Cache-conscious scheduling of streaming applications

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.
ABSTRACT

This paper considers the problem of scheduling streaming applications on uniprocessors in order to minimize the number of cache-misses. Streaming applications are represented as a directed graph (or multigraph), where nodes are computation modules and edges are channels. When a module fires, it consumes some data-items from its input channels and produces some items on its output channels. In addition, each module may have some state (either code or data) which represents the memory locations that must be loaded into cache in order to execute the module. We consider synchronous dataflow graphs where the input and output rates of modules are known in advance and do not change during execution. We also assume that the state size of modules is known in advance.

Our main contribution is to show that for a large and important class of streaming computations, cache-efficient scheduling is essentially equivalent to solving a constrained graph partitioning problem. A streaming computation from this class has a cache-efficient schedule if and only if its graph has a low-bandwidth partition of the modules into components (subgraphs) whose total state fits within the cache, where the bandwidth of the partition is the number of data items that cross intercomponent channels per data item that enters the graph.

Given a good partition, we describe a runtime strategy for scheduling two classes of streaming graphs: pipelines, where the graph consists of a single directed chain, and a fairly general class of directed acyclic graphs (dags) with some additional restrictions. The runtime scheduling strategy consists of adding large external buffers at the input and output edges of each component, allowing each component to be executed many times. Partitioning enables a reduction in cache misses in two ways. First, any items that are generated on edges internal to subgraphs are never written out to memory, but remain in cache. Second, each subgraph is executed many times, allowing the state to be reused.

We prove the optimality of this runtime scheduling for all pipelines and for dags that meet certain conditions on buffer-size requirements. Specifically, we show that with constant-factor memory augmentation, partitioning on these graphs guarantees the optimal number of cache misses to within a constant factor. For the pipeline case, we also prove that such a partition can be found in polynomial time. For the dags we prove optimality if a good partition is provided; the partitioning problem itself is NP-complete.

Categories and Subject Descriptors
F.2 [Analysis of Algorithms and Problem Complexity]: General

General Terms
Algorithms, Performance, Theory

Keywords
Caching, dag, graph, partitioning, pipelining, scheduling, streaming, synchronous dataflow.

1. INTRODUCTION

Streaming programming models have received enormous attention in recent years, mostly because they are perceived as enablers of high-performance computing and because of the continuing shift from signal processing by analog hardware to digital signal processing. Examples include academic projects like StreamIt [27] and StreamC/KernelC [12], community-based open-source projects like GNU Radio [9], and commercial products including Simulink® [20] and LabVIEW [16].

Naturally, there has been extensive research on how to map these programming models to hardware and on how to schedule them so as to maximize various objective functions, often subject to constraints. Much of the research has focused on maximizing throughput, defined as the number of data items processed per unit time, and on minimizing delay or latency (the period between the time an input data item enters the computation and the time it affects an output
data item) [6, 13]. More recently, additional metrics have been considered, such as power consumption [29], memory usage [4, 23], and cache efficiency [15, 21, 25].

Because streaming computational models are restricted, it is possible to map and schedule them efficiently. A streaming application can be represented as a graph (or multigraph) where nodes are computation modules and the edges are channels connecting the modules. Channels have buffers associated with them. The modules send data in the form of tokens (also called messages or items),¹ to each other via these channels. When a module fires, it consumes some tokens from its incoming channels, performs some computation, and produces some tokens on its outgoing channels. A module is ready to fire each time each of its input channels contains sufficient tokens to enable it to compute. Each module accesses some associated state, almost always of static size (but not identical across modules), when it fires. Designated channels stream (a possibly infinite number of) tokens into and out of the application.

This paper focuses on cache-efficient scheduling of streaming computations on a single processor. Because cache efficiency is such an important determinant of performance, there is a rich history of algorithms and data structures designed to minimize the number of cache misses incurred by a program (see [1, 3, 5, 7, 11, 24] for a sample). Cache efficiency is often overlooked in streaming models, since data items in channels are written exactly once and read exactly once, and they are not reused. Streaming applications exhibit two kinds of cache misses, however, that can be controlled using intelligent scheduling. First, since modules access their state when they fire, it may be advantageous to execute the same module many times once its state has been brought into the cache. Second, consider a module A which outputs some data item on a channel between itself and a module B. If modules A and B are scheduled in a quick succession, then this data item remains in cache. If they are not, then this data item may be spilled out to main memory, leading to a cache miss when module B eventually reads it. Therefore, it is advantageous to execute consecutive modules one after the other whenever possible. Minimizing one of these two kinds of cache misses may compete with the objective of minimizing the other, however, and we must balance the concerns intelligently while scheduling streaming applications.

The algorithms in this paper for minimizing the number of cache misses are based on the idea of partitioning. The streaming graph is divided into subgraphs, each of which fits in cache. That is, given a cache of size $M$, the sum of the state size in any single subgraph does not exceed $M$. Within subgraphs, we use small buffers (the minimum required on that edge), whereas on channels between subgraphs, we use large buffers. This approach reduces cache misses in two ways. After a particular subgraph is cached, it can be executed any number of times — as long as sufficient input data is available — without causing any cache misses to load or spill state. Additionally, each module of the subgraph is executed so that the data moving through the subgraph remains cached, with the exception of the initial input and final output. The key to minimizing cache misses using this method is to partition the modules to minimize the amount of data transferred through the channels that cross between subgraphs.

We assume that each time a module fires, the number of items it produces and consumes from each of its input and output channels is known in advance and that this value does not change during execution. Such computations are called synchronous dataflow graphs. Without loss of generality, we also assume that all data items are unit sized. Finally, we assume that the state size of each module is known in advance. These assumptions are satisfied by a wide range of streaming computations.²

This paper makes the following contributions:

- A theoretical argument that provides a lower bound for the cost of scheduling a directed acyclic graph (dag). Specifically, consider an optimal “well-ordered” partition $P$ into subgraphs each with $O(M)$ total state, where “well-ordered” means that the graph induced by contracting each component is acyclic. An optimal partition is one with minimum “bandwidth,” which is the total number of messages that must cross partition boundaries for each input item consumed. Then every schedule incurs an amortized cost of at least $\Omega((1/B)\text{ bandwidth}(P))$ cache misses per input item, where $M$ is the cache size, $B$ is the block size.

- An upper bound for certain dags. Specifically, suppose that we are given a well-ordered partition $P$ subject to certain additional restrictions. We show how to schedule the dag on a machine with cache size $O(M)$ and block size $B$ with $O((1/B)\text{ bandwidth}(P))$ amortized cache misses per input item. If $P$ is optimal, the upper and lower bound match to within constant factors. Therefore, the partitioned scheduler incurs at most a constant factor more cache misses than the optimal scheduler if given a constant-factor larger cache.

- We also show how to find a good partition for the case of pipelines, where the streaming graph consists of a single directed chain of modules. For more general dag topologies, finding the minimum bandwidth well-ordered partition is NP-complete [8, ND15: Acyclic Partition]. The additional restrictions for the upper bound require the partition to have $O(M/B)$ edges leaving each subgraph, and each component to be schedulable with internal buffers having a total of $O(M)$ size. The latter condition holds when all modules have uniform input and output rates, and both conditions always hold for the pipeline even without uniform rates. Both conditions also hold for a wide class of dags without uniform rates.

²Some streaming applications satisfy our assumptions except for very few modules, such as modules that extract symbols or packets from a waveform, modules that produce television frames from a compressed stream, and so on. Our techniques can still be used to schedule these computations, perhaps suboptimally, by forcing these models to the boundaries of subgraphs.

2. MODEL AND DEFINITIONS

This section describes the analysis and streaming models. We discuss assumptions about the streaming graphs used throughout the paper and define needed terminology.

Analysis model

To analyze the cost of a schedule for a streaming application, we use the external-memory model or I/O model [1], sometimes also called the disk-access model (DAM).
I/O model is a two-level memory hierarchy consisting of a fast internal memory (cache) of size $M$ and a slow, arbitrarily large external memory (disk), organized into contiguous blocks of size $B$. An algorithm may only operate on data that resides in the cache — requesting data that is not in cache causes a cache miss, wherein the block containing the data must first be moved from disk to cache (likely causing other data to be evicted from the bounded cache). The cost in this model is the number of cache misses.

**Streaming model**

We model a streaming computation as a directed acyclic graph (dag) $G = (V, E)$. Each vertex $v \in V$ corresponds to a module, which has a predefined state size denoted by $s(v)$. Each edge corresponds to a directed channel between modules, with buffers (implementing FIFO queues) along each edge to store messages that have not yet been consumed by the receiving module. A module has (integral) parameters $\text{in}(u, v)$ and $\text{out}(v, w)$ specifying, respectively, the number of messages that must be consumed from the incoming edge $(u, v)$’s buffer each time $v$ executes and the number of messages produced along the outgoing edge $(v, w)$ when $v$ executes. When each module has at most 1 input and output (as in the pipeline case), we use $\text{in}(v)$ and $\text{out}(v)$ as a shorthand for the inputs/outputs consumed by that module. A streaming dag is homogeneous if $\text{in}(u, v) = \text{out}(u, v) = 1$ for every edge $(u, v)$, that is, each module consumes exactly one message from each of its input channels and produces exactly one message to each of its output channels.

In order to execute, or fire a module $v$, the entire state of that module must be loaded into the cache. Moreover, the module may only fire if all of its input buffers contain at least the requisite minimum number of messages.

**Assumptions**

Throughout the paper, we make the following assumptions about the streaming graph. Except for the last one, these assumptions are all either without loss of generality (and made only to simplify the exposition) or necessary to admit any reasonable solution.

We assume that all messages are unit size and that the state of each module is at most $M$. The former assumption is without loss of generality given the arbitrary input and output rates. The latter is necessary to allow a module to be fully loaded into cache when fired.

We assume that the streaming graph contains a single source node $s$ with no incoming edges that produces an infinite stream of input data. Similarly, the graph contains a single sink node $t$ with no outgoing edges that consumes all terminal outputs. This assumption is without loss of generality, as a multisource or multisink dag can be transformed into one with a single source and sink.

We assume that the streaming dag is rate matched, by which we mean that the value $\prod_{(u, v) \in p} (\text{out}(u, v) / \text{in}(u, v))$ is identical for all directed paths $p$ between a fixed pair of vertices. This property is necessary and sufficient to allow the dag to be scheduled without deadlocks with bounded buffers on all channels.

Finally, let $\text{minBuf}(e)$ denote the minimum buffer size required by channel $e$, which can be computed for rate-matched dags using the procedure described in [17]. We assume that for any subgraph $G_i = (V_i, E_i)$ induced by a subset of vertices $V_i \subseteq V$, we have $\sum_{e \in E_i} \text{minBuf}(e) = O(\sum_{v \in V_i} s(v))$. In other words, for any subset of modules, the state size of the modules exceeds the minimum buffer size of channels connecting those modules. This assumption allows us to amortize the cost of reading/writing the buffers for modules against loading the module. For a large class of applications, such as pipelines and homogeneous dags, $\text{minBuf}(e) = \text{in}(e) + \text{out}(e)$, making this condition hold without loss of generality, since a module must regardless load this much of its input and output each time it fires.

**Additional definitions**

Finally, we define some terms and notation used throughout the paper. When there exists a directed path from $u$ to $v$ in the dag, we say that $u$ precedes $v$, denoted by $u < v$. When referring to a particular execution of a module $v$, the progeny of this execution are those messages that may (eventually) appear later in the dag as a result of this execution. In other words, the progeny of a particular execution of $v$ are those messages that may be produced only after $v$ is fired, but before the next time $v$ is fired. We similarly define the progeny of a particular message $m$ along channel $(u, v)$ as the progeny of the execution of $v$ that consumes message $m$.

We use the term “gain,” defined as follows, to describe the rate of amplification of messages along paths through the dag. Gain is only well defined for rate-matched dags.

**Definition 1.** For a vertex $v$, the gain of the module is the number of times $v$ fires for each time the source $s$ fires. That is, for any path $p : s = x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_{|p|} = v$, we define $\text{gain}(v) = \prod_{i=1}^{|p|} (\text{out}(x_{i-1}, x_i) / \text{in}(x_{i-1}, x_i))$.

For an edge $(u, v)$, the gain of the edge is defined as the number of messages produced along the edge for each time the source $s$ fires, denoted by $\text{gain}(u, v) = \text{gain}(u) \cdot \text{out}(u, v)$.

### 3. Scheduling and Execution

In this paper, we concentrate on schedules induced by a “partition” of the dag. Sections 4 and 5 shows that partitioning is a good strategy for scheduling streaming dags of various topologies if the goal is to minimize the number of cache misses. This section describes partition scheduling and the execution model that ensues when streaming applications are scheduled using partition scheduling.

A partition of a streaming dag $G$ is a collection of disjoint subsets of vertices $\{V_i\}$ such that $\bigcup_i V_i = V$. We call each of the sets $V_i$ in the partition a component of the partition. The edges that are internal to a particular component are called internal edges and the edges that cross from one component to another are called cross edges. We are interested in “well-ordered” partitions:

**Definition 2.** Consider a partition $\mathcal{P} = \{V_1, V_2, \ldots, V_k\}$ of a streaming dag $G$. The partition $\mathcal{P}$ is well ordered if the multigraph induced by contracting each component $V_i$ to a single vertex is a dag.

In addition, we want the partition to have the property that each component $V_i$ “fits” in cache. That is, the state of all the modules in the subset, as well as the buffers on internal edges, all fit in cache at the same time. We call
a partition of modules \( \{V_1, V_2, \ldots, V_k\} \) a **c-bounded partition** if the total state size \( \sum_{u \in V} s(u) \) of all modules within each subset is at most \( cM \). The quality of a partition depends on its “bandwidth”:

**Definition 3.** Consider a partition \( P = \{V_1, V_2, \ldots, V_k\} \) of a streaming dag \( V \), and let \( C \) be the set of cross edges. The **bandwidth** of \( P \) is the sum of the gains of the cross edges:

\[
\text{bandwidth}(P) = \sum_{(x,y) \in C} \text{gain}(x,y).
\]

Thus, for homogeneous applications, the bandwidth is simply the total number of crossing edges.

We say that a well-ordered c-partition is an **optimal c-bounded partition** if there is no other well-ordered c-partition of smaller bandwidth. We denote the bandwidth of an optimal c-bounded partition of the graph \( G \) by \( \text{minBW}_c(G) \).

Partition scheduling begins by finding a c-bounded partition with small bandwidth. The scheduling of a partitioned dag is now considered at two levels: the higher level corresponds coarsely to scheduling components, and the lower level corresponds to scheduling modules within each component.

At the lower level, once a component is brought into cache, the modules within it must be scheduled. Each internal edge \( e \) is allocated a buffer with size \( \text{minBuf}(e) \), which is the minimum buffer size required by \( e \) to avoid deadlocks. Since we have

\[
\sum_{e \text{ incident on } v} \text{minBuf}(e) = O(s(v))
\]

for all \( v \), the sum of internal buffers sizes of a component \( V_i \) is \( O(s(V_i)) = O(M) \). Therefore, these internal buffers fit in cache. Since a component is a rate-matched dag, one can always schedule at the lower level without overflowing these buffers [17].

For the higher level, the main idea is that once a component is loaded into the cache, it may be executed many times without incurring any additional cache misses, except when reading from or writing to those cross edges leaving the component. Two properties of the scheduler and partition enable this approach: the addition of large buffers along cross edges, and the fact that the partition is well ordered.\(^4\)

Since the component is executed many times, we can amortize the cost of loading the component’s state against the number of reads/writes to cross edges (the bandwidth of the partition). We say that a component is **schedulable** whenever (1) “enough” inputs exist, and (2) “enough” space remains in the output buffers. In particular, the component is schedulable if it can be executed continuously until it consumes a total of at least \( \Omega(M) \) inputs (or produces at least \( \Omega(M) \) outputs) along cross edges to pay for loading the \( O(M) \) state. The buffers on the cross edges must be large enough to ensure that, at all times, some component is schedulable. By enforcing this condition, we not only guarantee that the scheduling algorithm is deadlock free, but we allow dynamic schedules to be computed easily. The sizes of

\(^4\)If the partition were not well ordered, then a single component could not necessarily be scheduled repeatedly in isolation: some other module might need to be scheduled in between modules of the component.

buffers on cross edges to satisfy this schedulability constraint is different for different stream topologies.

**Scheduling inhomogeneous graphs**

Inhomogeneous graphs may require large buffers on cross edges. How to improve the buffer sizes for inhomogeneous graphs is an interesting problem, but since the size of the cross-edge buffers does not affect our cache bounds, we leave this problem open. To design a schedule for this case, first compute any value \( T \) such that for every edge \( (u,v) \), the value \( T \text{gain}(u,v) \) is integral, is divisible by \( \text{out}(u,v) \) and by \( \text{in}(u,v) \), and is at least \( M \). Under these conditions, all progeny of the \( T \) executions of the source node \( s \) can pass through the entire dag and be consumed by the sink \( t \) without requiring any future inputs at any node in the dag. We thus schedule at a granularity of \( T \) inputs. For each cross edge \( (u,v) \), allocate a buffer with size \( T \text{gain}(u,v) \). For each internal edge, as already stated, allocate a buffer of size \( \text{minBuf}(u,v) \).

At the higher level, execute the components in a topological-sort order, loading each component \( V_i \) exactly once per \( T \) inputs. For the lower level, execute the modules within the component \( V_i \) until all relevant progeny of the \( T \) source-node executions have been consumed and no items are buffered by this component except in the outgoing cross edges. The low-level scheduling can be accomplished by repeatedly choosing any module that can be fired without exceeding output buffer size.

Note that at the high level, once a component \( V_i \) is loaded, it is executed fully before proceeding to a different component. Moreover, it is never executed again until the next batch of \( T \) inputs. In contrast, at the lower level, many different modules may be interleaved with a successive executions of a particular module due to the tighter size restrictions on internal buffers.

**Scheduling homogeneous graphs**

For stream graphs where every module reads and writes exactly one item from its input and output channels, the scheduling is simpler. In this case, setting \( T = M \) and proceeding as above suffices, but the lower level also becomes even more straightforward as \( \text{minBuf}(e) = 1 \) along all edges. In particular, at the higher level, each time the component is loaded, it is fired repeatedly until consuming \( M \) inputs along all incoming cross edges and producing \( M \) outputs along all outgoing cross edges. At the lower level, the modules are topologically sorted and are each fired just once in order; this lower-level schedule repeats \( M \) times.

In this case, one can extend this approach to an asynchronous or parallel dynamic schedule. (Throughout the remainder of this paper, however, our analyses are performed assuming the uniprocessor case.) To schedule components, choose any component(s) with \( M \) data items on all incoming cross edges and empty outgoing cross edges. Then schedule each internal module \( M \) times as above until the incoming buffers are empty and the outgoing buffers are full. The homogeneity of the graph ensures that it is always possible to find a schedulable component.

**Scheduling pipelines**

In the case of **pipelines**, where modules form of a chain (single directed path) but can have nonunit input and output rates, it is again possible to reduce the buffer sizes on cross edges. Each cross edge is assigned a buffer size of \( \Theta(M) \). The
schedule works a little differently, however. A component is schedulable whenever its input buffer is at least half full and its output buffer is at most half full. It is then executed until either the input buffer becomes empty or the output buffer becomes full. Thus, either $\Omega(M)$ inputs are read, or $\Omega(M)$ outputs are written. A continuity argument now suffices to show that some component can always be scheduled. Scan the cross edges in topological-sort order until the first cross edge that is at most half full is encountered. The preceding component has a more-than-half-full-full input buffer by construction, and so it is schedulable. (The sink of the graph is treated as though its output buffer is always empty.) This schedule also readily generalizes to the asynchronous or parallel case as with the homogeneous graphs.

4. STREAMING PIPELINES

This section proves that a good partitioning schedule is nearly optimal for the problem of scheduling a pipeline with arbitrary input and output rates. More precisely, a good partitioning schedule has $O(Q)$ cache misses when run on a cache of size $O(M)$, where $Q$ is the minimum possible number of cache misses of any schedule on a size-$M$ cache. In other words, this schedule is $O(1)$-competitive when given $O(1)$ cache augmentation. Moreover, this schedule can be found in polynomial time. Proving the optimality of the schedule hinges on a strong lower bound, which is a substantial part of the technical content of this section.

The crux of the lower bound is to show that any schedule must “pay” for messages to pass certain edges in the pipeline, essentially showing that the bandwidth of the best partition provides a lower bound on the cost of any schedule. We later present a more general lower bound (Theorem 10 of Section 5), which extends this partitioning lower bound to the more general case of dags. Theorem 10 thus subsumes the lower bound of this section, albeit with different constant factors. Nevertheless, we include both, as the lower bound here is less complicated and provides different intuition. Moreover, although finding the optimal partition for general dags is NP-complete, and we are not aware of any approximation algorithms, the pipeline lower bound immediately suggests a polynomial algorithm for finding an asymptotically optimal schedule for minimizing cache misses on pipelines.

For a pipeline, the well-ordered partitions can be represented compactly as a collection of segments. A $u$-$v$ segment, denoted by $(u,v)$, corresponds to the set of all modules $x$ with $u \leq x \leq v$. We define the gain-minimizing edge $gainMin(u,v)$ to be an edge with minimum gain in the segment $(u,v)$.

The following lemma states that if the first node $u$ in a segment $(u,v)$ is fired enough times, then at least one of two things happens: (1) many messages are buffered within the segment, or (2) some output is produced from $v$. In either case, a certain number of cache misses occur, either due to the buffered messages in the case of (1) or due to the loading of state of the entire segment in the case of (2). One subtle point about the proof is that although the lemma is stated with respect to the number of times the module $u$ is fired, we shall in fact charge against the gain-minimizing edge. Specifically, the eventual lower-bound theorem does not charge for reading the inputs to $u$ — cache misses are assessed only for loading the state of $(u,v)$ or buffering inside $(u,v)$.

**Lemma 1.** Consider a segment $(u,v)$ of a pipeline graph with gain-minimizing edge $(x,y)$. Then module $u$ may be fired at most $2M(gain(u)/gain(x,y))$ times before either

- some progeny of one of these executions of $u$ is output from module $v$, or
- at least $2M$ progeny of these executions of $u$ are stored in buffers between modules $u$ and $v$.

**Proof.** If any progeny is output from the segment, the claim holds trivially. Consider the case where no progeny are output from module $v$. Then the progeny of $u$ must be buffered somewhere between $u$ and $v$. The term $gain(z_1,z_2)/gain(u)$ is by definition the number of messages that (eventually) pass through edge $(z_1,z_2)$ each time $u$ fires. Hence if $u$ fires at least $2M(gain(u)/gain(x,y))$ times, then at least $2M$ messages must be buffered.

If we apply Lemma 1 to a large enough segment, we see that if $u$ is fired enough times, then either $\Omega(M)$ messages must be buffered, or $\Omega(M)$ state must be loaded, thereby incurring $\Omega(M/B)$ cache misses implying a lower bound on the number of cache misses. The following corollary formalizes this observation.

**Corollary 2.** Consider any segment $(u,v)$ of a pipeline graph satisfying the constraint $\sum_{(u,v)} s(z) \geq 2M$, and suppose that $(x,y)$ is the gain-minimizing edge for this segment. Then any (sub)schedule that fires module $u$ at least $2M(gain(u)/gain(x,y))$ times must incur at least $\Omega(M/B)$ cache misses. In other words, $\Omega((1/B)gain(x,y)/gain(u))$ is a lower bound on the amortized cost of firing $u$. This bound holds even if $u$’s inputs and $v$’s outputs are not counted towards cache misses.

**Proof.** Between the time a new message enters the segment and some progeny of that message leaves the segment, each of the modules in the segment must be in cache. Since the total state of these modules is at least $2M$ and can only initially hold at most $M$ of this state, there must be at least $M/B$ cache misses.

Similarly, if $2M$ new messages are buffered, they must be buffered in different memory locations, at most $M$ of which already reside in cache. Thus, there must be at least $M/B$ cache misses.

Applying Lemma 1, we see that the maximum number of times $u$ can be fired before either of these events occurs is $2M(gain(u)/gain(x,y))$, and each event causes $\Omega(M/B)$ cache misses. To calculate the amortized cost, divide the $\Omega(M/B)$ cache misses by the number of executions of $u$.

We conclude by showing that the total bandwidth of the gain-minimizing edges provide a lower bound on the schedule cost. In this theorem, we consider any schedule that fires the sink $t$ of the graph, which is the last node in the pipeline, at least $T \cdot gain(t)$ times, for $T$ a large-enough integer. In other words, we consider schedules that produce a large-enough number of outputs from the pipeline. Note that in order for the sink node to fire at least $T \cdot gain(t)$ times, the source node $s$ must also fire at least $T$ times. The reason to consider outputs here is to avoid any accounting ambiguity due to buffered items throughout the rest of the pipeline.
Consider a pipeline graph in which $S = \{(u_i, v_i)\}$ is any collection of disjoint segments such that each segment has total state at least $2M$. Then any schedule of the graph that fires the sink node at least $T \cdot \text{gain}(t)$ times must incur at least $\Omega((T/B) \sum_{s \in S} \text{gain}(\text{gainMin}(s)))$ cache misses, as long as $T$ is sufficiently large.

**Proof.** If each segment can be considered separately, this theorem follows directly from Corollary 2, since firing at least $T \cdot \text{gain}(t)$ times implies that module $u_i$ must be fired at least $T \cdot \text{gain}(u_i)$ times.

To see that each $(u_i, v_i)$ can be considered separately, notice that the only shared state across different segments is on the cross edges shared between two segments. Since Corollary 2 does not count those edges towards the number of cache misses, there is no double-counting of state/buffer cache misses.

To provide an upper bound, we show constructively that there indeed exists a partitioning schedule that has cost asymptotically matching the lower bound. The following lemma shows that the number of cache misses of a partitioned schedule is related to the bandwidth of a partition. This lemma uses unbounded buffers on cross edges, but this issue is resolved at the end of the section.

**Lemma 4.** Consider any partition $P = \{V_i\}$ of a pipeline graph into segments such that each segment contains at most $M$ total state. Then on a machine with cache size $O(M)$, it is possible to schedule the pipeline such that the sink fires $|T \cdot \text{gain}(t)|$ times using $O((T/B) \cdot \text{bandwidth}(P))$ cache misses in total, as long as $T$ is sufficiently large.

**Proof.** Let $C$ be the set of cross edges induced by partition $P$. Consider the segments $V_1, V_2, \ldots$ in topologically sorted order, where $V_i = (u_i, v_i)$ and the edge $(v_i, u_{i+1}) \in C$. In sorted order, fire $u_i$ a total of $O(T \cdot \text{gain}(u_i))$ times using a local schedule for $V_i$ that has small bounded buffers on internal edges as described in Section 3. Since each $V_i$ fits in $O(M)$ space including the internal buffers, we can repeatedly execute the loaded modules while only paying for reading and writing to external buffers. The total cost is therefore $O(M/B)$ to load $V_i$’s state plus $O((1/B)T(\text{gain}(v_{i-1}, u_i) + \text{gain}(v_i, u_{i+1})))$ to read inputs for $u_i$ and to write outputs from $v_i$. For sufficiently large $T = \Omega(M/\text{gain}(v_{i-1}, u_i))$, the number of items read by $V_i$ is $O(M)$, costing $O(M/B)$ cache misses, which dominates the cost. (The value $T$ must also be large enough that it can be fired $|T \cdot \text{gain}(t)|$ times after firing $s$ at most $O(T)$ times, which occurs when $T$ is at least as large as the gain of every edge in the graph.) Summing across all $V_i$ completes the proof.

The following theorem concludes that our lower bound is tight, modulo constant factors in both the number of cache misses and cache augmentation. Specifically, if $Q$ is the optimal cache cost provided by any schedule (partitioned or not) on a machine with $M$ cache, then there exists a partitioned schedule with cost $O(Q)$ cache cost on a machine with $O(M)$ cache. The upper-bound schedule is exhibited constructively.

**Theorem 5.** There exists a partition $P = \{V_i\}$ of any pipeline graph into segments such that for sufficiently large integer $T$,

- every schedule (not necessarily a partitioned schedule) that fires $t$ at least $T \cdot \text{gain}(t)$ times must cost $\Omega((T/B) \cdot \text{bandwidth}(P))$ cache misses in total for a machine with a size-$M$ cache, and
- there exists a partitioned schedule (based on $P$) that fires $t$ a total of $|T \cdot \text{gain}(t)|$ times and costs $O((T/B) \cdot \text{bandwidth}(P))$ cache misses in total for a machine with a size-$O(M)$ cache.

Moreover, such a partition can be found in polynomial time.

**Proof.** Construct segments $W_i$ as follows. These segments will be used to build $V_i$ later. Start at the beginning of the pipeline, with the current segment initially being $W_1$, and consider all modules in topologically sorted order. Add modules to the current segment $W_i$ until the total state size of $W_i$ exceeds $2M$. If there is more than $2M$ state remaining, finish with $W_i$ and proceed with initially empty $W_{i+1}$ as the current segment. If, on the other hand, less than $2M$ state remains, add all remaining modules to the current segment. By construction, each $W_i$ has total state at least $2M$. Moreover, since all modules have state size at most $M$, each $W_i$ contains at most $5M$ state. (In fact, only the last segment can be this large, as all preceding segments have state between $2M$ and $3M$.)

For each $W_i$ produced through this process, let $(x_i, y_i) = \text{gainMin}(W_i)$ be the gain-minimizing edge for the segment. Then let $C = \bigcup \{(x_i, y_i)\}$ be the set of cross edges. These cross edges induce a partition $P = \{V_i\}$, that is, with $V_0 = (s, x_0), V_i = (y_i, x_{i+1})$ for $1 \leq i < |C|$, and $V_{|C|} = (y_{|C|}, t)$, where $s$ and $t$ are the source and sink of the pipeline, respectively. Since each $W_i$ contains at most $5M$ state, and each $V_i$ spans at most two segments $W_i$ and $W_{i+1}$, it follows that each $V_i$ contains at most $10M$ state. In fact, this bound may be tightened to $8M$, since only the last segment can be so large. Therefore, for $c = 8$, this partition has the property that each segment is of size at most $cM$.

To achieve the lower bound, apply Theorem 3 with $S = \{W_i\}$. To achieve the upper bound, apply Lemma 4 with partition $\{V_i\}$.

**Corollary 6.** For a given pipeline graph and a sufficiently large number of outputs to produce, one can construct a schedule that incurs at most $O(1)$ times as many cache misses as the optimal schedule, as long as the schedule is allowed to use a constant factor more cache than the optimal schedule.

Although the partition described in Theorem 5 provides an asymptotically optimal upper bound on the number of cache misses (modulo memory augmentation), this partition is not the minimum bandwidth $c$-bounded partition. As it turns out, one can find the minimum bandwidth $c$-bounded partition for pipelines using a simple dynamic program. This (minimum bandwidth) partition provides no more cache misses than the partition described in Theorem 5, but not asymptotically fewer. Moreover, this optimal partition does not guarantee optimal cache performance. It still only guarantees asymptotically optimal cache performance.

**Producing an optimal dynamic schedule**

Lemma 4 (and Theorem 5) give schedules for the pipeline that rely on the number of outputs to produce. In particular, as specified, each segment $V_i$ in the partition is considered in order and fired repeatedly until enough outputs are produced.
produced. For the pipeline case, these schedules can be easily transformed into dynamic schedules, where the number \( T \cdot \text{gain}(T) \) of times the sink must be fired is not specified \textit{a priori}. Specifically, Lemma 4 requires only that each \( V_i \) consume \( \Omega(M) \) inputs or produce \( \Omega(M) \) outputs each time it is loaded. Thus, the segments can be scheduled as described in Section 3 in order to get the same bounds.

5. STREAMING DAGS

This section considers the case when the stream graph is a general directed acyclic graph (dag). A natural generalization of the pipeline schedule is the following: partition the dag into regions that fit in cache while minimizing the total gain of edges crossing the partitions. The key to showing that this type of strategy yields an asymptotically optimal cache bound is to provide a lower bound on the cache complexity of any schedule, showing that every schedule must incur at least as many misses (asymptotically) as some partition-based schedule. We first prove this claim for the homogeneous dataflow case (all gains are 1), and then we generalize to nonunit gains later in the section.

Proving a lower bound for the cache complexity of scheduling a streaming dag is significantly more complicated than for a pipeline. The following theorem shows that the bandwidth of an optimal \( c \)-bounded partition does indeed provide such a lower bound for the case of homogeneous dataflow. Recall that the term \( \text{minBW}_G(G) \) used in the statement of the next theorem denotes the minimum possible bandwidth of a well-ordered partition into components such that each component has total state size at most \( 3M \). Without loss of generality, we assume a single input node (source) and a single output node (sink) to simplify exposition of the proof.

**Theorem 7.** Any schedule for a homogeneous dataflow dag that fires the sink node at least \( T \) times must incur at least \( \Omega(T/B) \cdot \text{minBW}_G(G) \) cache misses, where \( B \) is the block size.

**Proof.** The main idea of this proof is to look at (the progeny of) a single message passing through the entire dag from the source to the sink. We will argue that any schedule of this dag must pay at least \( \Omega(1/B) \) block transfers for each edge crossing some \( 3 \)-bounded partition (in particular, an optimal \( 3 \)-bounded partition). Since modules may be executed more than once each time they are loaded, we must be careful about the way we count the block transfers incurred by loading the state of a module. It is not obvious that we can analyze a single message in isolation, although the proof will essentially do just this.

Consider any schedule \( \pi \) of modules that fires the sink \( T \) times. A schedule \( \pi \) is a list of module executions \( \pi = u_1, u_2, \ldots, u_m \) for some \( m \). The same module can occur many times in the list (i.e., \( u_i = u_j \) for some \( j \neq i \)). In particular, with all gains equal to 1, every module occurs at least \( T \) times in any schedule that produces \( T \) outputs. This interpretation of a schedule ignores whether messages are stored in intermediate buffers or not, but the proof charges for buffering where necessary.

Starting at the beginning of the schedule \( \pi \), partition the schedule into contiguous subschedules \( \pi = \pi_1, \ldots, \pi_r \), each containing between \( 2M \) and \( 3M \) distinct state (except for the last subschedule which may be smaller). Such a partition is possible because each module has size at most \( M \).

We first claim that beginning each subschedule \( \pi_i \) with an empty cache does not increase the overall cost of the schedule \( \pi \) by more than a constant factor. This fact follows because each \( \pi_i \) contains at least \( 2M \) state, and thus it must incur \( \Omega(M/B) \) cache misses regardless of what is in the cache at the beginning of that subschedule. We can therefore afford to pay \( O(M/B) \) to load any arbitrary initial cache state at the beginning of each subschedule without asymptotically increasing the cost.

Now look at any subschedule \( \pi_i \). Let \( K_i \) be the total number of outputs produced by modules in \( \pi_i \), that are not consumed as input by another module within \( \pi_i \). Each of these outputs must be written to cache or memory, taking a total of \( K_i \) space. Since we can assume that the cache is flushed at the end of each subschedule, the cost of executing \( \pi_i \) becomes \( \Omega(K_i/B) \).

Showing that \( \sum_i K_i \geq T \cdot \text{minBW}_G(G) \) will complete the proof. We do so by arguing that the progeny of a single source-node firing must contribute at least \( \text{minBW}_G(G) \) edges to \( \sum_i K_i \) (and hence the total for \( T \) firings is at least \( T \) times more). Proving this fact has two components. We first show that \( \pi \) can be reduced to a schedule \( \pi' \), tracking only the progeny of one firing, such that every edge crossing a subschedule boundary in \( \pi' \) also crosses the same boundary in \( \pi \). Then we argue that \( \pi' \) is a well-ordered, \( 3 \)-bounded partition. If both of these claims hold, then the number of edges that a single firing contributes to \( \sum_i K_i \) is at least the number of edges crossing some well-ordered \( 3 \)-bounded partition, and hence at least \( \text{minBW}_G(G) \).

To reduce \( \pi \) to \( \pi' \), consider a single source-node execution and all of its progeny throughout the dag, and consider the schedule \( \pi' = \pi_1', \ldots, \pi_k' \) induced by removing all but the corresponding modules from \( \pi \). This schedule corresponds to every module firing exactly once within \( \pi' \). All edges in \( \pi' \) also occur in \( \pi \). Thus, each edge crossing subschedule boundaries in \( \pi' \) also crosses subschedule boundaries in \( \pi \) and contribute to some \( K_i \).

To show that \( \pi' \) is a well-ordered \( 3 \)-bounded partition, observe that since each \( \pi_i \) (and hence \( \pi'_i \)) contains at most \( 3M \) state, \( \pi'_i \) corresponds to a \( 3 \)-bounded partition. Moreover, since every module is included just once in the schedule and the schedule obeys precedence constraints, it follows that modules \( \pi'_1, \ldots, \pi'_k \) have no edges pointing to earlier modules. Hence each contracted \( \pi'_i \) has no edges pointing to the earlier components, and the contracted graph is a dag, meaning that it is well-ordered.

We now prove the upper bound, that is, if a good partition exists, then it is possible to schedule the pipeline with asymptotically optimal cache performance on a machine with \( O(1) \) cache-size augmentation. In order to schedule a dag using partitioning, we need the partition to be \textit{degree-limited}, meaning each component of the partition has degree \( O(M/B) \), for the following reason. When a component is loaded into cache and executed, the cache should also be large enough to accommodate at least one block from the buffers on cross edges that are incident on this component. For a wide class of graphs, all \( O(1) \)-bounded partitions are degree limited. For example, if every module \( u \) contains at least \( \Omega(Bd_u) \) state, where \( d_u \) is its degree, then all \( O(1) \)-bounded partitions are degree limited. The following lemma shows that the bandwidth of the partition determines the number of cache misses by a partitioned schedule.
Lemma 8. Consider any degree-limited \(O(1)\)-bounded partition \(P = \{V_i\}\) of a homogeneous dataflow dag. Then on a machine with \(O(M)\) cache size and for sufficiently large \(T\), there exists a schedule of the dag that fires the sink node \(T\) times with \(O((T/B) \cdot \text{bandwidth}(P))\) cache misses in total.

Proof. As mentioned in Section 3, we add (large) buffers of size \(\Theta(M)\) to each cross edge and then load each component into cache one at a time, executing each \(O(M)\) times once loaded. Since each \(V_i\) fits in \(O(M)\) space including the internal buffers, we can repeatedly execute the loaded modules while only paying for reading all the inputs from and writing all the outputs to external buffers. The total cost is then \(O(M/B)\) to load \(V_i\)'s state plus the cost of reading the inputs from the incoming edges and writing outputs to outgoing edges. For large enough \(T\), i.e., \(T > M\), the number of inputs read is \(\Theta(M)\) from each input edge (similarly for outputs) each time a component is scheduled. Since the degree of each partition is \(O(M/B)\), we can afford to keep at least 1 block for each external buffer in cache at the same time, and thus reading/writing these external messages has cost \(\Theta(1/B)\) times the number of message read or written. Therefore, reading the inputs and writing the outputs costs \(\Theta(M/B)\) per cross edge, which dominates the cost of reading the state. Summing across all \(V_i\) and repeating \(T/M\) times completes the proof.

The following corollary (which follows directly from Theorem 7 and Lemma 8) says that a partitioned schedule is asymptotically optimal given \(O(1)\)-memory augmentation. More generally, since finding the optimal well-ordered partition is NP-complete [8], we show that if we have an \(O(\alpha)\)-approximation for the partitioning problem, then we have an \(O(\alpha)\)-competitive schedule.

Corollary 9. Let \(P_{opt}\) be an optimal 3-bounded partition of a homogeneous streaming dag \(G\), and let \(P\) be any 3-bounded degree-limited partition of \(G\). Suppose that we have \(\text{bandwidth}(P) \leq \alpha \cdot \text{bandwidth}(P_{opt})\) for some \(\alpha \geq 1\). Then for any sufficiently large number of outputs produced, the partition schedule using \(P\) incurs at most \(O(\alpha)\) times as many cache misses as the optimal schedule, as long as the partitioning schedule is allowed to use a constant times larger cache than the optimal schedule uses.

Notes on the upper bound

Corollary 9 is most useful for those dags with the following property: there exists a degree-limited 3-bounded partition with similar bandwidth to the optimal (not necessarily degree-limited) 3-bounded partition. When this property does not hold, the cost of the upper-bound partition may be worse by a factor of \(B\) (every read and write to cross-edge buffers may cause a cache miss), implying only that our schedule is \(O(\alpha B)\)-competitive. The cost of a naive schedule, however, may be much worse, since only the messages moving along cross edges are charged. Moreover, we have not discussed how to find such a partition. Like most partitioning problems, finding an optimal well-formed 3-bounded partition is NP-hard. Since the partitioning occurs at compile time, however, and the streaming application is intended to be longer running, it may be reasonable to use an exponential-time algorithm for constructing a good partition. In any event, we have reduced the problem of scheduling to that of partitioning.

6. Related Work

Most of the work on scheduling streaming applications aims to maximize throughput and/or minimize latency [2, 6, 18], minimize buffer sizes [4, 23, 28], or to optimize both while avoiding deadlocks [13, 19]. A few papers address other issues, such as power consumption [29]. None of these relates directly to our results.

Heuristic cache-aware scheduling of streaming programs on both single processors and multiprocessors has been studied by several research groups [15, 21, 25], but the proposed heuristics are all evaluated empirically. To the best of our
knowledge, no previous work provides any theoretical guarantees comparable to ours. Kohli [15] proposes a greedy scheduling heuristic for streaming graphs that have a unique topological ordering (this class is a minor generalization of pipelines). The heuristic makes local decisions as to whether to continue to execute one module or to move on to its successor in the topological ordering based on an estimate of the number of cache misses that either decision will generate. Since this heuristic makes only local decisions, it does not provide an asymptotically optimal number of cache misses. Another difference from our work is that Kohli considers data-cache and instruction-cache misses separately, while we assume either that instruction cache is not an important bottleneck or that instructions are part of the state of the module. Sermulins et al. [25] also try to heuristically optimize for both the data cache and the instruction cache. Their schedule-optimization method starts from some given steady-state schedule that maintains bounded buffer sizes. Their optimizer takes this schedule and produces a new schedule which replaces each module invocation in the steady-state schedule by $s$ back-to-back invocations. This scaling leads to state reuse, but it may cause spilling of buffers into main memory. Their method computes the largest $s$ that avoids catastrophic spills. Since this method generates a small range of schedules, all of which are derived from the given steady-state schedule, it is suboptimal in many cases. Sermulins et al. also propose a module-fusion heuristic that can also reduce cache misses. This heuristic can be viewed as a special case of our partitioning method. Moonen et al. [21] also scale a given schedule in the same way, but in a more general setup that includes multiple processors and computational graphs that allow module to change their gains in a cyclic fashion. They do not provide any theoretical guarantees (it is unlikely that any can be proved for the method), but they do show convincing empirical results, demonstrating a cache-miss reduction of over a factor of 4 on a real-world application. Taken together, the empirical results strongly support our fundamental claim that effective scheduling can dramatically improve the performance of streaming applications.

Graph partitioning has also been used to improve cache efficiency under computational models other than streaming models. Examples include mesh processing for visualization [26], which provides theoretical guarantees, and repeated sparse matrix-vector multiplication [30], which employs an empirical heuristic methodology.

7. CONCLUSIONS

We have shown that in many cases the problem of cache-efficient scheduling of streaming applications can be reduced to the problem of partitioning the computational graph into components that have a small boundary and have a small enough state to fit within the cache. On some classes of graphs, such as pipelines, solving the resulting partitioning problem is computationally easy. For more complex applications where the streaming graph is a synchronous dag, we can still show that the scheduling problem reduces to a partitioning problem. The problem of finding the optimal well-ordered partition for general daggs is NP-complete [8], however. This situation is common to many mapping and scheduling problems in high-performance computing.

There are a few ways to address this difficulty. One practical approach is to use an exact integer-programming graph partitioner when the dag is relatively small. This strategy has proved effective in scheduling streaming computations on distributed memory-limited systems [22], for example. Another approach is to use a heuristic graph partitioner (see, for example, [10, 14]). Heuristic strategies are widely used to map and schedule large-scale parallel applications. Alternatively, since our results are approximation preserving, one can try to find a provably good approximation algorithm for the partitioning problem. We plan to address this issue in future work.

Another direction for future research is to study the cache-efficient scheduling of streaming computations on multiprocessors. If the number of cache misses is the only criterion, then the optimal uniprocessor schedule is trivially the optimal multiprocessor schedule. When considering multiprocessors, however, we must consider both load balancing and the number cache misses simultaneously.

Our work has also delineated the tougher theoretical problems in scheduling streaming applications. These include, not surprisingly, feedback (cycles in the graph) and modules whose output rates are not simple functions of their input rates (such as modules that sift through data and produce an output when they find something interesting, modules that make routing decisions, etc.). These computational structures also raise theoretical and practical difficulties unrelated to caching, such as the possibility of deadlocks due to insufficient buffer space. These issues, some of which are clearly online-scheduling problems, represent good opportunities for future research.

Acknowledgments

We would like to thank Fanny Dufossé of ENS Lyon for helpful discussions. This research was supported in part by NSF grants CCF-1150036, CNS-1017058, and CCF-0937860, by grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by grant 2010231 from the United-States-Israel Binational Science Foundation.

8. REFERENCES


