Hitting the Sweet Spot for Streaming Languages: Dynamic Expressivity with Static Optimization

NYU CS Technical Report TR2012-948

Robert Soulé, New York University, soule@cs.nyu.edu
Michael I. Gordon, Massachusetts Institute of Technology, mgordon@mit.edu
Saman Amarasinghe, Massachusetts Institute of Technology, saman@mit.edu
Robert Grimm, New York University, rgrimm@cs.nyu.edu
Martin Hirzel, IBM Research, hirzel@us.ibm.com

Abstract

Developers increasingly use stream processing languages to write applications that process large volumes of data with high throughput. Unfortunately, when choosing which stream processing 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 cannot express 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, and evaluated its performance against three scheduling techniques used by dynamic systems: OS thread, demand, and no-op. 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.

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 [15], can be used to write a broader range of applications. Unfortunately, many optimizations cannot be applied dynamically without incurring large runtime costs [10]. 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 StreamIt [13], Esterel [4], Brook [5], and Lime [3] 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. time-based or attribute-delta based windows).
- *Networking.* Software routers and network monitors such as Snort [11] require data-dependent routing.
- *Parsing/Extraction.* Examples include tokenization (e.g. input string, output words), twitter analysis (e.g. input

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

While these applications fundamentally need dynamism, only 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. Events arriving from a financial exchange occur at irregular time intervals, but the 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. Each static subgraph is assigned its own thread, and our scheduler executes the threads such that upstream components execute before downstream components, maximizing throughput [1].

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. 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 stream processing 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 show significant speedup over fully dynamic scheduling, while allowing stream developers to write a larger set of applications. We believe our scheduler occupies the sweet spot between expressivity and performance for streaming languages.

2. Related Work

Table 1 presents an overview of various approaches for scheduling stream processing languages. The simplest approach is **sequential scheduling**. All operators are placed into a single thread, with no support for parallel execution. The StreamIt Library [13] 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 [1], and is similar to the approach used by the SEDA [15] 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* [8], 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 an optimal order, so there are frequent cache-misses, unnecessary thread switching, 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* [8], by placing multiple operators on the same thread to execute. Fusion reduces communication overhead, and the frequency of thread switching. Like Aurora, our 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 and 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*. Therefore, 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.

In the **hardware pipelining** strategy, 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

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. Latency at the expense of throughput.
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.

this approach [7] for several different target platforms, including clusters of workstations [12], the MIT Raw microprocessor [14], and Tilera's line of microprocessors [16]. 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. 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 path 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** implementation extends the static data-parallelism path of the StreamIt compiler. Throughout the rest of this paper, the term *StreamIt* refers to the static data-parallel version of the StreamIt compiler that targets SMP multicores. 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 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.

3. 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. The compiler can then treat each subgraph as if it were a separate static application. This means that within a subgraph, operators communicate though static buffers, and the compiler can statically optimize each subgraph independently of the rest of the application. The compiler is also responsible for placing subgraphs onto threads and cores. This section discusses the techniques

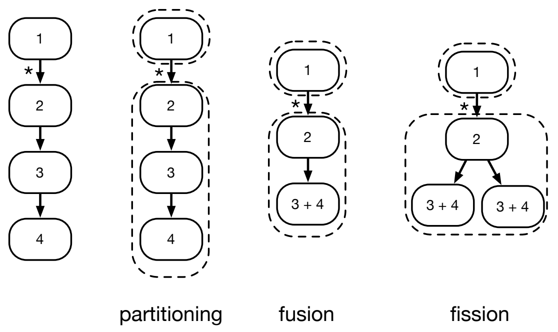


Figure 1: The data-flow graph is first partitioned into static subgraphs. Solid edges are streams from producers to consumers. The asterisk indicates a dynamic communication channel, and the dashed line indicates the static subgraphs. Each static subgraph is optimized by fusing and then parallelizing.

used by the compiler to support our hybrid scheduler, while Section 4 presents the runtime techniques.

3.1 Partitioning

To partition the application, the compiler cuts the data-flow graph so that each subgraph can 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. A *dynamic communication channel* is an edge between two operators where either the producer, the consumer, or both have dynamic rate communication. The partitioning algorithm is simply a breadth-first search of the data-flow graph that cuts an edge if it is a dynamic communication channel. If there is a branch in the graph, such as after a split operator, then there must be a cut on every branch. The criterion is that a cut must completely bisect the graph. Cuts that do not bisect the graph are illegal, because they do not produce independent static subgraphs.

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 flatten the data-flow graph to remove the hierarchy before performing partitioning. 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 permits cuts in 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. Dynamic split-join topologies are necessary for applications such as an MPEG decoder, which routes i-frames and p-frames along different paths for separate processing.

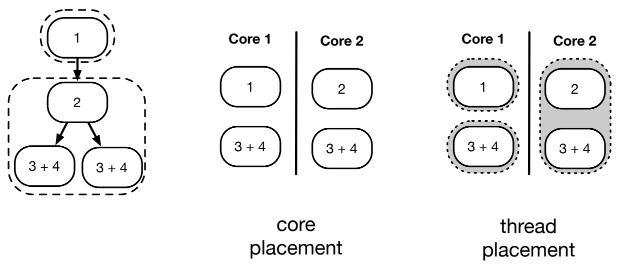


Figure 2: Operators are first mapped to cores and then assigned to threads. Operators on the same thread appear in the same shaded oval.

3.2 Optimization

Once the graph is partitioned, the compiler can optimize each subgraph independently. The StreamIt compiler first aggressively fuses operators to remove the communication overhead between them, and then data-parallelizes (or performs *fission* on) the fused operators. Figure 1 illustrates the changes to the operator data-flow graph during the partitioning and optimization stages of our compiler.

Ideally, our modified compiler could treat the static optimizer as a black-box. However, adding dynamic rates modifies the standard fusion and fission optimizations. Operators with dynamic communication channels are not data-parallelized. In general, they could be, 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.

3.3 Placement

The StreamIt compiler assigns operators to cores using a greedy bin-packing algorithm that respects spatial constraints. That is, the mapping algorithm tries to place producers and consumers on the same core, while at the same time balancing the workload across available cores.

We extend the compiler to not only map operators to cores, but additionally assigns operators to threads. In Figure 2, the partitioned operators are first mapped to cores and then assigned to threads. As will be explained in Section 4.1, 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 their own thread. In Figure 2, operators 1 and 2 are each assigned to separate threads, because they are in separate static subgraphs. The data-parallelized 3+4 operator on core 1 is in its own thread.

The data-parallelized 3+4 operator on core 2 is placed on the same thread as its producer.

4. Runtime Techniques

Section 3 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.

4.1 Coroutines

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 on the input buffer?

To support the producers, 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. This means that a producer can always write to its output communication channel. There is a small performance hit each time a queue needs to be resized. The total number of resizing 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.

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 need coroutines. We considered several approaches in search of a lightweight 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 [9], which stores stack frames on the heap, would work, but there was no readily available implementation to use.

Ultimately, we chose to use user-level threads. Threads are, after all, the standard abstraction for saving the stack and registers. However, using threads had several 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. Second, we needed to add infrastructure for scheduling multiple threads. And finally, we needed to offset the performance impact that results from thread switching.

4.2 Scheduling

StreamIt 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 StreamIt compiler to use multiple threads per core. This complicates the scheduler, as it has to coordinate between the various threads. To prevent multiple threads on a core from running 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 3 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 3, operator 2 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 3, operator 2 executes on the data that operator 1 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

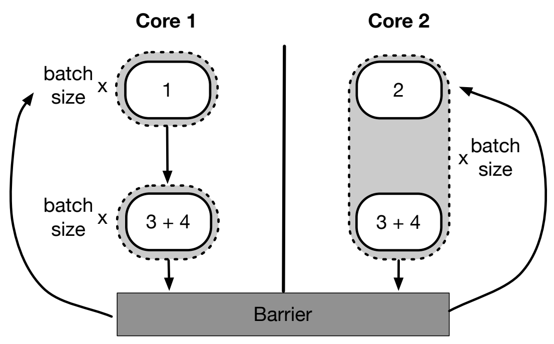


Figure 3: Each thread executes in data-flow order on its assigned core. Each thread is responsible for scheduling the next thread on its core. The solid arrow indicate control transfer.

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 choice to use here, as it could allow for greater concurrent execution, but unfortunately, we are not aware of any lock-free queues that support a lock-free *resize* operation.

4.3 Batching

As mentioned in Section 4.2, 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 5.1.3, running an application with the batch size set to 100 can triple the performance.

5. 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. To evaluate our system, we explore two issues:

- Section 5.1 evaluates the overhead we can expect for our hybrid scheduler as compared to fully static scheduling.
- Section 5.2 evaluates what performance improvement we can expect compared to completely dynamic schedulers.

For each question, we ran a set of micro-benchmarks. Because there are a lot of experiments, we have grouped the results together in Figures 4, 5, 6, and 7 to make the material more accessible. 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 declares its static input data rate. A number to the right indicates is 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:

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

Each subsection below discusses an experiment in detail. All experiments were run on a 64 bit Intel Xeon (X7550) processor with 32 cores running at 3.00GHz, and an L3 cache size of 18MB. Each section starts with the intuition or question that motivates the experiment, followed by a discussion of the setup and results. Overall, the results are encouraging. Our hybrid scheduler outperforms three alternative dynamic schedulers: OS thread, demand, and no-op.

5.1 Comparing Dynamic to Static

We expect that our hybrid compiler will be less performant than a fully static compiler. This is the tradeoff that the compiler makes in order to get better expressivity. The following set of experiments quantify the overhead of our hybrid scheduler when compared to fully static scheduling.

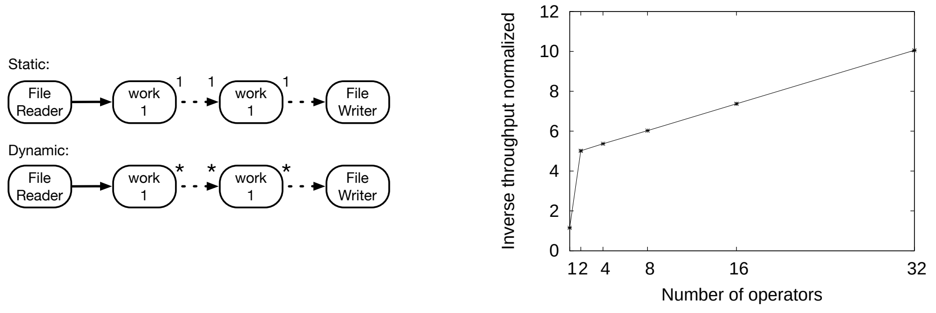
5.1.1 Worst-Case Overhead without Batching

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

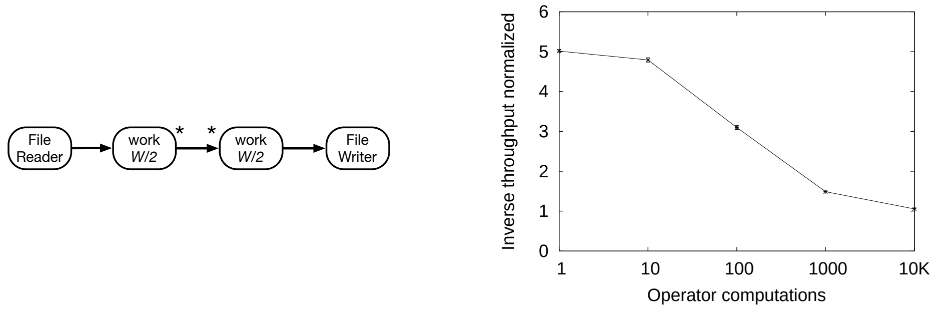
The worst-case scenario for our scheduler is if the operators do not perform any computation, so the communication overheads cannot be amortized. The experiment in Figure 4 (a) shows shows the worst-case overhead for dynamic scheduling as compared to static scheduling. The application is a pipeline of n operators communicating through dynamic queues. Each operator simply forwards any data it receives without performing any computation. The results are normalized to a static application, also of n operators, where all operators are fused. The experiment is run on a single core.

In the figure, the y-axis is the inverse 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 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.

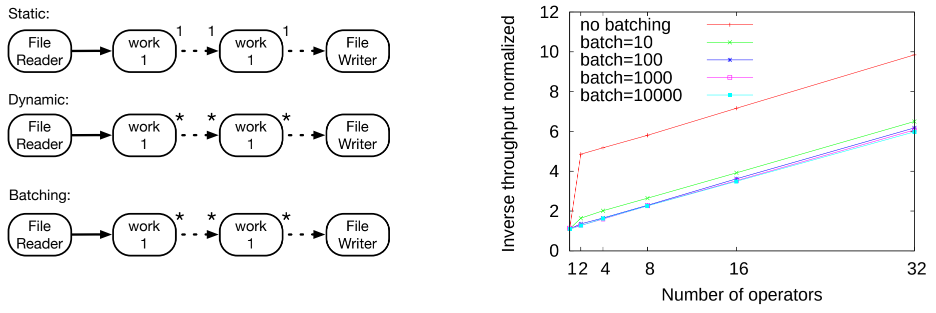
The biggest detriment to performance comes from switching threads. In the experiment in Section 5.1.3, we show that



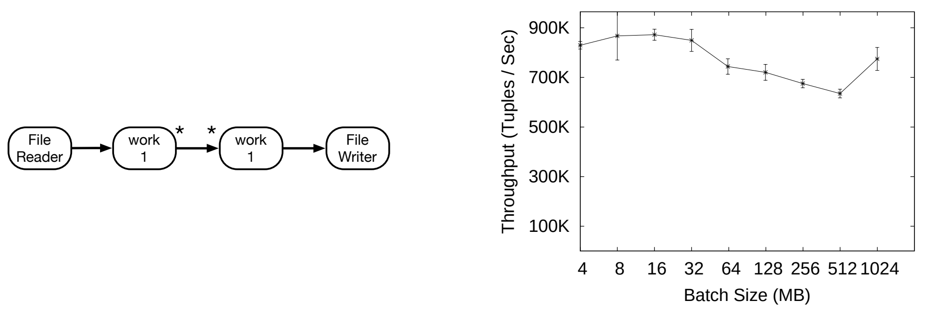
(a) *Dynamic scheduling show a 5x slowdown with two threads compared to static scheduling.*



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



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



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

Figure 4: Experiments with fused operators.

the overhead from thread switching can be ameliorated by increasing the batch size.

5.1.2 Operator Workload

How does the operator workload affect the performance?

Operators for most applications perform more work than in Section 5.1.1. The experiment in Figure 4 (b) explores the effect of operator workload on for our scheduler. The application is a pipeline of two operators communicating through a dynamic queue, running on a single core. We define the total workload for the application to be W, where each operator performs $W/2$ computations. and ran the application with increasing workloads.

The results are shown normalized to a static application with two fused operators. The y-axis shows the inverse throughput and the x-axis shows workload. As the 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.

5.1.3 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 5.1.1. The experiment in Figure 4 (c) demonstrates that batching improves the performance of a dynamic application. It repeats the experiment from Section 5.1.1, 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.

5.1.4 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 4 (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.

5.1.5 Dynamism with Parallelism

How does dynamism affect parallelism?

Adding dynamism to StreamIt applications introduces bottlenecks into the operator graph, since operators that have dynamic communication rates are not parallelized. The experiment in Figure 5 (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 ran both applications with increasing degrees of parallelism. In the static case, all operator are fused, and then 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.

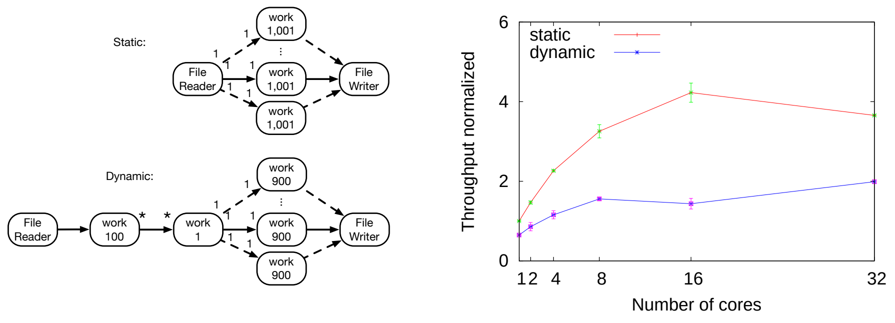
5.1.6 Dynamism with Parallelism and Increased Workload

How does the operator workload affect the performance of fission?

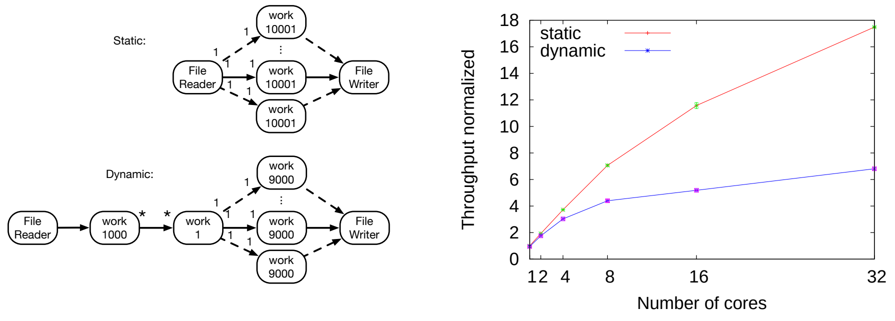
Figure 5 (b) repeats the experiment in Figure 5 (a), but with an increased operator workload. For each data item, the first operator does 1000 computations, and the third operator does 9000 computations. The static version of the application effectively parallelizes the work, getting a 17x speedup over the non-parallelized version. The dynamic version also sees a performance improvement, despite the bottleneck, achieving 6.8x increase in throughput.

5.2 Alternative Dynamic Scheduling techniques

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 an increase in 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.



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



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

Figure 5: Experiments with parallelized operators.

5.2.1 Thread O.S. Scheduling

How does our implementation compare to OS thread scheduling?

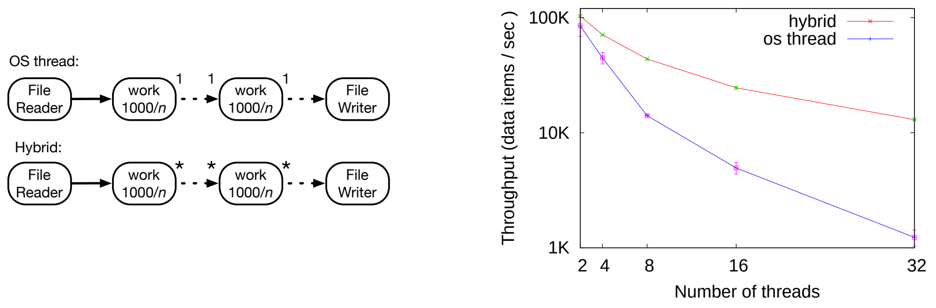
The experiment in Figure 6 (a) compares our hybrid scheduler to an OS thread scheduler. The application consists of n operators arranged in a pipeline. We ran the application with both schedulers on one core with an increasing number of operators.

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.

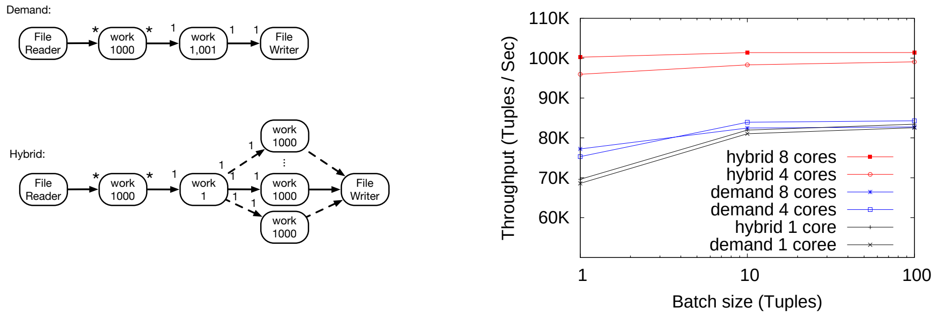
5.2.2 Demand Scheduling

How does our implementation compare to demand scheduling?

As discussed in the Section 2, the demand scheduler implemented by Aurora uses fusion and batching to increase performance, but does not support data-parallelization. To compare our hybrid scheduler to a demand scheduler, we ran the experiment in Figure 6 (b). The application consists of three operators arranged in a pipeline. The first does 1000 computations of work, the second is the identity filter, and the third does 1000 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 5.1.6 showed that increasing the workload in the parallelized operators would increase the performance gains of the hybrid scheduler.



(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 6: Comparing the hybrid scheduler to OS thread and demand scheduling.

5.2.3 No-op Scheduling

How does our implementation 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 see how our hybrid scheduler compares to no-op scheduling, we implemented no-op versions of two dynamic operators. The first is a *selection*, which filters values by a predicate. The second operator is a *volume weighted average price*, or VWAP. 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. At the end of that window, it sends the average.

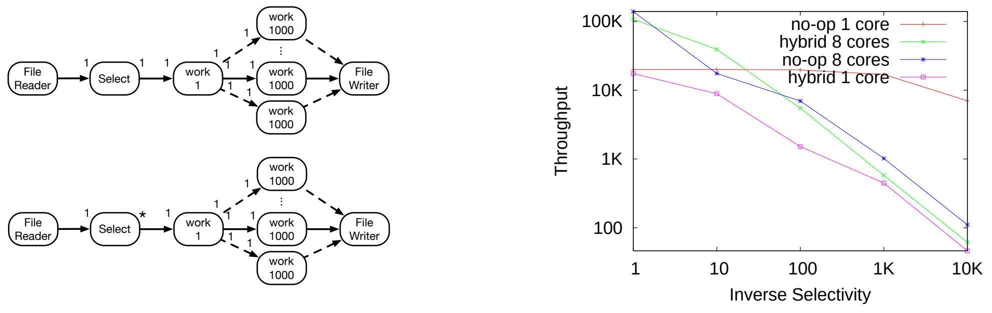
Both the VWAP and selection operator we placed in an application consisting of three operators in a pipeline. The VWAP or selection was first, followed by the identity operator, followed by an operator that processed the data by performing 1000 computations. Both the no-op scheduler and the hybrid scheduler parallelize the third operator.

For the selection experiment, we varied the selectivity of the input data. In the graphs in Figure 7 (a) and (c), a selectivity of 10 means that for every 10 inputs, the selection filters 9. Put another way, 1 in 10 inputs would result in meaningful downstream computation.

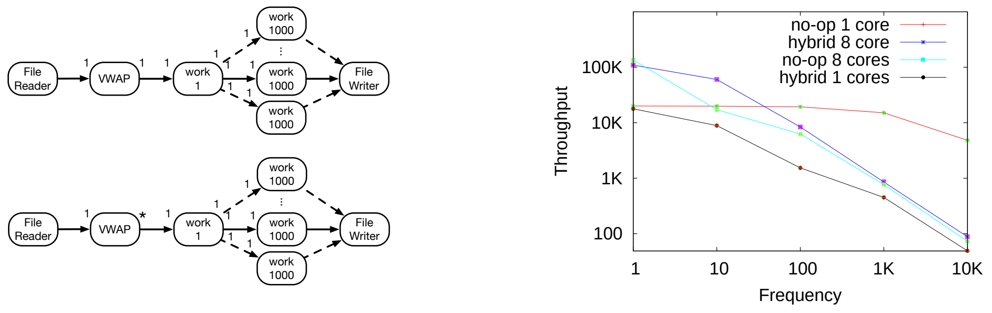
For the vwap experiment, we varied the frequency of the input data. In the graphs in Figure 7 (b) and (d), a frequency of 10 means the 10th input data item would results in the next window. That is, a total of 10 trades appear in the time window.

Figure 7 (a) and (b) show the first version of this experiment. In this version, the no-op scheduler and hybrid scheduler have fairly comparable performance. The hybrid scheduler performs better when the selectivity and frequency is set to 10. However, when the selectivity and frequency are higher, the no-op scheduler on one core performs the best. The no-op scheduler on 8 cores does not perform well, because of the load imbalance. The hybrid scheduler does not perform well, because there are frequent thread switches that do not produce meaningful work.

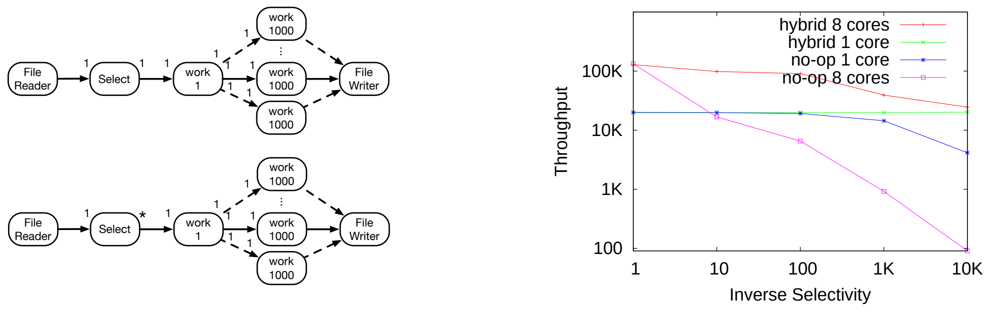
However, because the hybrid scheduler offers true dynamism, as opposed to simulated dynamism, it has a signif-



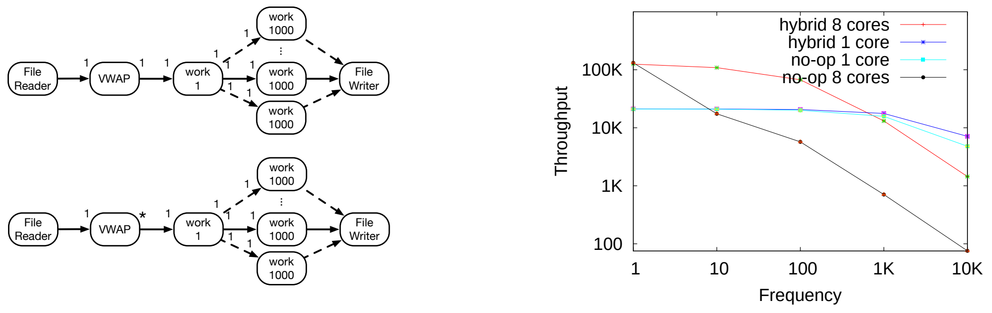
(a) *For the selection operator, the performance of both schedulers depends on the selectivity.*



(c) *For the VWAP operator, the performance of both schedulers depends on the frequency.*



(b) *When the selection operator has a dynamic input rate, the hybrid scheduler can perform 4.9x faster.*



(d) *When the VWAP operator has a dynamic input rate, the hybrid scheduler can perform 5.1x faster.*

Figure 7: Comparing the hybrid scheduler to no-op scheduling.

icant advantage over the no-op scheduler. Both the selection and VWAP operator can be re-written to add dynamism at the operators input. For example, the selection operator can be re-written like this:

```
while (peek(0) > THRESHOLD) {
    pop();
}
push(pop());
```

When the operators are written this way, they will read from their input queue until there is no data available, reducing the number of thread switches. The experiments in Figure 7 (c) and (d) answer the question:

How does our implementation with dynamic input compare to no-op scheduling?

The dynamic input versions of the operators show significant performance improvements over the no-op versions. When the selectivity is 10, the hybrid VWAP operator shows a 4.9x performance compared to the best no-op version (1 core). They hybrid selection operator on 8 cores shows a 5.1x performance over the best no-op version (1 core) when the selectivity is 10.

6. Conclusion

This paper presents the design of a hybrid static/dynamic scheduler for stream processing languages. Stream processing has become an essential programming paradigm for applications that process large-volumes of data with high throughput. Stream processing languages are widely used for applications in government, finance, and entertainment.

The first contribution of this paper is to explore the tradeoffs between dynamically scheduled and statically scheduled languages. 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 subgraphs 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 of our hybrid scheduler for the StreamIt language. To evaluate our system, we compared it with three alternative dynamic scheduling techniques: OS thread, demand, and no-op. In all three cases, our hybrid scheduler outperformed the alternative.

In summary, our approach show 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.

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, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. Research Report RC25215, IBM, Sept. 2011.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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