplicity, the parent task's tail and head pointers to its current segment are set to null. This is to simplify concurrency control on segments. If the parent task will perform further push or pop operations, it will lookup the relevant segment again using the updated sequence number.

The *push* and *pop* operations use the properties of the queue within the current task to perform the desired operations. They moreover check the correctness condition that not too many, and not too few, elements are pushed or popped by a task.

4.3. Producer-Consumer Synchronization

So far we have attributed a range of sequence numbers to queue segments when they are produced and to consuming tasks when they are spawned. The next step is to match queue segments to consuming tasks. This is a two-way synchronization process: either queue segments may be created prior to the consuming tasks, or the consuming tasks may be spawned prior to the required queue segments. To capture these two situations, we create an index consisting of two data structures: one to hold queue segments that have not yet been consumed, and one to hold consuming tasks that wait for the required queue segments. Each of these data structures is a skip list [Pugh 1990]. The skip lists are protected by a lock variable as matching either a queue segment or a consuming task requires access to both skip lists: a lookup in one skip list and a possible insertion in the other. These must be performed atomically.

The key functionality of the index is to determine whether consuming tasks are ready to execute. As such, we extend consuming tasks with a count of the number of elements that still need to be produced. The count is decremented as segments are added to the index. When the count is zero, the task is ready to execute. It is removed

from the index and pushed onto a ready list as in the case of dependency tracking on dataflow variables [Vandierendonck et al. 2013].

The outstanding element count is initialized to the length of the prefix of the spawned task. When a queue segment is matched against the index, it is first matched against the list of consuming tasks. If any tasks match, their outstanding element counters are decremented according to the overlap of sequence numbers between the queue segment and the tasks. When a task's counter drops to zero it is ready to execute and is removed from the index. Finally, the queue segment is added to the list of queue segments regardless of whether the dependent tasks are ready to execute. The queue segments are stored in the index until all of their elements have been consumed.

When a consuming task is spawned, it is also matched against the index. A similar process is followed whereby all queue segments with overlapping ranges result in a reduction of the outstanding element count. When the task's count drops to zero upon matching with completed queue segments, the task is executed immediately. Otherwise, it is added to the list of consuming tasks.

Queue segments are removed from the index only when all of their elements have been consumed.

4.4. Non-Blocking Prefix Access Mode

The **prefixdep** argument type comes in two versions: a blocking version and a non-blocking version. The blocking version is subject to resolution of data-flow dependencies through the queue. The non-blocking version is useful to pass a queue prefix unmodified through multiple levels of function calls.

The blocking version (**prefixdep<block>**) indicates that a task may start executing only when all of the queue elements it will pop have been produced. This guarantees that the task will not busy-wait for elements in the queue. If not all arguments are available, the task will be pending. It is handled similarly as tasks with data-flow dependencies are handled [Vandierendonck et al. 2011b; 2013]: the task is moved to a ready list once it becomes ready for execution. Once on the ready list, it will be picked up by one of the worker threads as part of the work stealing policy.

The non-blocking version (**prefixdep<noblock>**) is always allowed to start executing, regardless of whether the range of elements assigned to it are available or not. Such a task is however not allowed to pop elements from the queue itself. Instead it must spawn a blocking task to pop the elements.

The blocking nature of consumers in the counted hyperqueue implies that tasks never block during execution. This is in contrast to the reducing hyperqueue where the consumer can block when the queue is temporarily empty.

4.5. Counted Hyperqueue Example

Let us consider an example program to clarify the operation of the counted hyperqueue (Figure 5). The main task pipeline calls the tasks `parallel_producer` and `parallel_consumer` in parallel. Each of these tasks will produce or consume, respectively, 20 elements. To this end, the tasks each call two other tasks. Task 5 and Task 6 will each consume 10 elements, while Task 2 produces 5 elements and Task 3 produces 15 elements. By consequence, Task 5 will consume data produced by Task 2 and Task 3, while Task 6 only consumes data produced by Task 3.

Note that the `parallel_consumer` task (Task 4) is called with a non-blocking attributed on the queue. This creates more parallelism as it allows to execute Task 5 before all of the input data for Task 6 is available.

Figure 6 shows a possible parallel execution of the program. First Task 1 and 2 are created (Figure 6(a)). The sequence number ranges accessible to each task is updated as tasks are spawned and created. Spawning Task 1 with a suffix of length 20 advances

```

1 struct data {
2     ...
3 };
4 void
5 produce(suffixdep<data>queue, int n) {
6     /* Task 2, Task 3 */
7     for( int i=0; i < n; ++i ) {
8         data d = ...; // produce data
9         queue.push(d);
10    }
11 }
12 void
13 parallel_produce(suffixdep<data> queue,
14                 int n) {
15     /* Task 1 */
16     spawn produce(queue.suffix(5), 5);
17     spawn produce(queue.suffix(15), 15);
18     sync;
19 }
20 void consume(prefixdep<data,block> queue) {
21     /* Task 5, Task 6 */
22     while( !queue.empty() ) {
23         data d = queue.pop();
24         // ... operate on data
25     }
26 void parallel_consume(prefixdep<data> queue) {
27     /* Task 4 */
28     spawn consume(queue.prefix<block>(10));
29     spawn consume(queue.prefix<block>(10));
30     sync;
31 }
32 void pipeline() { /* Task 0 */
33     counted_hyperqueue<data> queue;
34     spawn parallel_produce(queue.suffix(20), 20);
35     spawn parallel_consume(
36         queue.prefix<noblock>(20));
37     sync;
38 }

```

Fig. 5. Example producer/consumer program using the counted hyperqueue.

Fig. 6. Illustration of the concurrent execution of producing and consuming tasks and their effect on the construction and destruction of queue segments and the index of the counted hyperqueue. Legend: Shaded tasks indicate tasks not ready to execute. These are not occupying a CPU core.

the start index for pushing in its parent Task 0 by 20. The volume is increased by 20 as Task 1 must push 20 elements. As such, there are 20 elements less that Task 0 may push from now on. Similarly, the range of Task 1 is updated when Task 2 is created.

Task 2 has produced the first element in the queue. As such, its start sequence number is 0 and volume is 1.

When Task 3 is created, the volume of Task 1 is increased to 20 (Figure 6(b)). This is equal to the length, so Task 1 is not allowed to push additional elements, or spawn tasks with non-zero suffix lengths. Task 4 has been created and can execute due to the non-blocking annotation. It spawns Task 5 with blocking mode. As the input data is not available in full yet, Task 5 cannot start execution. This is indicated by the shaded background. Task 5 is recorded in the index as waiting for the range of sequence numbers 0–10 and waiting for 10 elements to be produced.

When Task 2 completes (Figure 6(c)), it matches its queue segment with the index. It finds an overlap of 5 elements with the range of Task 5. It reduces the number of elements waited for by Task 5 accordingly. The queue segment is furthermore registered in the index such that Task 5 can acquire it when it executes. In the mean time Task 6 is spawned and registered on the index.

When Task 3 completes (Figure 6(d)), it matches its queue segment against the index in a similar manner. Now it finds overlapping ranges with both Task 5 and Task 6. The outstanding element count for both tasks now drops to 0, which makes both tasks runnable.

When Task 5 and Task 6 execute, they will acquire the segments from the index. The counters associated to each segment are decremented as each task consumes the elements from the segments. The segments are removed from the index and deallocated when the count drops to zero.

5. DISCUSSION

Peeking. Both versions of the hyperqueue allow peek operations, i.e., reading ahead values without removing them from the queue. Peeking is complicated in our design because of the way accesses to the queue are optimized. In short, *pop* and *peek* operations are implemented as a direct, unconditional access to the queue segment in order to minimize runtime overhead. We realize this as follows. When a hyperqueue is created, the programmer needs to declare the maximum distance by which *peek* may look-ahead past the head of the queue. Let us call this distance D_{peek} . The *empty* method blocks until the head of the queue has been produced, as well as the next D_{peek} elements. *Pop* and *peek* can now access the array in the head segment by adding an offset to the pointer to the head element.

An issue arises when the peeked values are straggling across the boundary of queue segments. In this case, we would need additional control logic to check whether peeked values are located in the head segment or in the next segment. This control would be required for every call to peek, while only a very small fraction of peeks straggle across the segment boundary. To avoid such performance overhead, we replicate the last D_{peek} elements in a segment at the head of the next segment. The elements are replicated when a new queue segment is pushed. Moreover, in the reducing hyperqueue we replicate D_{peek} elements when queue segments are linked together during reduction. In the counted hyperqueue, replication is performed when matching queue segments in the index.

Deadlock freedom. To demonstrate that both hyperqueues are free of deadlock we need to show that there cannot be dependence cycles between tasks [Coffman et al. 1971]. To be more precise, we will demonstrate that there cannot be dependence cycles between *strands*. The Cilk programming model defines dependences between strands such that the parallelism defined by *spawn* statements is exposed, and the serialization of *sync* statements is enforced. The dependences between strands are a partial ordering of strands that respects the total order of strands defined by sequential program order, i.e., the serial elision of the program.

On top of these dependences, we introduce additional producer-consumer dependences for the hyperqueues. These producer-consumer dependences also respect program order: only strands containing an *empty()* or *pop()* call depend on other strands and they can only depend on strands earlier in program order. As such, neither the Cilk-defined dependences nor the hyperqueue dependences introduce a dependence between strands that does not exist in the serial elision, which is a total order. It follows that the total set of dependences cannot contain cycles. As such, there always exists at least one strand that the scheduler can execute. This guarantees forward progress.

Alternative hyperqueue realizations. The two hyperqueue implementations differ by (i) how they deal with queue underflow (when required elements have not yet been pushed on the queue) and (ii) whether they support concurrent execution of consumers. These design choices have resulted in two very different implementations of the hyperqueue, one using a list-based structure and list reduction, the other using a centralized index structure to match producers and consumers.

We have identified three potential ways to deal with queue underflow: (i) the executing task and worker may block until the underflow is resolved, (ii) the executing task may be suspended and the worker may continue operating on a distinct task, or (iii) underflow is prevented through inter-task dataflow dependencies. The reducing hyperqueue uses the blocking strategy while the counted hyperqueue uses dataflow dependencies. It appeared that suspending tasks would have high overhead and would sit at odds with the *busy-leaves* property of Cilk [Blumofe and Leiserson 1994]. On the other hand, blocking causes the workers to wait and violates the time bound of work stealing.

The counted hyperqueue requires a different API than the reducing hyperqueue in order to support concurrent execution of consuming tasks. By consequence, these implementations cater to different types of programs depending on whether the number of elements pushed or popped by a task is known prior to its execution. We believe there is scope to define a hyperqueue that combines the APIs of the reducing and counted hyperqueues.

6. PROGRAMMING TECHNIQUES

Several programming techniques are helpful to optimize performance.

6.1. Queue Segment Length Tuning

The programmer often knows the best queue segment size for a program. E.g., a program performing producing or consuming data in parallel may generate the same number of values in each leaf task. It is beneficial to set the queue segment length equal to this number. Alternatively, the programmer may know that the total queue size is often around a particular size, or that the consumer and producer require a particular queue buffer length to remain in balanced execution without blocking. The queue segment length may be set at queue initialization time as a parameter to the constructor of the hyperqueue class.

6.2. Queue Slices

Queue slices are a performance optimization whereby a data structure is created (the slice) that provides fast-path implementations of the push, pop and empty operations. When creating a slice, the application must specify how many push or pop operations it intends to perform. The runtime system ensures that appropriate buffer space is available for push operations, and it records how much data is currently available in the queue for pop operations. Time-consuming operations relating to concurrency control and memory management are performed only when creating or destroying the slice.


```

1 bool producer( pushdep<int> queue, int block ) {
2   for( int i=0; i < block; ++i )
3     queue.push( ... );
4   return more_work_to_do() ? true : false;
5 }
6 void consumer( popdep<int> queue ) {
7   while( !queue.empty() )
8     ... = queue.pop();
9 }
10 void pipeline() {
11   hyperqueue<int> queue;
12   while( producer( (pushdep<int>)queue, 10 ) ) {
13     spawn consumer( (popdep<int>)queue );
14   }
15   sync;
16 }

```

Fig. 7. Taking the main queue iteration loop outside the tasks.

Read slices can be requested from tasks with pop privileges. The system returns the slice starting at the current head of the queue up to the requested length under the constraints that (i) the data must have been pushed and (ii) the slice must fit inside a single segment. If not, a shorter slice will be returned.

Write slices can be requested from tasks with push privileges. A new queue segment may be created to accommodate the requested slice length.

6.3. Queue Loop Split and Interchange

Another potential protection against unbounded queue growth is to split each stage's main loop over queue values and bring the outer loop outwards of the queue. This technique is illustrated in Figure 7. Instead of calling the *producer* function once, it is now called once for every 10 elements. The total degree of parallelism is equal to that of a solution with a single call to *producer* and *consumer*, except that memory usage is limited to grow by a factor 10 when the program is executed serially.

The same technique is useful in the case of counted hyperqueues. The technique can also help to limit the number of pending tasks (those not ready to execute) created by consumer tasks with the non-blocking attribute on the **prefixdep**.

7. EVALUATION

We evaluate the performance of pipeline parallel benchmarks implemented with POSIX threads, Intel's Threading Building Blocks and Swan, a task dataflow system [Vandierendonck et al. 2011b]. Moreover, the hyperqueues are also implemented in Swan in order to leverage the dataflow ordering functionality required to sequence tasks with pop privileges. Our implementation is published at <http://github.com/hvdieren/swan>.

The experimental system is a multi-core node with 2 AMD Opteron 6272 (Bulldozer) processors. On this processor, pairs of cores share a floating-point unit (FPU). The processors have 6144 KB L3 cache shared per 8 cores. Main memory is distributed over 4 NUMA nodes. The system runs the Ubuntu OS version 12.04.1 LTS and gcc version 4.6.3. We use Intel Threading Building Blocks (TBB) version 4.1 20130314oss.

We evaluate the hyperqueue on the ferret and dedup pipeline parallel benchmarks from the PARSEC suite [Bienia 2011]. The original hyperqueue publication contains observations also on the bzip2 application [Vandierendonck et al. 2013]. We evaluate the counted hyperqueue on the fm and lattice benchmarks from the StreamIt benchmark suite [Thies and Amarasinghe 2010]. Note that the API of the counted hyper-

Table II. Characterization of ferret's pipeline. The column "Parallel" indicates whether distinct iterations of a pipeline stage may execute in parallel or must execute serially.

Pipeline Stage	Parallel	Iterations	Time (s)	Time (%)
Input	serial	1	34.000	4.48
Segmentation	parallel	3500	26.800	3.57
Extraction	parallel	3500	2.773	0.35
Vectorizing	parallel	3500	133.939	16.20
Ranking	parallel	3500	603.286	75.30
Output	serial	3500	2.000	0.10

queue is not applicable to the PARSEC benchmarks because their tasks do not push or pop a fixed number of elements. The API of the reducing hyperqueue on the other hand can be applied to the StreamIt benchmarks but the loss of concurrency between multiple consumers would severely limit the scalability of the codes. The PARSEC codes are available from <http://github.com/hvdieren/parsec-swan>. The StreamIt benchmarks are contained in <http://github.com/hvdieren/swan>.

7.1. Ferret

Ferret performs content-based similarity search, determining for a set of images which images contain the same kind of object. The required computation is spread over a 6-stage pipeline consisting of, respectively, input (loading images from disk), segmentation, feature extraction, vectorizing, ranking and output. The first (input) and last (output) stages are serial stages, implying that these stages must operate on all images strictly in their original order. The stages in between have no permanent state. As such, multiple instances of these stages may be executing in parallel on distinct images.

We have measured the amount of time taken by each stage when executing the serial version of the benchmark on the PARSEC 'native' input (Table II). This table shows that the majority of execution time is taken by the *ranking* stage (75.3%), while the *vectorizing* stage also takes a sizable fraction of execution time (16.2%). The segmentation and extraction stages are less time consuming.

Serial stages can pose major limitations to scalability. The input stage takes about 4.5% of execution time. According to Amdahl's Law, scalability is limited to roughly 22 if we fail to overlap execution of the input stage with other work.

While the structure of the computation of ferret does not pose any problems toward parallelization (it is a common pipeline pattern), the code exposes a generic programmability issue. The input stage is a recursive directory traversal that collects image files in a directory tree. Written in Pthreads, files are pushed on a queue as they are discovered.

Turning ferret into a pipeline structure using programming models such as TBB or Swan is not impossible. However, it requires thoroughly restructuring the input stage in such a way that it can be called repeatedly to produce the next file [Reed et al. 2011]. To this end, its internal state must be made explicit (i.e., its current position in the traversal of the directory tree) and passed as an argument to the first stage. This is all but rocket science. But it is tedious and error-prone.

Hyperqueues avoid restructuring the program, thereby making it much easier to extract the latent parallelism in the program. With hyperqueues, the directory traversal pushes discovered image files on the queue, as in the pthreads version. These images can be concurrently consumed by the next pipeline stage. The appropriate variant is the reducing hyperqueue as it is unknown ahead of time how many files will be produced during directory traversal. It is not a major restriction that only one consumer can be active on the pipeline. We address by immediately spawning a task representing

```

1 void t_load( const char * dir, pushdep<load_data> queue ) {
2     // Recursively scan directory 'dir'. Pass queue to callees.
3     // Callees may push on queue
4     scan_dir( dir, queue );
5 }
6 void t_sxvr( load_data ld, pushdep<rank_data> outq ) {
7     seg_data seg;
8     rank_data rank;
9     extract_data x;
10    vec_query_data vq;
11    t_seg( load, &seg );
12    t_extract( &seg, &x );
13    t_vec( &x, &vq );
14    t_rank( &vq, &rank );
15    outq.push( rank );
16 }
17 void t_sxvr( popdep<load_data> inq, pushdep<rank_data> outq ) {
18     while( !inq.empty() ) {
19         load_data ld = load_queue.pop();
20         spawn t_sxvr( ld, outq );
21     }
22     sync;
23 }
24 void t_out( popdep<rank_data> queue ) {
25     while( !queue.empty() ) {
26         rank_data rank = queue.pop();
27         // append rank data to file ...
28     }
29 }
30 int main() {
31     hyperqueue<load_data> input_queue;
32     hyperqueue<rank_data> output_queue;
33     spawn t_load( dir, (pushdep<load_data>)input_queue );
34     spawn t_sxvr( (popdep<load_data>)input_queue, (pushdep<rank_data>)write_queue );
35     spawn t_out( (popdep<rank_data>)write_queue );
36     sync;
37 }

```

Fig. 8. Pseudo-code for ferret.

the rest of the pipeline after popping an element from the hyperqueue. A pseudo-code of the the pipeline structure is listed in Figure 8.

We measured the performance of ferret using Pthreads, TBB and Swan. We show the performance of two versions of the program using Swan. The “objects” version uses the baseline task dataflow model. In this case, we did not implement the code restructuring of the input stage as with the TBB code in order to demonstrate the importance of overlapping the execution of the input stage with the remainder of the pipeline. The “hyperqueue” version uses a hyperqueue to communicate data between the input stage and the segmentation stage, and also to communicate between ranking and output. The latter hyperqueue was inserted because of the fine granularity of the output stage. As such, we avoid spawning many small tasks. Instead a single large task is spawned for this stage which iterates over all elements in the queue.

Figure 9 shows the speedup of the pthreads, TBB, objects and hyperqueue implementations relative to the serial implementation. Performance of the objects version is clearly limited by not overlapping the input stage with the remainder of the pipeline. The remaining implementations show nearly the same performance.

Note a slight decrease of scalability when the number of cores exceeds 16. This is due to the sharing of FPUs between pairs of cores in the Bulldozer architecture.

Fig. 9. Ferret speedup by using various programming models.

Table III. Characterization of the dedup pipeline. The number of small chunks per large chunk varies between 473 and 65537 and is most often in the range 500-600.

Pipeline Stage	Parallel	Iterations	Time (s)	Time (%)
Fragment	serial	336	1.900	3.08
FragmentRefine	parallel	336	3.916	6.35
Deduplicate	mutual exclusion	369950	4.854	7.90
Compress	parallel	168364	45.881	74.48
Output	serial	369950	5.049	8.19

The pthreads version uses massive core oversubscription. It starts 28 threads for each of the parallel stages. Launching the same number of threads is clearly not justified by the breakdown in Table II. For best performance, the number of threads per stage needs to be tuned individually. The number 28 was experimentally determined and is likely a result of the maximum number of cores we used (32) and the fact that one stage dominates the execution time. As such, it is important to assign many threads to this stage. The hyperqueue implementation obtains the same performance as pthreads and does not require core-count dependent tuning.

Further insight in the efficiency of hyperqueues can be gleaned from comparison to Cilk-P [Lee et al. 2013], another pipeline-parallel system. We were not able to rebuild and deploy Cilk-P due to software version conflicts, however, the published performance analysis presents useful comparison points. Cilk-P achieves comparable performance to the P-threads and TBB implementations for ferret. Likewise, hyperqueues achieve comparable performance to these reference baselines, so we can infer that the hyperqueue performs comparable to Cilk-P.

7.2. Dedup

Dedup performs file compression through deduplication (eliminating duplicate data blocks) and compression. Dedup has a 5-stage pipeline that is tricky to implement efficiently using structured programming models such as TBB and Swan. The dedup pipeline stages consist of fragmentation (dividing the input file in large chunks), refining (splitting large chunks in small chunks), deduplication (finding equal chunks), compression of chunks and output. This pipeline poses implementation problems because of the variable number of input and output items in several stages. In particular, the fragment refining stage produces a variable number of small chunks per large chunk and the compression stage is skipped for duplicate chunks.

Table III shows the number of chunks processed and the time spent per pipeline stage. Execution time is biased towards *Compress*. Instances of this stage can execute

```

1 void Fragment( pushdep<chunk_t*>write_queue ) {
2   while( more coarse fragments ) {
3     chunk_t * chunk = ...;
4     { // Set up inner pipeline with local queue
5       hyperqueue<chunk_t*> * q = new hyperqueue<chunk_t*>;
6       spawn FragmentRefine( chunk, (pushdep<chunk_t*>)*q );
7       spawn DeduplicateAndCompress( (popdep<chunk_t*>)*q, (pushdep<chunk_t*>)write_queue );
8     }
9   }
10  sync;
11 }
12 int main() {
13   hyperqueue<chunk_t*> write_queue;
14   spawn Fragment( (pushdep<chunk_t*>)write_queue );
15   spawn Output( (popdep<chunk_t*>)write_queue );
16   sync;
17 }

```

(c) Hyperqueue implementation of dedup.

Fig. 10. Alternative implementation choices for dedup. The graphics (a) and (b) show dynamic instantiations of each pipeline stage, how they are grouped and where collections of data elements are used. Dashed lines indicate instances of the inner pipeline. (c) Sketch of hyperqueue code according to (b).

in parallel. The *Output* stage is the most limiting serial stage. Taking 8.2% of the execution time, it limits overall application speedup to 12.7.

Reed *et al* observed that dedup exhibits a nested pipeline [Reed et al. 2011]. The outer pipeline, handling large chunks, consists of three stages: *Fragment*, *Inner-Pipeline* and *Output*. The inner pipeline consists of *FragmentRefine*, *Deduplicate* and *Compress*. A new instance of the inner pipeline is created for every large chunk and produces a list of small chunks that makes up the corresponding large chunk.

Figure 10 (a) shows the dynamic instantiations of all pipeline stages. Two large chunks have been found, where the first is further split in three small chunks and the latter is split two-ways. This graphic demonstrates a shortcoming of the nested pipeline approach: all the small chunks for a large chunk must be completed and gathered on a list before the output stage can proceed. This puts an important limit to scalability, as the number of small chunks per inner pipeline is typically 500-600 and may run up to 65537, potentially resulting in long and skewed delays.

Hyperqueues allow consuming elements concurrently to pushes, removing the wait times of the output stage until large chunks have been fully processed as in the case of nested pipelines. Moreover, like Cilk++ list reducers, hyperqueues allow us to construct parts of the list concurrently and merge list segments as appropriate. This way, all nested pipelines can push elements on the same hyperqueue and the write actions

Fig. 11. Dedup speedup with various programming models.

become synchronized and ordered between invocations of the nested pipeline. Finally, hyperqueues can be used directly as a drop-in replacement for lists, as they support the required push and pop operations (Figure 10 (b)).

Our hyperqueue implementation inserts a local hyperqueue between the *Fragment-Refine* stage and the *Deduplication* stage. Also, all instances of the *Deduplication* and *Compress* stages that correspond to the same nested pipeline (large chunk) are merged into a single sequential task. This design was chosen to coarsen the tasks and reduce dynamic scheduling overhead (which is absent in the pthreads implementation). Ample parallelism remains in the program.

Our formulation of dedup follows the original sequential algorithm, which greatly affects programmer productivity. Figure 10 (c) shows a sketch, where the *main* procedure spawns two tasks *Fragment* and *Output*. *Fragment* calls all but the output stage in a recursive manner: whenever a large chunk is constructed, a nested pipeline is created using two tasks that communicate through a local hyperqueue. Completed small chunks are produced on the *write.queue*. In contrast, the TBB version of dedup requires significant restructuring of the code in order to match the structure imposed by TBB.

Note that the hyperqueue enforces dependencies across procedure boundaries. This is an effect that is hard to achieve in Swan, where dataflow dependencies can exist only within the scope of a procedure.

Figure 11 shows speedup for dedup in the pthreads, TBB and Swan programming models. Dedup is heavily memory bound as it searches for repeated patterns in large data volumes, e.g., disk images. This limits scalability. While Reed *et al* demonstrated improved performance of their TBB implementation relative to the pthreads implementation in PARSEC 2.1 [Reed et al. 2011], our evaluation using PARSEC 3.0 shows that the TBB implementation is slower than the pthreads implementation. The Swan implementation with hyperqueues outperforms the pthread version by at least 12% and up to 30% in the region of 6-8 threads. The hyperqueue implementation loses some of its advantage for 22 threads and higher due to task granularity and locality issues.

Cilk-P [Lee et al. 2013] provides further information to benchmark the performance of hyperqueues. The authors of Cilk-P modified the dedup program such that only one level of parallelism is used. This makes it hard to make a direct performance comparison. Given their change, their performance meets that of the TBB implementation, which is better than the P-threads implementation, and they achieve a speedup of 6.77 on 16 threads. Our implementation with hyperqueues exceeds the performance

Table IV. Characterization of the FMRadio pipeline when processing 10 million samples in the final stage. FMRadio uses a filter-bank of 10 filters and 64 taps per filter.

Pipeline Stage	Parallel	Output Elements	Input Peek	Time (s)	Time (%)
Signal generation/capture	serial	$5.0e7$	n/a	0.03	0.05
Low-pass filter	parallel	$1.0e7$	70	1.87	8.19
Demodulation	parallel	$1.0e7$	1	0.13	0.60
Equalizer	parallel	$1.0e7$	64	19.72	91.13
Aggregation/output	serial	n/a	0	0.02	0.05

Fig. 12. FMRadio speedup using dataflow objects and the counted hyperqueue.

of the P-threads and TBB implementations with multi-level parallelism. The speedup measured on our system is 6.3, however, we had issues replicating the reported performance with TBB.

7.3. FMRadio

The FMRadio benchmark models an FM signal demodulator with multi-band equalizer [Thies and Amarasinghe 2010]. The FMRadio pipeline consists of 5 stages (Table IV). The first (input) and final (output) stages execute serially while the inner pipeline stages can operate in parallel. The equalizer stage (4th) consumes most time: about 91.1% of the execution time. The low-pass filter stage (2nd) consumes about 8.2% of time. The remaining stages are not performance-critical.

We have implemented FMRadio using the counted hyperqueue, which allows high degrees of parallelism between pipeline stages and also within each of the inner stages. The main complexity with the FMRadio code is the use of varying peek distances, ranging from 0 to 70 (Table IV). Peeking is used in the code to implement sliding windows over samples, a typical construct in filters. The FMRadio code uses the “queue loop split and interchange” idiom (Section 6.3).

We also implemented a reference version using dataflow objects. We create two dataflow objects per queue and alternate use of the dataflow objects, i.e., while one iteration is producing data into one object, the prior iteration may be consuming the data from the other object. Moreover, peeked data is copied over from the other object in order to store all data in consecutive memory locations. Moreover, the runtime system dynamically renames objects in order to increase parallelism [Vandierendonck et al. 2011a], implying that more than two objects can exist per pipeline stage during execution. Speedup of the hand-optimized version saturates at 10x (Figure 12). This is due to a data dependency related to copying the peeked elements between successive objects in the same pipeline stage. This is a consequence of expressing dataflow

Fig. 13. Lattice speedup using hand-optimized code and the counted hyperqueue.

dependencies at the granularity of chunks of data. Ideally, we would express dependencies at the granularity of individual data items, which would be inefficient using dataflow. In contrast, the counted hyperqueue does track dependencies at the finest possible granularity and achieves nearly linear scalability with a speedup of 29.5 at 32 threads.

The counted hyperqueue again improves programmability. We find that implementation is fairly straightforward to write. The version using dataflow objects is more complex to write because (i) we need to explicitly manage the peeked data and copy it between objects (buffers), (ii) we need to make detailed calculations of how many elements are popped and peeked in order to make the code correct and (iii) we need to manage the start-up phase of the queue, where the number of elements popped and peeked differs from the steady-state. All of these issues disappear with hyperqueues.

7.4. Lattice

The Lattice benchmark implements a 10-stage lattice filter (Figure 13 (a)). Each delay pipeline stage ('dly' in the figure) pops one floating-point value and peeks one value from its input queue, and produces two new values (Figure 13 (b)). This grows the data volume and introduces imbalance between pipeline stages: The first two items generated by 'gen' are sufficient to generate two items in every stage of the pipeline. When 'gen' generates a third item, then the first delay stage generates two items from this and the previous item, the second delay stage generates 4 items, etc. In regime, each stage produces two values for every input value and the total data volume doubles in every stage. Each pipeline stage moreover performs minimal computation, resulting in a benchmark that heavily stresses the queue implementation. We implement this benchmark using the counted hyperqueue.

Figure 13 (c) shows the speedup of lattice when using the counted hyperqueue. The scalability of this code is relatively low. This is, however, a property of the code. Prior work has shown that Lattice is the least parallelizable of the StreamIt benchmarks. Unnikrishnan et al. [2009] have obtained on average no speedup on a 16-core soft core processor on FPGA. As such, a speedup of 3 is a good result.

We have identified that the poor performance of lattice results from bad locality. A straightforward (sequential) implementation first runs stage 0 to the end, followed by stage 1, etc. This results in very bad temporal locality as the queue content is fully instantiated before it is used once. For large data sets, the queue content may not fit in the CPU caches and incur increased latency.

A more cache-friendly execution order would send data through all pipeline stages before generating additional data in the 'gen' stage. The first two items generated by 'gen' are thus processed by all pipeline stages prior to generating a third item in 'gen'. For the third item in 'gen', the first delay stage generates two items. Of these, the first is processed through all stages prior to using the second one. This process is repeated in each pipeline stage. This limits the volume of data extent in the queues at any moment. We implemented this idea using dataflow dependencies and use 4K-element chunks of values to minimize runtime system overhead. This code performs much better than the hyperqueue implementation and achieves a speedup of 7 over serial execution. A similar optimization can be performed on the hyperqueue implementation. It would, however, result in a highly unnatural way of expressing a pipeline-parallel program.

7.5. Discussion

Cilk-P shows detailed results for three programs: dedup, ferret and x264. The authors of Cilk-P modified the dedup program, such that only the performance achieved by Cilk-P for ferret is comparable to the P-threads and TBB implementations. We have shown that the hyperqueue implementation also achieves performance that matches the P-threads and TBB implementations. We have not applied hyperqueues to x264.

8. RELATED WORK

We describe related work concerning the properties of the programming model and also the runtime scheduler.

8.1. Programming Model

The Threading Building Blocks (TBB) [Intel 2010] provide parallel skeletons that allow programmers to express parallel code structures in a generic way. TBB, however, does not define a serialization of the program and does not guarantee determinism, even in the case of specially crafted functionality [Katrano 2012]. TBB programs tend to be free of thread-count dependent parameters.

StreamIt [Thies et al. 2002] defines a language and compiler for streaming programs, which are closely related to pipelined programs. StreamIt programs are scale-free. However, the StreamIt compiler statically schedules the computations to cores, at which point this property is lost. StreamIt programs may be non-deterministic in which case there exists no unique serialization.

A fine-grain scheduler for GRAMPS graphics pipelines is described in [Sanchez et al. 2011]. The paper does not discuss aspects of determinism nor the existence of a serialization of GRAMPS programs. It does not provide examples to demonstrate that the system encourages scale-free programs.

Phasers are a multi-purpose synchronization construct applicable also to pipelines [Shirako et al. 2008]. Programs constructed with phasers are not serializable and are not scale-free, although they are deterministic [Cavé et al. 2011].

OpenSTREAM is a system for stream- and task-based programming [Pop and Cohen 2013]. Similarly to the counted hyperqueue, it uses a matching mechanism to line up producers and consumers. It lines up the producers and consumers in one queue each and matches the head of the producer queue with the head of the consumer queue. OpenSTREAM programs are deterministic provided that consuming tasks are created before producing tasks. Removing parallel constructs from OpenSTREAM programs does not deliver a workable serialization. OpenSTREAM does, however, provide compiler support to optimize the execution of stream-based programs.

Cilk-P, also known as Piper, is another pipeline extension to Cilk [Lee et al. 2013; ?]. The basic pipeline pattern supported by Cilk-P is a grid-like pattern where every stage of the pipeline is serialized with other instances of the same stage. Cilk-P also

supports relaxations where a subset of the cross-iteration pipeline stage dependencies are removed. The main restriction of Cilk-P is that producers and consumers need to synchronize at the level of the pipeline-parallel loop. This makes it cumbersome to program producers or consumers that consist of recursive functions or otherwise deeply nested function calls. As such, cases like *ferret* and *dedup* require the same extensive code transformations for Cilk-P as for Intel TBB. Also, extensive function inlining needs to be performed in order to move the deeply nested producer-consumer dependency edge to an inter-stage dependency edge in the pipeline-parallel loop.

Concurrent data structures [An et al. 2003; Lea 2013] can be used in conjunction with thread-oriented parallel programming abstractions such as POSIX threads and Java threads. Concurrent data structures allow multiple threads to access the data structure concurrently with a guarantee that each thread's effects occur in some perceived order, as in the case of the linearizability condition [Herlihy and Wing 1990]. Concurrent data structures are not deterministic (in the sense used in this paper) and they do not provide a serialization of the program.

A FlowPool [Prokopec et al. 2013] is a programming abstraction that provides determinism guarantees for concurrent data structures. A FlowPool is a data structure that contains an unordered collection of data items. It allows concurrent, side effect-free operations on the collection that calculate reduction operations, called aggregates in FlowPool terminology. Determinism in FlowPool implies that every execution either results in the same computed value, or in a failure. There is no guarantee that failures always occur in the same location. As FlowPools store elements without order, they implement reductions that are associative and commutative. By consequence, it does not provide determinism for reductions that compute lists of elements, or queues, as these are non-commutative.

8.2. Scheduling

It has been shown that pipeline parallelism is best scheduled dynamically in order to cope with imbalanced pipeline stages [Navarro et al. 2009]. The baseline Swan runtime system performs such dynamic load balancing very effectively, also for pipeline parallel programs [Vandierendonck et al. 2011b].

Pipeline stages may be seen as transformations on work items [Macdonald et al. 2004]. Threads pick work items from queues holding work items from various stages in the pipeline. Threads advance the work items to the next stage and return them to the queues until processing is completed. This model is scalable as more threads are easily added to execute the pipeline. It also closely corresponds to the way the baseline Swan system executes pipelines, except that Swan retains program order and gives preferences to complete older work items before generating new ones.

DoPE [Raman et al. 2011] adapts the degree of parallelism in statically scheduled programs by switching dynamically between static schedules. DoPE introduces some opportunity to change the scale, but switching between versions is costly as it requires to drain the pipeline.

Others have devised specific strategies to identify performance limiting stages [Suleman et al. 2010]. Additional threads are assigned to the limiting stages and taken away from the others. Swan achieves this effect automatically, without analyzing per-thread performance.

GRAMPS [Sanchez et al. 2011] implements techniques to limit memory footprint during execution, to optimize the usage of intermediate buffer space and to recycle thread state for serial pipeline stages. Overall, the Swan scheduler executes a comparable schedule, but its genericity foregoes optimization specific to pipeline parallelism.

Each iteration of a pipeline-parallel loop in Cilk-P is converted by the compiler to a serial task progressing through the stages of the pipeline [Lee et al. 2013]. Tasks are al-

lowed to progress to the next stage only if prior iterations have reached the corresponding stage. The Cilk-P scheduler implements two optimizations to checking progress across stages. Lazy enabling checks completion of a previous stage rather than notifying future stages that they may continue. This minimizes redundant checks. Dependency folding is an optimization that builds on the observation that the prior task can only make progress over its stages. Thus, the progress of the prior task can be cached and progress checks can be omitted until the same stage is reached.

9. CONCLUSION

Determinism and scale-free parallelism are key characteristics of ubiquitous parallel programming models that improve programmer productivity and code quality. This paper presents hyperqueues, a programming abstraction of queues that allows to specify deterministic and scale-free programs with pipeline parallelism.

We explain the semantics and implementation of two types of hyperqueues in the context of a task dataflow programming language and runtime system. Exposing different APIs, these hyperqueues tailor the implementation to program properties exposed in the API. Both hyperqueues give the programmer the perception of a single-producer single-consumer queue. They, however, allow concurrent execution of producers while retaining this property. The reducing hyperqueue furthermore uses active polling in the consumer in case the queue runs empty. The counted hyperqueue supports concurrent execution of consumers and avoids polling by deferring consumers until their data is available. Program properties determine which variation of the hyperqueue is applicable.

Application to several irregular pipeline parallel programs shows that the hyperqueues achieve the same performance for ferret, 30% better performance for dedup and 2.9x better performance for FMRadio on a 32-core shared memory machine compared to other parallel models for expressing pipeline parallelism.

We have identified scope for alternative definitions of hyperqueues, making different trade-offs in the programming interface and implementation.

REFERENCES

- P. An, A. Julia, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. 2003. STAPL: an adaptive, generic parallel C++ library. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing (LCPC'01)*. Springer-Verlag, Berlin, Heidelberg, 193–208. <http://dl.acm.org/citation.cfm?id=1769331.1769344>
- M. Bauer, S. Treichler, E. Slaughter, and A. Aitken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. <http://dl.acm.org/citation.cfm?id=2388996.2389086>
- C. Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.
- G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 181–192. DOI: <http://dx.doi.org/10.1145/2145816.2145840>
- R. D. Blumofe and C. E. Leiserson. 1994. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 356–368. DOI: <http://dx.doi.org/10.1109/SFCS.1994.365680>
- R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*. ACM, New York, NY, USA, 535–548. DOI: <http://dx.doi.org/10.1145/1926385.1926447>
- V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. 2011. Habanero-Java: The New Adventures of Old X11. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, New York, NY, USA, 51–61. DOI: <http://dx.doi.org/10.1145/2093157.2093165>

- E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (June 1971), 67–78. DOI: <http://dx.doi.org/10.1145/356586.356588>
- P. Fatourou and N. D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (SPAA '11)*. ACM, New York, NY, USA, 325–334. DOI: <http://dx.doi.org/10.1145/1989493.1989549>
- M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 79–90. DOI: <http://dx.doi.org/10.1145/1583991.1584017>
- M. P. Herlihy and J. M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. DOI: <http://dx.doi.org/10.1145/78969.78972>
- Intel 2010. *Intel Threading Building Blocks*. Intel. Document Number 319872-006US.
- J. C. Jenista, Y. h. Eom, and B. C. Demsky. 2011. OoOJava: software out-of-order execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11)*. ACM, New York, NY, USA, 57–68. DOI: <http://dx.doi.org/10.1145/1941553.1941563>
- A. Katranov. 2012. Deterministic Reduction: a new Community Preview Feature in Intel® Threading Building Blocks. <http://software.intel.com/en-us/blogs/2012/05/11/deterministic-reduction-a-new-community-preview-feature-in-intel-threading-building-blocks>. (Nov. 2012). Retrieved April. 24th, 2013.
- L. Lamport. 1983. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.* 5, 2 (April 1983), 190–222. DOI: <http://dx.doi.org/10.1145/69624.357207>
- D. Lea. 2013. Concurrency JSR-166 Interest Site. <http://gee.cs.oswego.edu/dl/concurrency-interest/>. (2013). Last retrieved: July 2013.
- I-T. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. 2013. On-the-fly Pipeline Parallelism. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. ACM, New York, NY, USA, 140–151. DOI: <http://dx.doi.org/10.1145/2486159.2486174>
- S. Macdonald, D. Szafron, and J. Schaeffer. 2004. Rethinking the pipeline as object-oriented states with transformations. In *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*. IEEE Computer Society, Washington, DC, USA, 12–21. DOI: <http://dx.doi.org/10.1109/HIPS.2004.1299186>
- M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96)*. ACM, New York, NY, USA, 267–275. DOI: <http://dx.doi.org/10.1145/248052.248106>
- A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. 2009. Analytical Modeling of Pipeline Parallelism. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*. ACM, New York, NY, USA, 281–290. DOI: <http://dx.doi.org/10.1109/PACT.2009.28>
- OpenMP 2013. OpenMP Application Programming Interface, version 4.0. <http://www.openmp.org/>. (July 2013).
- Hannes Payer, Harald Roeck, Christoph M. Kirsch, and Ana Sokolova. 2011. Scalability Versus Semantics of Concurrent FIFO Queues. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '11)*. ACM, New York, NY, USA, 331–332. DOI: <http://dx.doi.org/10.1145/1993806.1993869>
- A. Pop and A. Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages. DOI: <http://dx.doi.org/10.1145/2400682.2400712>
- P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. 2011. A programming model for deterministic task parallelism. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '11)*. ACM, New York, NY, USA, 7–12. DOI: <http://dx.doi.org/10.1145/1988915.1988918>
- Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2013. *Flow-Pools: A Lock-Free Deterministic Concurrent Dataflow Abstraction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 158–173. DOI: http://dx.doi.org/10.1007/978-3-642-37658-0_11
- W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. DOI: <http://dx.doi.org/10.1145/78973.78977>
- A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. 2011. Parallelism orchestration using DoPE: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. ACM, New York, NY, USA, 26–37. DOI: <http://dx.doi.org/10.1145/1993498.1993502>
- E. C. Reed, N. Chen, and R. E. Johnson. 2011. Expressing Pipeline Parallelism Using TBB Constructs: A Case Study on What Works and What Doesn'T. In *Proceedings of the Compilation of the Co-located Work-*

- shops on DSM'11, TMC'11, AGERE! 2011, AOOPEs'11, NEAT'11, & VMIL'11 (SPLASH '11 Workshops)*. ACM, New York, NY, USA, 133–138. DOI: <http://dx.doi.org/10.1145/2095050.2095074>
- D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, Washington, DC, USA, 22–32. DOI: <http://dx.doi.org/10.1109/PACT.2011.9>
- J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. 2008. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing (ICS '08)*. ACM, New York, NY, USA, 277–288. DOI: <http://dx.doi.org/10.1145/1375527.1375568>
- M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. 2010. Feedback-directed pipeline parallelism. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*. ACM, New York, NY, USA, 147–156. DOI: <http://dx.doi.org/10.1145/1854273.1854296>
- W. Thies and S. Amarasinghe. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*. ACM, New York, NY, USA, 365–376. DOI: <http://dx.doi.org/10.1145/1854273.1854319>
- W. Thies, M. Karczmarek, and S. P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, 179–196. <http://dl.acm.org/citation.cfm?id=647478.727935>
- D. Unnikrishnan, J. Zhao, and R. Tessier. 2009. Application Specific Customization and Scalability of Soft Multiprocessors. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*. 123–130. DOI: <http://dx.doi.org/10.1109/FCCM.2009.41>
- J.D. Valois. 1994. Implementing Lock-Free Queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*. IEEE Computer Society, Las Vegas, 64–69.
- H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos. 2013. Deterministic Scale-free Pipeline Parallelism with Hyperqueues. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 32, 12 pages. DOI: <http://dx.doi.org/10.1145/2503210.2503233>
- H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. 2011a. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*. USENIX Association, Berkeley, CA, USA, 6.
- H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. 2011b. A Unified Scheduler for Recursive and Task Dataflow Parallelism. In *PACT '11: Proceedings of the 20th international conference on Parallel architectures and compilation techniques*. IEEE Computer Society, Washington, DC, USA, 1–11.
- H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. 2013. Analysis of Dependence Tracking Algorithms for Task Dataflow Execution. *ACM Trans. Archit. Code Optim.* 10, 4, Article 61 (Dec. 2013), 24 pages. DOI: <http://dx.doi.org/10.1145/2555289.2555316>

Received June 2016; revised July 2019; accepted July 2019