View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

Dynamic Expressivity with Static Optimization for Streaming Languages

Robert Soulé
Cornell University
soule@cs.cornell.edu

Michael I. Gordon, Saman Amarasinghe
Massachusetts Institute of Technology
{mgordon,saman}@mit.edu

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Martin Hirzel
IBM Research
hirzel@us.ibm.com

ABSTRACT

Developers increasingly use streaming languages to write applications that process large volumes of data with high throughput. Unfortunately, when picking which streaming language to use, they face a difficult choice. On the one hand, dynamically scheduled languages allow developers to write a wider range of applications, but cannot take advantage of many crucial optimizations. On the other hand, statically scheduled languages are extremely performant, but have difficulty expressing many important streaming applications.

This paper presents the design of a hybrid scheduler for stream processing languages. The compiler partitions the streaming application into coarse-grained subgraphs separated by dynamic rate boundaries. It then applies static optimizations to those subgraphs. We have implemented this scheduler as an extension to the StreamIt compiler. To evaluate its performance, we compare it to three scheduling techniques used by dynamic systems (OS thread, demand, and no-op) on a combination of micro-benchmarks and real-world inspired synthetic benchmarks. Our scheduler not only allows the previously static version of StreamIt to run dynamic rate applications, but it outperforms the three dynamic alternatives. This demonstrates that our scheduler strikes the right balance between expressivity and performance for stream processing languages.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization*

Keywords

Stream Processing; StreamIt

1. INTRODUCTION

The greater availability of data from audio/video streams, sensors, and financial exchanges has led to an increased demand for applications that process large volumes of data with high throughput. More and more, developers are using *stream processing* languages to write these programs. Indeed, streaming applications have become ubiquitous in government, finance, and entertainment.

A streaming application is, in essence, a data-flow graph of streams and operators. A *stream* is an infinite sequence of data items, and an *operator* transforms the data. The *data transfer rate* of an operator is the number of data items that it consumes and produces each time it fires. In *statically scheduled* stream processing languages, every operator must have a fixed data transfer rate at compile time. In contrast, *dynamically scheduled languages* place no restriction on the data transfer rate, which is determined at runtime.

Without the restriction of a fixed data transfer rate, dynamic streaming systems, such as STREAM [2], Aurora [1], and SEDA [17], can be used to write a broader range of applications. Unfortunately, many optimizations cannot be applied dynamically without incurring large runtime costs [11]. As a result, while dynamic languages are more expressive, they are fundamentally less performant. Using fixed data transfer rates, compilers for static languages such as Lime [3], StreamIt [15], Esterel [4], and Brook [5] can create a fully static schedule for the streaming application that minimizes data copies, memory allocations, and scheduling overhead. They can take advantage of data locality to reduce communication costs between operators [7], and they can automatically replicate operators to process data in parallel with minimal synchronization [6]. This paper addresses the problem of how to balance the tradeoffs between *expressivity* and *performance* with a hybrid approach.

In an ideal world, all applications could be expressed statically, and thus benefit from static optimization. In the real world, that is not the case, as there are many important applications that need dynamism. We identify four major classes of such applications:

- *Compression/Decompression.* MPEG, JPEG, H264, gzip, and similar programs have data-dependent transfer rates.
- *Event monitoring.* Applications for automated financial trading, surveillance, and anomaly detection for natural disasters critically rely on the ability to filter (e.g. drop data based on a predicate) and aggregate (e.g. average over time-based or attribute-delta based window).
- *Networking.* Software routers and network monitors such as Snort [13] require data-dependent routing.
- *Parsing/Extraction.* Examples include tokenization (e.g. input string, output words), twitter analysis (e.g. input

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.
Copyright 2013 ACM 978-1-4503-1758-0/13/06 ...$15.00.

tweet, output hashtags), and regular expression pattern matching (e.g. input string, output all matches).

While these applications fundamentally need dynamism, only a few streams in the data-flow graph are fully dynamic. For instance, an MPEG decoder uses dynamism to route i-frames and p-frames along different paths, but the operators on those paths that process the frames all have static rates. Financial computations require identifying events in data-dependent windows, but static rate operators process those events. A network monitor recognizes network protocols dynamically, but then identifies security violations by applying a static rate pattern matcher.

Based on this observation, we developed our hybrid scheduling scheme. The compiler partitions the streaming application into coarse-grained subgraphs separated by dynamic rate boundaries. It then applies static optimizations to those subgraphs, which reduce the communication overhead, exploit automatic parallelization, and apply inter-operator improvements such as scalarization and cache optimization.

We have implemented this hybrid scheduling scheme for the StreamIt language. To evaluate its performance, we compared it to three scheduling techniques used by dynamic systems (OS thread, demand, and no-op) on a combination of micro-benchmarks and real-world inspired synthetic benchmarks. In all three cases, our hybrid scheduler outperformed the alternative, demonstrating up to 10x, 1.2x, and 5.1x speedups, respectively. In summary, this paper makes the following contributions:

- An exploration of the tradeoffs between static and dynamic scheduling.
- The design of a hybrid static-dynamic scheduler for streaming languages that balances expressivity and performance.
- An implementation of our hybrid scheduler for the StreamIt language that outperforms three fully dynamic schedulers.

Overall, our approach yields significant speedup over fully dynamic scheduling, while allowing stream developers to write a larger set of applications than with only static scheduling.

2. STREAMIT BACKGROUND

Before presenting the design of our hybrid scheduler, we briefly describe the StreamIt language. The left-hand side of Figure 1 shows a snippet of StreamIt code used for video processing. The right-hand side shows a graphical representation of the same code, which we will use throughout this paper. The program receives a video stream as input, and decodes it by applying a sequence of operations: Huffman decoding, inverse quantization, and an inverse discrete cosine transformation.

The central abstraction provided by StreamIt is an operator (called a *filter* in the StreamIt literature). Programmers can combine operators into fixed topologies using three composite operators: *pipeline*, *split-join* and *feedback-loop*. Composite operators can be nested in other composites. The example code shows four operators composed in a pipeline. The operators are `VideoInput`, `Huffman`, `IQuant`, and `IDCT`.

Each operator has a *work* function that processes streaming data. To simplify the code presentation, we have elided the bodies of the work functions. When writing a work function, a programmer must specify the *pop* and *push* rates for that function. The pop rate declares how many data items from the input stream are consumed each time an operator

```
1  float->float pipeline Decoder {
2      add float->float filter VideoInput() {
3          work pop 1 push 1 {
4             float input, result;
5             input = pop();
6            /* details elided */
7             push(result);
8          }
9      }
10     add float->float filter Huffman () {
11         work pop * push 1
12         { /* details elided */ }
13     }
14     add float->float filter IQuant () {
15         work pop 64 push 64
16         { /* details elided */ }
17     }
18     add float->float filter IDCT () {
19         work pop 8 push 8
20         { /* details elided */; }
21     }
22 }
```

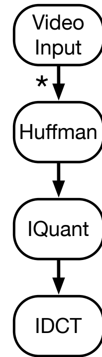


Figure 1: StreamIt code for video processing.

executes. The push rate declares how many data items are produced.

When all pop and push rates are known at compilation time, a StreamIt program can be statically scheduled. In the example, all of the operators except `Huffman` have static pop and push rates. If a pop or push rate can only be determined at run time, then the character `*` is used instead of a number literal to indicate that a rate is dynamic. An `*` appears on both line 11 of the `Huffman` operator, and in the graphical representation.

To access the data items that are popped and pushed, StreamIt provides built-in *pop()* and *push()* functions, such as appear in lines 5 and 7. These functions implicitly read from an input stream or write to an output stream.

There are several execution paths for the StreamIt compiler, which use different scheduling strategies. The next section reviews these different strategies.

3. RELATED WORK

Table 1 presents an overview of various approaches for scheduling stream processing languages, which we contrast with our hybrid scheduling technique. Our scheduler appears in the last row, shaded in grey.

The simplest approach is **sequential scheduling**. All operators are placed into a single thread, with no support for parallel execution. The StreamIt Library [15] uses this approach, and implements dynamism by having downstream operators directly call upstream operators when they need more data.

In **OS thread scheduling**, each operator is placed in its own thread, and the scheduling is left to the underlying operating system. This approach is used by some database implementations, and is similar to the approach used by the SEDA [17] framework for providing event-driven Internet services. To improve performance, SEDA increases the number of times each operator on the thread executes. This optimization, called *batching* [9], increases the throughput of the application at the expense of latency. This form of dynamic scheduling is easy to use, since all scheduling is left to the operating system. However, without application knowledge, the operating system cannot schedule the threads in

Scheduling Scheme	Approach	Benefits and Drawbacks
Sequential	Operators are placed in a single thread and execute sequentially.	No parallelism, but low latency.
OS Thread	Each operator gets its own thread. The operating system handles the scheduling.	Easy to implement. Suffers from lock contention, cache misses, and frequent thread switching.
Demand	Fused operators are scheduled to run when data is available.	Uses fusion to reduce the number of threads and batching to improve throughput. It is not spatially-aware, does not optimize across operators, and has no data parallelization.
No-op	Implements dynamism by varying the size of the data. Always sends a data item, but the data item can be a nonce.	Does not implement data-parallelization. Increased costs associated with sending no-op data values.
Hardware Pipelining	Stream graph is partitioned into contiguous, load-balanced regions, and each region is assigned to a different core.	Low latency, but load-balancing is very difficult, leading to low utilizations.
Static Data-Parallelism	Data-parallelism applied to coarse-grained stateless operators. Double buffering alleviates stateful operator bottlenecks.	No dynamic applications. Increases throughput at the expense of latency.
Hybrid Static/Dynamic	Partition into coarse-grained components with dynamic boundaries. Apply static optimizations to the components.	Allows for dynamic data transfer rates, is spatially-aware, implements fusion, batching, cross-operator, and data-parallel optimizations.

Table 1: Overview of scheduling approaches. Our scheduler appears in the last row, shaded in grey.

an optimal order, so there are frequent cache misses, unnecessary thread switches, and increased lock contention.

In **demand scheduling**, the scheduler determines which operators are eligible to execute by monitoring the size of their input queues. When an operator is scheduled, it is assigned to a thread from a thread pool. One example of a system using this technique is Aurora [1]. Aurora does not map threads and operators to cores with consideration to their data requirements, i.e., it is not spatially aware. However, it does provide two optimizations that improve on basic demand scheduling. First, like SEDA, it implements batching. Second, it implements a form of operator *fusion* [9], by placing multiple operators on the same thread to execute. Fusion reduces communication overhead and the frequency of thread switching. Like Aurora, our hybrid scheduler implements both the fusion and batching optimizations. Unlike Aurora, our scheduler data-parallelizes operators. With *data-parallelization* replicas of the same operator on different cores process different portions of the data concurrently. Additionally, our scheduler can optimize across fused operators, such as by performing scalarization to further reduce inter-operator communication costs.

One common approach that static languages use to implement dynamic scheduling is **no-op scheduling**. With this approach, special messages are reserved to indicate that an operator should perform a *no-operation*. An operator always produces a fixed number of outputs, but some of those outputs are not used for computation. CQL [2] implements a variation of this approach. In CQL, each operator always produces a bag (i.e. a set with duplicates) of tuples. The size of the bag, however, can vary. Therefore, an operator can send an empty bag to indicate that no computation should be performed by downstream operators. As a result, it suffers from increased costs associated with sending no-op values. In contrast to our scheduler, CQL does not data-parallelize operators.

With **hardware pipelining**, neighboring operators are fused until there are fewer or equal operators as cores. Each fused operator is then assigned to a single core for the life of the program. This allows upstream and downstream operators to execute in parallel. The StreamIt infrastructure includes a compilation path that mainly exploits this approach [7] for several different target platforms, including clusters of workstations [14], the MIT Raw microprocessor [16], and Tilera's line of microprocessors [18]. The hardware pipelining path supports dynamic rates, but it cannot fuse operators if they have dynamic data transfer rates. The challenge for hardware pipelining is to ensure that each fused set of operators performs approximately the same amount of work, so that the application is properly load balanced. For real-world applications, this is difficult [6]. Dynamic data transfer rates make the problem even harder, because there is no way to statically estimate how much work an operator with a dynamic data transfer rate will perform. Consequently, hardware pipelining was largely abandoned as a compilation strategy by StreamIt, except when targeting FPGAs.

With **static data-parallelism**, the compiler tries to aggressively fuse all operators, and then data-parallelize the fused operators so that they occupy all cores. One complication for this strategy is that operators with stateful computations cannot be parallelized, and therefore introduce bottlenecks. There are compilation paths of the StreamIt compiler [6] that target commodity SMP multicores and Tilera multicore processors using the static data-parallelism approach. The StreamIt compiler offsets the effects of stateful operator bottlenecks by introducing *double buffering* between operators. With double buffering, non-parallelized operators can execute concurrently, because the buffer that a producer writes to is different from the buffer from which the consumer reads.

Our **hybrid scheduler** extends the static data-parallelism path of the StreamIt compiler. The static data-parallelism strategy is scalable and performant across varying multicore architectures (both shared memory and distributed memory) for real world static streaming applications [6], but

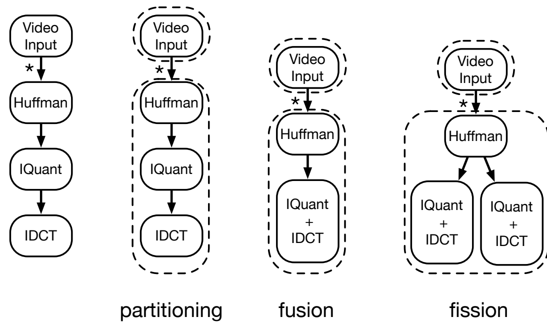


Figure 2: The data-flow graph is first partitioned into static subgraphs. Solid edges are streams from producers to consumers. The asterisk indicates a dynamic communication channel, and the dashed line indicates the static subgraphs. Each static subgraph is optimized by fusing and then parallelizing.

does not include the expressiveness of dynamic data transfer rates. In this work we achieve scalable parallelism with minimal communication for a wider set of streaming applications. Our strategy partitions the application into static subgraphs separated by dynamic rates, and applies the static data-parallelism optimizations to the subgraphs.

4. COMPILER TECHNIQUES

In practice, many streaming applications contain only a small number of operators with dynamic data transfer rates, while the rest of the application is static. This observation motivates our design. The high-level intuition is that the compiler can partition the operators into subgraphs separated by dynamic rate boundaries. It can then treat each subgraph as if it were a separate static application, using a *static data-parallelism* scheduler. In other words, within a subgraph, operators communicate through static buffers, and the compiler can statically optimize each subgraph independently of the rest of the application.

The compiler for our hybrid scheduler therefore extends a static data-parallelism compiler in two ways. First, it must partition the application into subgraphs. Second, it must assign subgraphs onto threads and cores. This section discusses the techniques used by the compiler to support our hybrid scheduler. Section 5 presents the runtime techniques for scheduling each of the partitions.

4.1 Partitioning

To partition the application, the compiler runs a breadth-first search on the data-flow graph to find weakly connected components obtained by deleting edges with *dynamic communication channels*. A *dynamic communication channel* is an edge between two operators where either the producer, the consumer, or both have dynamic rate communication.

If there exists an edge with a dynamic communication channel where the endpoints belong to the same weakly connected component, the compiler reports an error stating that the application in invalid. In future work, we are exploring extensions to this algorithm which would allow the compiler to also partition along static edges instead of reporting an error.

The result of the partitioning algorithm is a set of subgraphs that can each be treated as a separate static application. This allows us to leverage the static compiler and optimizer almost *as-is* by running them on each subgraph independently.

We have implemented partitioning as an extension to the StreamIt compiler. As discussed in Section 3, our extension modifies the static data-parallel compilation path of the compiler. Other compilation paths in the compiler support dynamic data rates. However, in contrast to our extension, they do not target SMP multicores, and do not allow either fission or fusion optimizations.

Using StreamIt as a source language has two implications for the partitioning algorithm. First, because the data-flow graphs in StreamIt are hierarchical, the compiler must first turn composite operators into their constituent operators to flatten the data-flow graph. Second, partitioning must respect the topological constraints enforced by the StreamIt language. In StreamIt, the operator-graph must be a *pipeline*, *split-join*, or *feedback-loop* topology. Our current implementation only partitions the graph into pipeline topologies. Although the partitioning algorithm works for other topologies, the StreamIt language would need to add split and join operators that can process tuples out-of-order. A static *round-robin* join operator, for example, would interleave the outputs of dynamic rate operators on its input branches, resulting in errors.

Although dynamic split-join topologies would be useful to implement applications such as an MPEG decoder, which routes i-frames and p-frames along different paths for separate processing, we have found that many applications only use dynamic rates within pipeline topologies at the ingress or egress points of the data-flow graph. This is consistent with the use of dynamic rates for filtering or parsing data before or after some heavy-weight computation. Examples of such applications include automated financial trading, anomaly detection, and graphics pipelines.

4.2 Optimization

Once the graph is partitioned, the compiler can optimize each subgraph independently. Ideally, our modified compiler could treat the static optimizer as a black-box, and simply re-use the existing static data-parallel compilation path to first fuse operators to remove the communication overhead between them, and then data-parallelize (or performs *fission* on, in StreamIt terminology) the fused operators.

However, we had to slightly modify the standard fusion and fission optimizations to support dynamic rates. Our compiler does not data-parallelize operators with dynamic communication channels. In general, they could be parallelized, as long as there were some way to preserve the order of their outputs. We plan to address this in future work. This restriction impacts the fusion optimizations. Operators with dynamic input rates but static output rates are not fused with downstream operators. Although such a transformation would be *safe*, it would not be *profitable* because the fusion would inhibit parallelization. Figure 2 illustrates the changes to the operator data-flow graph during the partitioning and optimization stages of our compiler.

4.3 Placement and Thread Assignment

The static data-parallel compilation path of the StreamIt compiler assigns operators to cores using a greedy bin-packing algorithm that optimizes for spatial locality. That is, the

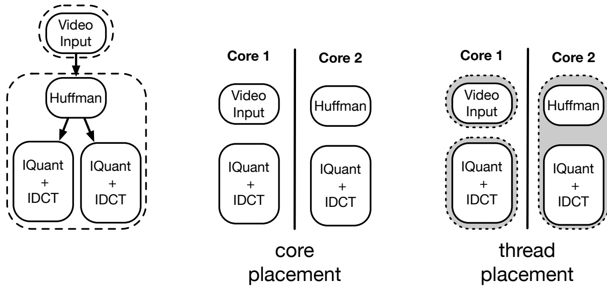


Figure 3: Operators are first assigned to cores and then to threads. Operators on the same thread appear in the same shaded oval.

mapping algorithm tries to place producers and consumers on the same core, while at the same time balancing the workload across available cores. The work estimates come from a static analysis of the operator code [7].

To support dynamic rate partitions, we extend the compiler to not only assign operators to cores, but additionally assign operators to threads. In Figure 3, the partitioned operators are first mapped to cores and then assigned to threads. As will be explained in Section 5.1, each static subgraph is placed on its own thread. Data-parallelized static rate operators on the same core as their producer are placed in the same thread as their producer. Data-parallelized static rate operators not on the same core as their producer are assigned to different threads. In Figure 3, the `VideoInput` and `Huffman` operators are each assigned to separate threads, because they are in separate static subgraphs. The data-parallelized `IQuant+IDCT` operator on core 1 is in its own thread. The data-parallelized `IQuant+IDCT` operator on core 2 is placed on the same thread as its producer.

5. RUNTIME TECHNIQUES

Section 4 discusses the compiler techniques used to support our hybrid scheduler. This section presents the runtime techniques. At runtime, operators in different subgraphs communicate through dynamically-sized queues, adding the flexibility for dynamic rate communication. Within a subgraph, communication is unchanged from the completely static version. Operators communicate through static buffers, even across cores. Each subgraph runs in its own thread, which allows operators to suspend execution midway through a computation if there is no data available on its input queues. Threads run according to the *data-flow order* of the operators they contain, meaning that upstream subgraphs run before downstream subgraphs. This ordering makes it more likely that downstream subgraphs have data available on their input queues when they execute. If data is not available for an operator, the thread blocks, and the next thread runs. Finally, batching is used to reduce the overhead of thread switching.

5.1 Suspending and Resuming Subgraphs

To support dynamic rate communication between operators, we need to consider two questions: (1) what happens if a producer needs to write more data than will fit into an output buffer, and (2) what happens if a consumer needs to read more data than is available from an input buffer?

If a producer needs to write more data than will fit into an output buffer, we need to grow the buffer. In other words, the writer must not block. If a writer could block, then it might never produce enough data for a downstream operator to consume, leading to deadlock. Therefore, we use dynamically-sized queues for communication between the subgraphs. If a producer needs to write more data than will fit into the queue, the queue size is doubled. There is a small performance hit each time a queue needs to be resized. The total number of resizings is logarithmic in the maximum queue size experienced by the application. For most applications, resizing only happens during program startup, as the queues quickly grow to a suitable size. Our current implementation does not decrease queue sizes. In ongoing work, we are investigating adaptive schemes which would adjust the queue size as the workload changes.

Supporting dynamic consumers is more difficult. A stateful operator may run out of data to read partway through a computation. For example, an operator that performs a run-length encoding needs to count the number of consecutive characters in an input sequence. If the data is unavailable for the encoder to read, it needs to store its current character count until it can resume execution. The challenge for dynamic consumers is how to suspend execution, and save any partial state, until more input data becomes available.

To support this behavior, we needed an implementation that is tantamount to coroutines. We chose to use Posix threads that are suspended and resumed with condition variables, although user-level threads would be a viable alternative. Threads are, after all, the standard abstraction for saving the stack and registers. However, using threads had three implications for our design. First, prior versions of StreamIt use one thread per core. We needed to modify the runtime to support running multiple threads per core, one at a time. Second, we needed to add infrastructure for scheduling multiple threads. Finally, switching between threads had a significant negative impact on performance. We needed to explore techniques to offset that impact.

We considered several alternatives to using threads that we thought might incur less of a performance hit. However, we were not able to find a better solution. Closures, such as provided by Objective-C blocks or C++0x lambdas are not sufficient, as they cannot preserve state through a partial execution. We considered adding explicit code to the operators to save the stack and registers, but that code would be brittle (since it is low-level, and breaks abstractions usually hidden by the compiler and runtime system), and not portable across different architectures. A dynamic consumer could invoke an upstream operator directly to produce more data, but the scheduling logic would get complicated as each upstream operator would have to call its predecessor in a chain. On Stack Replacement [10], which stores stack frames on the heap, would work, but there was no readily available implementation to use.

5.2 Scheduling

The code generated by the static data-parallel path of the StreamIt compiler uses only one thread per core. Each operator in the thread executes sequentially in a loop. At the end of each loop iteration, the thread reaches a barrier. The barrier guarantees that all operators are in synch at the beginning of each global iteration of the schedule.

To support dynamic rate communication, we extend the

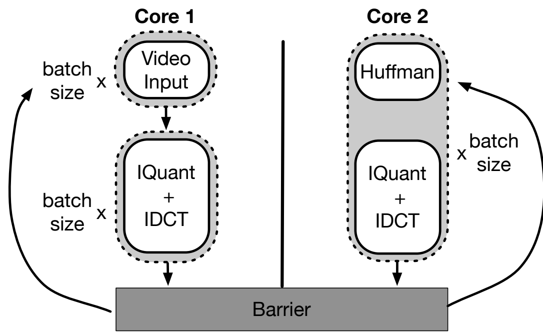


Figure 4: Each thread executes in data-flow order on its assigned core. Each thread is responsible for scheduling the next thread on its core. The solid arrow indicate control transfer.

StreamIt compiler to multiplex multiple threads on the same core. This complicates the scheduler, as it has to coordinate between the various threads.

To prevent multiple threads on a core from being eligible to run at the same time, each thread is guarded by a condition variable. A thread will not run until it is signalled. Our original design had a *master* thread for each core that signalled each thread when they were scheduled to run. However, we found that the biggest performance overhead for our dynamic applications comes from switching threads. To reduce the number of thread switches, we altered our design so that each thread is responsible for signaling the subsequent thread directly. The solid arrows in Figure 4 indicate the transfer of control between threads. Switching from the master thread approach to our direct call approach resulted in an 27% increase in throughput for an application with 32 threads.

The first operator assigned to a thread is the *leader* of that thread. In Figure 4, `Huffman` is the leader of the first thread on core 2. Threads run according to the data-flow order of the leaders. Running in data-flow order makes it more likely that downstream subgraphs have data available on their input queues when they execute.

At program startup, all dynamic queues are empty. As execution proceeds, though, the queues fill up as data travels downstream. This allows for *pipelining*, meaning that downstream operators can execute at the same time as upstream operators. In Figure 4, `Huffman` executes on the data that `VideoInput` processed in the previous iteration. Sometimes, an upstream operator might not produce data. This might occur, for example, with a selection operator that filters data. When this occurs, there is a slight hiccup in the pipelining that resolves when more data travels downstream.

To guard against concurrent accesses to a dynamic queue by producers and consumers, the *push* and *pop* operations are guarded by locks. A lock-free queue implementation would be an attractive alternative to use here, as it could allow for greater concurrent execution [12].

5.3 Batching

As mentioned in Section 5.2, the abandoned master thread approach taught us that the biggest performance overhead for our dynamic applications comes from switching threads. This insight led us to implement the *batching* optimization. With batching, each thread runs for *batch size* iterations before transferring control to the next thread. When the batch size is increased, more data items are stored on each dynamic queue. Batching increases the throughput of the application and reduces thread switching at the expense of increased memory usage and latency. As we will show in Section 6.1.3, running an application with the batch size set to 100 can triple the performance.

6. EVALUATION

Overall, our design strikes a balance between static and dynamic scheduling. It allows for dynamic communication between static components, and for aggressive optimization within the static components.

Because all of the scheduling strategies discussed in this paper are sensitive to variations in both the application structure and the input data set, we first evaluate our system using a set of micro-benchmarks. All of the micro-benchmarks are parameterizable in terms of computation, parallelism, and communication, allowing us to better explore tradeoffs that different scheduling strategies make. The micro-benchmarks are designed to highlight the effects of altering one of these parameters.

Each section starts with the intuition or question that motivates the experiment, followed by a discussion of the setup and results. Section 6.1 evaluates the overhead we can expect for our hybrid scheduler as compared to fully static scheduling. Section 6.2 evaluates what performance improvement we can expect compared to completely dynamic schedulers.

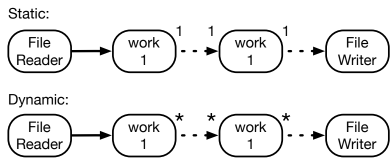
The benchmarks in Section 6.3 model the structure of three applications, and use parameterized workloads for the application logic. These applications make use of a predicate-based filter; an operator for computing volume-weighted average price; and a Huffman encoder and decoder. These experiments help to understand how our scheduler behaves for real-world applications.

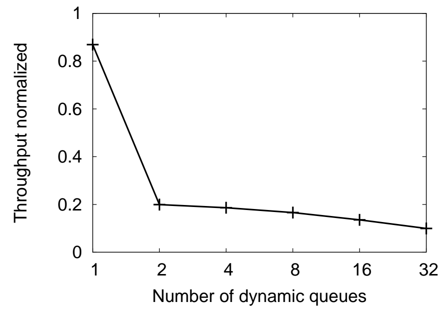
To make the material more accessible, we have grouped the results together in Figures 5, 6, 7, and 8. Each experiment has two figures associated with it. On the left is a topology diagram that illustrates the application that was run in the experiment. On the right is a chart that shows the result. In all topology diagrams, a number to the left of an operator is its static input data rate. A number to the right indicates its static output data rate. An asterisk indicates that the rate is dynamic.

In many of the experiments, we vary the amount of work performed by an operator. One work unit, or one computation, is defined as one iteration of the following loop:

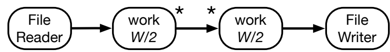
```
1 x = pop();
2 for (i = 0; i < WORK; i++) {
3     x += i * 3.0 - 1.0;
4 }
5 push(x);
```

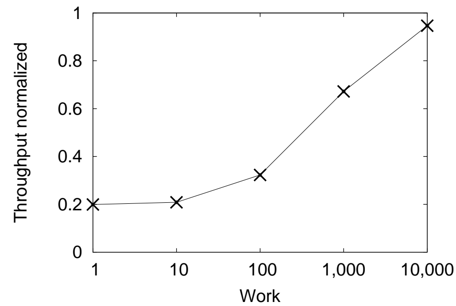
Each subsection below discusses an experiment in detail. All experiments were run on machines with four 64 bit Intel Xeon (X7550) processors, each with 8 cores (for a total of 32), running at 2.00GHz, and an L3 cache size of 18MB. All machines ran Debian 2.6.32-21 with kernel 2.6.32.19. Overall, the results are encouraging. Our hybrid scheduler outperforms three alternative dynamic schedulers: OS thread, demand, and no-op.



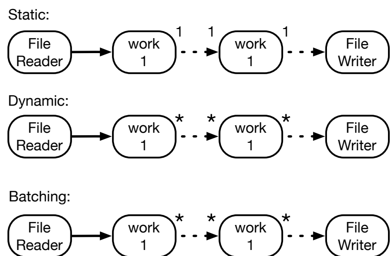


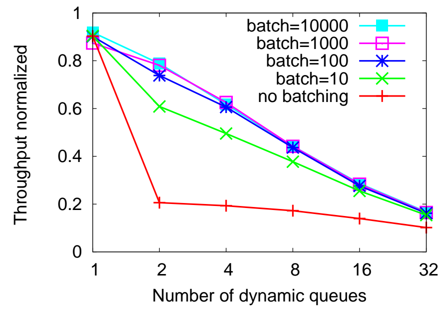
(a) *Dynamic scheduling shows a 5x slowdown with two dynamic queues compared to static scheduling.*



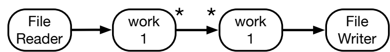


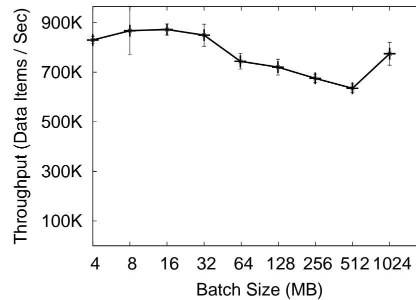
(b) *Larger operator workload amortizes dynamic scheduling overhead.*





(c) *Batching can further ameliorate the effects of thread switching.*





(d) *Batching beyond the cache size hurt performance.*

Figure 5: Experiments with fused operators.

6.1 Comparison to Static Schemes

We expect that the runtime mechanisms used to support dynamic communication will have higher overhead than the fully static equivalents. This is the tradeoff that the hybrid compiler makes in order to get better expressivity. The following set of experiments quantify the overhead of supporting dynamic communication.

6.1.1 Worst-Case Overhead without Batching

How does the communication overhead from dynamism compare to that of the static scheduling?

The worst-case scenario for our scheduler is if the operators do not perform any computation, so the communication overheads cannot be amortized. Figure 5 (a) shows the worst-case overhead for dynamic scheduling as compared to static scheduling. The application is a pipeline of $n + 1$ operators communicating through n dynamic queues. Each operator forwards any data it receives without performing any computation. The results are normalized to a static application, also of $n + 1$ operators, where all operators are fused. The experiment is run on a single core.

The y-axis is the normalized throughput and the x-axis has increasing values of n. As expected, there is significant overhead for adding dynamism. For the simple case of a single dynamic queue, there is a 5x decrease in throughput. The throughput decreases linearly as we add more queues. When there are 31 queues, there is a 10x performance hit.

The biggest detriment to performance comes from switching threads. In the experiment in Section 6.1.3, we show that the overhead from thread switching can be ameliorated by increasing the batch size.

6.1.2 Operator Workload

How does operator workload affect the performance?

Operators for most applications perform more work than in Section 6.1.1. Figure 5 (b) shows the effect of operator workload on our scheduler. The application is a pipeline of two operators communicating through a dynamic queue, running on a single core. We define W as the total workload for the application. Each operator performs $W/2$ computations, and we run the application with increasing workloads.

The results are shown normalized to a static application with two fused operators. The y-axis shows the throughput and the x-axis shows workload. As the operator workload increases, communication overheads are amortized. The 5x overhead with the identity filter improves to 1.48x overhead when the two operators perform 1,000 computations combined. Performance can be further improved with the batching optimization.

6.1.3 Batching

How does batching affect the performance?

In contrast to operator workload, the batch size is fully under control of the system. That is fortunate, because it means we can ameliorate the worst-case behavior from Section 6.1.1. The experiment in Figure 5 (c) demonstrates that batching improves the performance of a dynamic application. It repeats the experiment from Section 6.1.1, with increasing batch sizes. In the chart, each line is the dynamic application run with a different amount of batching.

The graph shows that increasing the batch size can significantly improve the throughput. The 5x overhead with the identity filter improves to 1.64x overhead when the batch size is set to 100. As the batch size increases, so does the throughput. However, as the next experiment shows, there is a limit.

6.1.4 Batching vs. Cache Size

Does batching too much negatively affect the performance?

Batching causes more data to be stored on the dynamic queues. The experiment in Figure 5 (d) tests if increasing the batch size beyond the cache size hurts performance. The application consists of two identity operators in a pipeline.

We ran the experiment with increasing batch sizes, shown in the x-axis. Although there is a lot of variance in the data points, we see that the performance does start to degrade as the batch size outgrows the cache size at 18MB. The performance degradation is not excessive, though, because streaming workloads mostly access memory sequentially, and can therefore benefit from hardware pre-fetching.

6.1.5 Dynamism with Parallelism

How does dynamism affect parallelism?

Adding dynamism to applications introduces bottlenecks into the operator graph, since operators with dynamic communication rates are not parallelized. The experiment in Figure 6 (a) explores how this bottleneck affects performance.

We compare two version of an application: one static and one dynamic. Both version consist of three operators in a pipeline. In the dynamic version, the first and second operators communicate through a dynamic queue. For each data item, the first operator does 100 computations, and the third operator does 900 computations. The second operator simply forwards data.

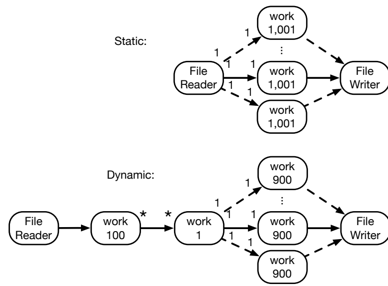
We increased the degree of parallelism for both applications. In the static case, all operators are fused and parallelized. In the dynamic case, only the third operator is parallelized.

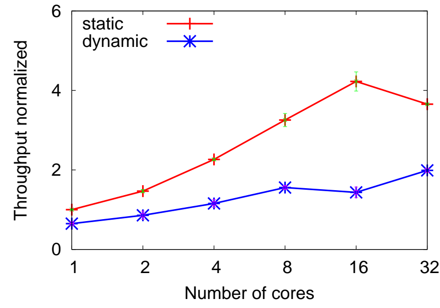
The effects of the bottleneck introduced by the dynamic rate are apparent, as the static case outperforms the dynamic case. However, neither case sees dramatic improvements when parallelized, and indeed the static case sees a drop in performance after 16 cores. There was not sufficient parallelized work to offset the extra communication costs. In the next experiment, we increase the operator workload.

6.1.6 Parallelism and Increased Workload

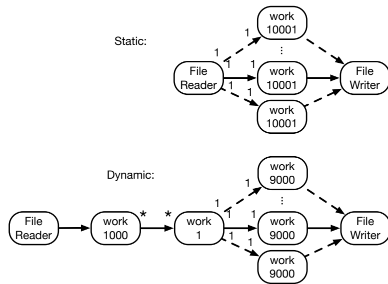
How does operator workload affect the fission optimization?

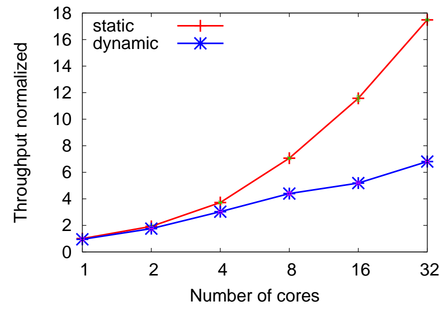
Figure 6 (b) repeats the experiment in Figure 6 (a), but with an increased operator workload. For each data item, the first operator does 1,000 computations, and the third operator does 9,000 computations. The static version of the application effectively parallelizes the work, getting a 17x speedup over the non-parallelized version. The dynamic version also sees a performance improvement, despite the bottleneck, achieving 6.8x increase in throughput.





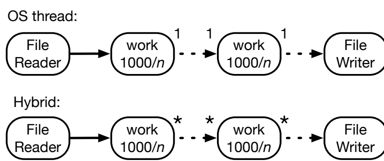
(a) *Dynamic rates introduce a bottleneck for data parallelization.*

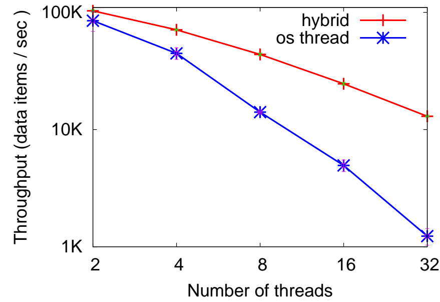




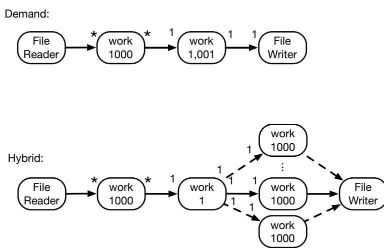
(b) *Larger operator workloads offset the impact of the bottleneck.*

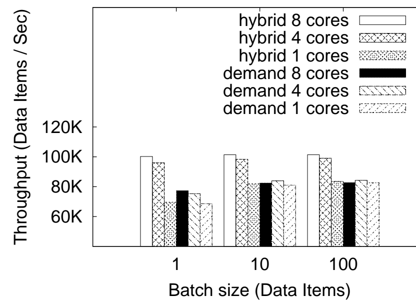
Figure 6: Experiments with parallelized operators.





(a) *The hybrid scheduler outperforms the OS thread scheduler by as much as 10x.*





(b) The hybrid scheduler outperforms the demand scheduler by 1.2x with a modest workload.

Figure 7: Comparing the hybrid scheduler to OS thread and demand scheduling.

6.2 Comparison to Dynamic Schemes

Our hybrid scheduler makes a tradeoff between performance and expressivity, trying to balance both demands. The last section demonstrates that, as expected, adding support for dynamic rate communication hurts performance. The real test of our scheduler, though, is to see if adding the static optimizations yields better performance when compared to other dynamic schedulers. In the following experiments, we compare our hybrid scheduler to OS thread, demand, and no-op schedulers. In all three cases, our hybrid scheduler outperforms the alternative.

6.2.1 OS Thread Scheduling

How does out scheduler compare to OS thread scheduling?

The experiment in Figure 7 (a) compares our hybrid scheduler to an OS thread scheduler. The application consists of n operators arranged in a pipeline. We ran the application with both schedulers on one core with an increasing number of operators. Recall that we ran all experiments on Debian 2.6.32-21 with kernel 2.6.32.19.

All communication between operators is through dynamic queues, and in both the hybrid and OS thread version, each operator executes in its own thread. In the hybrid version, our scheduler controls the scheduling of the threads, so that each thread executes in upstream to downstream order of the operators. In the OS thread scheduler version, the operating system schedules the threads. The results show that the hybrid version significantly outperforms the OS thread approach. In an application with 8 operators, it is 3.1x faster. When there are 32 operators, it is 10.5x faster.

6.2.2 Demand Scheduling

How does our scheduler compare to demand scheduling?

As discussed in Section 3, the demand scheduler in Aurora uses fusion and batching to increase performance, but does not support data-parallelization. It is not an inherent limitation of demand schedulers that they could not support data parallelization. However, it is more difficult to implement data-parallelization for demand schedulers than it is for static schedulers, because it requires machinery to ensure the correct ordering of data. Quantifying the overhead for that machinery is out of scope for this paper. Our comparison is faithful to the Aurora implementation.

The experiment in Figure 7 (b) compares our hybrid scheduler to a demand scheduler. The application consists of three operators in a pipeline. The first does 1,000 computations of work, the second is the identity filter, and the third does 1,000 computations. Since both the hybrid and demand schedulers perform fusion, it does not matter how many operators are downstream from the second operator, as they would be fused into a single operator during optimization. The same application was run in both experiments, but for the demand scheduler, data-parallelization was disabled. Since both the demand scheduler and the hybrid scheduler implement batching, we increased the batch size for different runs of the experiment. We ran both versions of the program on 1, 4, and 8 cores. The hybrid version on 4 and 8 cores outperforms the demand scheduler by 1.2x on 4 cores, and 1.3x on 8 cores. Although these improvements are modest, Section 6.1.6 showed that increasing the workload in the parallelized operators would increase the performance gains of the hybrid scheduler.

6.2.3 No-op Scheduling

How does our scheduler compare to no-op scheduling?

In no-op scheduling, special messages are reserved to indicate that an operator should perform a no-operation. Using this approach, a statically scheduled streaming language can simulate the behavior of a dynamically scheduled language. Because the no-op scheduler is static, it can be optimized with the static optimizer to take advantage of fusion and data-parallelization. However, because replicas receive no-op messages instead of actual work, the workload among replicas is often imbalanced.

To compare our hybrid scheduler with a no-op, we implemented two applications, bargain trade finder and predicate-based filtering, with both systems. The results for this experiment are discussed in detail in Sections 6.3.2 and 6.3.3. In both cases, the hybrid scheduler was about 5x faster than the no-op scheduler.

6.3 Real-World Inspired Benchmarks

The applications in this section are designed to model the structure and workload of three real world applications. The first application, Huffman encoder and decoder, is compared to the demand scheduler described in the previous section. The next two applications, bargain trade finder and predicate based filtering, are compared to a no-op scheduler. In all three experiments, the hybrid approach exhibited improved performance over the alternative approach.

6.3.1 Huffman Encoder and Decoder

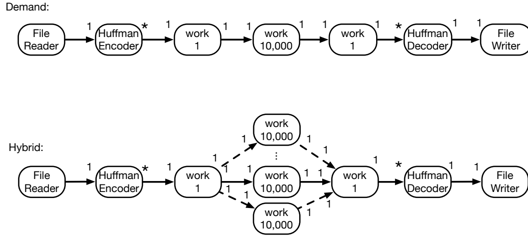
Many audio and video processing applications make use of a Huffman encoder for data compression. It serves as a good example of an operator that, by its very nature, cannot be expressed statically. Implementing a Huffman encoder operator with the no-op scheme, for example, would not make sense, since the operator would always output a maximum length byte string, with some of the bytes filled in as no-op padding, loosing the benefits of the encoding.

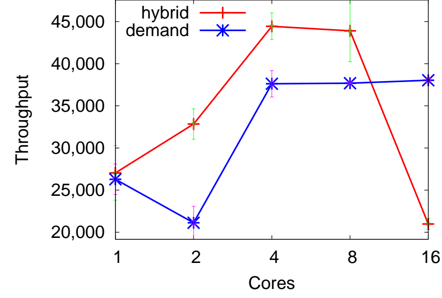
Our synthetic application has a Huffman encoder at the application ingress, and a decoder at the egress. Three operators are in-between the encoder and decoder, emulating additional processing (i.e., busy looping rather than doing the actual computation) that would be performed for data transmission over a lossy channel.

For the experiment, we increased the number of available cores, which allowed our hybrid scheduler to take advantage of the data parallelization. The demand scheduler does not perform data parallelization, which is consistent with the Aurora implementation. Figure 8 (a) shows the results. The hybrid scheme outperforms the demand scheduler by 20% when run on 8 cores. However, when we ran the experiment on 16 cores, we exceeded the benefits of parallelization.

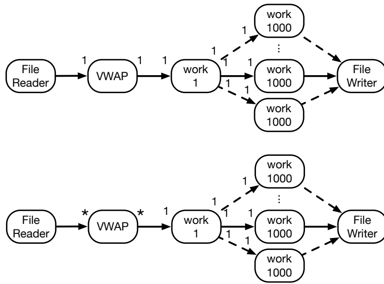
6.3.2 Bargain Trade Finder

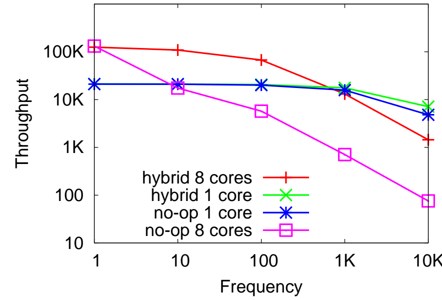
A *volume weighted average price*, or VWAP, is a computation often used in financial applications. It keeps a sum of both the price of trades and the volume of trades that occur during a given time window. By time we refer to application time, as embodied by a time attribute of each data item. And by window, we refer to a sequence of data items



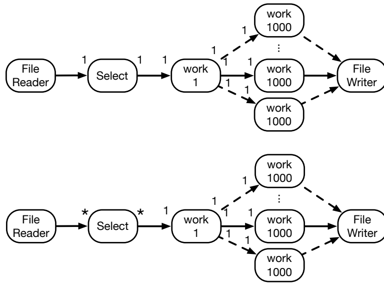


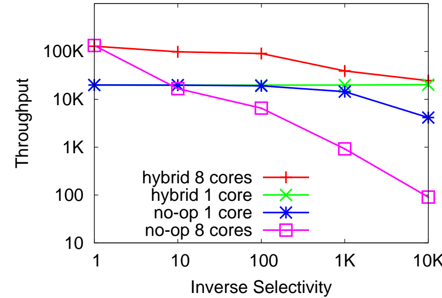
(a) *The hybrid scheduler outperforms the demand scheduler by 1.2x for a Huffman encoder and decoder..*





(b) *The hybrid scheduler can perform 5.1x faster than the no-op scheduler for applications that use VWAP.*





(c) *The hybrid scheduler can perform 4.9x faster than the no-op scheduler for predicate-based filtering.*

Figure 8: Experiments with real-world inspired synthetic benchmarks.

for which the time attribute differs at most by a prescribed amount. At the end of that window, it sends the average.

A hedge fund might use a VWAP operator in an application for finding bargains transactions [8]. A bargain occurs when a stock price dips below its average price over a past time window. To emulate the bargain finder application, we placed three operators in a pipeline. The first operator is the VWAP, for computing the rolling average. The remaining two operators represent the static work needed to perform a transaction once a bargain has been identified.

We implemented the bargain finder application using both our hybrid scheduler, and a no-op scheduler. Both schedulers parallelize the third operator. For both configurations, we varied the frequency of the input data. In the graphs in Figure 8 (b), a frequency of 10 means that on average every 10th input data item would produce the next window. That is, a total of 10 trades appear in the time window.

Figure 8 (b) shows the results. The hybrid scheme on 8 cores shows a 5.1x performance improvement over the best no-op version when the frequency is 10. When the frequency is greater than 1,000, the hybrid scheme on 1 core exhibits better throughput than the other configurations.

6.3.3 *Predicate-Based Filtering*

Many applications are only interested in processing *significant events*, where the definition of significant is application-dependent. For these applications, it is necessary to filter data based on some predicate. For example, ocean-based sensors constantly monitor tidal heights. If the tidal height exceeds some threshold, it can then be correlated with other

measurements, such as wind speed and barometric pressure, to predict the onset of a storm.

To emulate this application, we implemented an operator that that scans incoming data for items that match a predicate (e.g., data items above a threshold). The application consists of three operators. The first is the filter, and the remaining two operators process the significant events.

For the experiment, we varied the selectivity of the data. In Figure 8 (c), an inverse selectivity of 10 means that on average, for every 10 inputs, the selection filters 9. Put another way, 1 in 10 inputs would result in meaningful downstream computation. The hybrid scheduler shows significant performance improvements over the no-op version. When the inverse selectivity is 10, the hybrid scheduler shows a 4.9x higher performance compared to the best no-op version.

7. OUTLOOK AND CONCLUSION

This paper presents the design of a hybrid static/dynamic scheduler for streaming languages. Stream processing has become an essential programming paradigm for applications that process large volumes of data with high throughput.

In ongoing work, we are investigating mapping these hybrid scheduling techniques to a distributed architecture. Our current prototype uses shared memory segments for dynamic queues, but other implementations are possible. For example, prior work on distributed versions of StreamIt used remote memory spaces to run on the TILERA [18] architecture, or sockets in a Java-library implementation [15].

The first contribution of this paper is to explore the tradeoffs between dynamic and static scheduling. While statically scheduled languages allow for more aggressive optimization, dynamically scheduled languages are more expressive, and can be used to write a wider range of applications, including applications for compression/decompression, event monitoring, networking, and parsing.

The second contribution of this paper is the design of a hybrid static-dynamic scheduler for stream processing languages. The scheduler partitions the streaming application into static subgraphs separated by dynamic rate boundaries, and then applies static optimizations to those subgraphs. Each static subgraph is assigned its own thread, and the scheduler executes the threads such that upstream operators execute before downstream operators.

The third contribution of this paper is an implementation and evaluation of our hybrid scheduler in the context of the StreamIt language. When compared against three alternative dynamic scheduling techniques, OS thread, demand, and no-op, our scheduler exhibited better performance.

In summary, our approach shows significant speedup over fully dynamic scheduling, while allowing developers to write a larger set of applications. We believe that our scheduler strikes the right balance between expressivity and performance for stream processing languages.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We thank Bill Thies for his feedback on this work, and Robert Kleinberg for his suggestions on Section 4.1. This work is partially supported by NSF CCF-1162444.

8. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[3] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 89–108, Oct. 2010.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *Proc. ACM International Conference on Computer Graphics and Interactive Techniques*, pages 777–786, Aug. 2004.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, Oct. 2006.

[7] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proc. 10th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Dec. 2002.

[8] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. Streams processing language specification. Research Report RC24897, IBM, Nov. 2009.

[9] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. Technical Report RC25215, IBM, Sept. 2011.

[10] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 32–43, 1992.

[11] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proc. 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, Sept. 2009.

[12] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[13] M. Roesch. Snort- lightweight intrusion detection for networks. In *Proc. 13th USENIX Large Installation System Administration Conference*, pages 229–238, 1999.

[14] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 115–126, June 2005.

[15] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 179–196, Apr. 2002.

[16] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, et al. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, 1997.

[17] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Oct. 2001.

[18] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, Sept. 2007.