Cache-Conscious Scheduling of Streaming Pipelines on Parallel Machines with Private Caches

Kunal Agrawal and Jordyn Maglalang
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
Email: kunal@wustl.edu, jordyn.maglalang@wustl.edu

Jeremy T. Fineman
Department of Computer Science
Georgetown University
Washington, District of Columbia, USA
Email: jfineman@cs.georgetown.edu

***Abstract*—This paper studies the problem of scheduling a streaming pipeline on a multicore machine with private caches to maximize throughput. The theoretical contribution includes lower and upper bounds in the parallel external-memory model. We show that a simple greedy scheduling strategy is asymptotically optimal with a constant-factor memory augmentation. More specifically, we show that if our strategy has a running time of Q cache misses on a machine with size-M caches, then every "static" scheduling policy must have time at least that of $\Omega(Q)$ cache misses on a machine with size-$M/6$ caches. Our experimental study considers the question of whether scheduling based on cache effects is more important than scheduling based on only the number of computation steps. Using synthetic pipelines with a range of parameters, we compare our cache-based partitioning against several other static schedulers that load-balance computation. In most cases, the cache-based partitioning indeed beats the other schedulers, but there are some cases that go the other way. We conclude that considering cache effects is a good idea, but other features of the streaming pipeline are also important.**

I. Introduction

Streaming is an effective paradigm for parallelizing complex computations on large datasets. A streaming application can be described by a directed graph where nodes correspond to ***computation modules***, and edges represent directed FIFO channels between modules. The modules send data in the form of ***messages*** (also called ***tokens*** or ***items***) via these channels. In this paper, we only consider ***pipeline*** topologies where modules are connected in a linear chain via channels. The streaming paradigm has been applied to diverse application domains such as media [1], signal processing [2], computational science [3] and data mining [4]. Several languages explicitly support streaming semantics, including Brook [5], StreamC/KernelC [6], and StreamIt [7].

There has been extensive research on how to map and schedule streaming computations to parallel machines and how to schedule them. Much of the existing work concerns either maximizing ***throughput***, defined as the average amount of data processed per unit time, or minimizing ***latency***, defined as the maximum time to fully process a single input message. The goal of this paper is to optimize throughput on a shared-memory machine when taking the cache effects into account, a feature often ignored in prior work. Streaming applications exhibit two kinds of cache misses that can be controlled using intelligent scheduling. First, modules access their state when they fire, it is advantageous to fire the same module many times once its state has been loaded. Second, it is advantageous to execute consecutive modules in quick succession in order to keep the data produced by module by the first module on its output channel in cache until the second module consumes it. Since these heuristics are contradictory, we must balance concerns intelligently to design a good scheduling algorithm. We consider the problem of optimizing the throughput of a streaming pipeline on a shared-memory machine consisting of P cores, where each core has a single private cache. This work consists of both theory (Sections III and IV) and practice (Section V).

Cache efficiency is an important determinant in the performance of algorithms, and there has been extensive theoretical and practical work in designing cache-efficient algorithms for both sequential [8], [9] and parallel [10] machines. On a shared-memory machine, the communication cost roughly corresponds to a subset of the cache misses, but additional cache misses occur locally on each processor and are not reflected by communication cost. We are not aware of much prior work, particularly with a theoretical foundation, that considers the impact of cache effects on throughput in streaming applications. One example is from Agrawal et al. [11], who focus on designing cache-efficient scheduling of streaming applications on single-processor machines. Specifically, they show that for both pipelines and general acyclic graphs, a simple partitioning strategy generates a schedule that is asymptotically optimal for a single-level cache. In this paper, we consider the question of cache-conscious scheduling on parallel machines.

Our theoretical work adopts the parallel external-memory model [12], where a ***parallel I/O*** consists of simultaneous cache misses by up to P processors, and time is counted by the number of parallel I/Os. The model thus ignores the cost of computation — our hypothesis is that the cost of memory access often dominates, and hence this model is

fairly accurate. Note that a scheduler may sometimes leave a processor idle due to other scheduling constraints, so a single parallel I/O may consist of anything between 1 and P cache misses occurring in parallel. It is thus not enough to count just the total number of cache misses — one must also consider any sequential ordering that arises.

Most existing parallel or distributed schedulers for streaming pipelines are ***static***, meaning that each module is assigned to a specific processor and never migrates to another processor. Static schedules are easier to describe, they can often be computed *a priori* and repeated periodically, and they perform well in practice. Section III provides lower bounds on the runtime, or number of parallel I/Os, for static schedulers, as well as a slightly weaker lower bound for general schedulers. Section IV describes a simple static scheduler, called *seg_cache*, which is a greedy algorithm that partitions the pipeline into P contiguous pieces, one for each processor. The scheduler is a straightforward adaptation of Agrawal et al.'s [11] single-processor scheduler — a contribution of the present paper is a new upper bound on the parallel I/O complexity of *seg_cache*. We also show that *seg_cache* is asymptotically optimal with respect to static schedulers and a constant factor memory augmentation. That is, if *seg_cache* has Q parallel I/Os on a machine with size-M caches, then every static schedule on a machine with size-$M/6$ cache must incur at least $\Omega(Q)$ parallel I/Os.

We also conduct an experimental study (Section V) to test the efficacy of this scheduling strategy on a shared memory machine using randomly generated synthetic pipelines. We compare *seg_cache* against schedulers that try to load-balance the computation time across processors. We find that for a large variety of configurations, the cache-based *seg_cache* beats the computation-balancing policies. We also created a partitioning heuristic that takes both cache misses and computation into consideration while mapping streaming pipelines. While we don't prove that this heuristic is theoretically optimal, our experiments indicate that it often outperforms both cache-based segmentation and runtime load-balancing policies.

II. Model and Definitions

This section describes the analytic and streaming models. We also state various assumptions and definitions about pipelines used throughout the paper.

PEM model : Our theoretical analysis is with respect to the parallel external memory (PEM) model [12], which is an extension of the external memory model [8] and similar to many other private-cache models in the literature (e.g., in [9]). The PEM model is a computational model consisting of P processors, each with a private cache of size M, and a global shared memory. The caches and shared memory are organized into ***blocks*** of B consecutive addresses. A processor can only read or write data in its own cache; when accessing data not in cache, a ***cache miss*** or ***I/O*** occurs, whereby the block must be loaded from shared memory to the processor's cache. The model allows a cache to store any M/B blocks simultaneously (i.e., caches are fully associative). All communication between processors occurs through the shared memory in the form of I/Os.

Processors may perform I/Os concurrently, which is called a ***parallel I/O***. The complexity measure is the number of parallel I/Os. Interpreted as time, accessing data in cache is free, but each I/O takes unit time. Thus in a single timestep, each processor may load up to 1 block or B elements, for a total of P blocks or PB elements. A "linear" I/O bound for, e.g., touching n elements in an array is $O(n/(PB))$.

Variants of the model specify when the same data may be resident in multiple caches, but these are not important for the present paper. We require exclusivity with respect to modules only — the same module may not be resident in two caches simultaneously. Otherwise, our lower bound applies to all model variants, and our upper bound does not have any data simultaneously loaded on any caches.

Streaming model : A streaming pipeline consists of a sequence of n ***computational modules***, and each module u has exactly one ***incoming channel*** (from the previous module in the sequence) and one ***outgoing channel*** (to the subsequent module in the sequence). The modules send data in the form of ***messages*** to each other via these channels. Channels may have ***buffers*** (implementing FIFO queues) to store messages that have not yet been consumed by the receiving module. We say that module u precedes module v, denoted by $u \prec v$, if u is before v in the sequence. We assume that the incoming channel into ***source module*** s (the first module) streams an infinite amount of data into the pipeline and the outgoing channel from ***sink module*** t (the last module) streams it out.

Each module u has an associated state; we denote the size of this ***state*** by $s(u)$. In order to execute, or ***fire*** a module u on a processor, the entire state of that module must be loaded into that processor's cache. When the module fires, it consumes $in(u)$ data items from its incoming channel, performs some computation, and then produces $out(u)$ data items on its outgoing channel, where $in(u)$ and $out(u)$ are static parameters of the module. A module is ***ready*** to fire its input buffer contains at least $in(u)$ messages.

Assumptions and Definitions : Throughout the paper, we make the several assumptions about the streaming pipeline — these assumptions are either necessary to admit any reasonable solution or are without loss of generality (made only to simplify the exposition). We assume that all messages are unit size and that the state of size 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 resident in cache when fired. In addition, we assume that the state size of each module also counts the minimum buffer required on its incoming and outgoing channels.

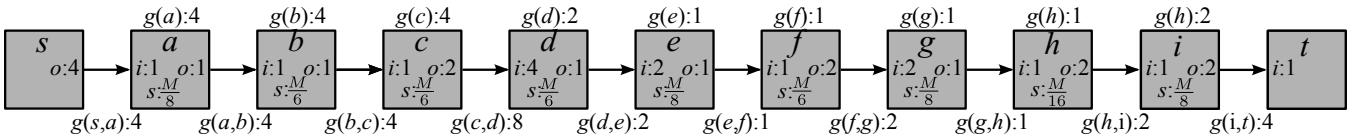


Figure 1: An example pipeline with annotated state size, input and output rates and gains for both modules and edges.

Finally, we define some terms and notation used throughout the paper. We use the term ***gain*** to describe the rate of amplification of messages along the pipeline. In particular, The gain of a module is the number of times that module fires, on average, each time the source module consumes an input item. Therefore, the gain of the module is $gain(w) = \prod_{u \prec w}(out(u)/\ in(u)) \times \frac{1}{in(w)}$. The gain of edge (v, w) is the number of items produced on that edge, on average, each time an item is consumed by the source module, and is given by $gain(v, w) = gain(v) \times out(v)$. Figure 1 represents an example pipeline with module and edge gains computed.

We call a set of consecutive modules a ***segment***, and we denote the segment comprising modules between u and v (inclusive) by $\langle u, v\rangle$. In each segment s, we call the edge with minimum gain a ***gain-minimizing edge*** (if there is more than one, choose arbitrarily), denoted by $gainMin(s)$.

III. Lower Bounds

This section gives lower bounds on cache misses when scheduling a streaming pipeline on multiple processors, which we leverage later to prove that a partitioned schedule is optimal. The two bounds are analogous to lower bounds for other load-balancing problems. Specifically, we first lower-bound the total number of misses, which implies a lower bound on parallel I/Os (or time) since $\leq P$ occur concurrently. Our second bound addresses the case where parallelism P is not achievable due to local imbalance, i.e., if some small part of the pipeline dominates the running time. Combining the two gives a lower bound on running time that matches the upper bound achievable by a partitioned scheduler. Unfortunately, and surprisingly to us, the second lower bound is restricted to the case of a static scheduler and does not hold in general. We discuss this limitation at the end of the section, highlighting what makes proving bounds on (non-static) parallel streaming schedulers difficult.

A. Lower Bound on Total Misses

We first bound the total number of cache misses — the proof is inspired by the approach of Agrawal et al. [11], but with some changes to cope with the multiprocessor setting. The basic intuition is that any schedule (static or not) must "pay" for the messages crossing certain edges in the pipeline. The first lemma bounds the cache-miss cost for a single processor with respect to a single edge.

Lemma 1: In the pipeline under consideration, let $s = \langle u, v\rangle$ be any segment with total state size at least $2M$, and let $e = gainMin(s)$ be its gain minimizing edge. Any subschedule that fires module v at least $2M\ gain(v)/\ gain(e)$ times on a particular processor p incurs at least M/B cache misses on p. Moreover, these M/B cache misses are all due to either loading state from modules or messages on edges within the segment $\langle u, v\rangle$.

Proof: The proof consists of two cases.

Case 1: Suppose processor p loads the entire segment $\langle u, v\rangle$ during the subschedule. At most M of that state can already be resident in p's cache at the start of the subschedule, so p must incur at least M/B cache misses to complete the load.

Case 2: Suppose that some module in $\langle u, v\rangle$ is not fired by p during the subschedule. We define a message m to be a ***crossing ancestor*** if m is consumed by a module running on processor p during the subschedule, but m is not generated by a module on processor p during the subschedule. (If m was generated on processor p but *before* the subschedule began, it is still considered a crossing ancestor.) Since p does not fire the entire segment during the subschedule, all inputs to v during the subschedule are the progeny of some crossing ancestor. The number of crossing ancestors is minimized if they all occur at the gain minimizing edge e, and hence there must be at least $2M$ crossing ancestors in order to fire v a total of $2M\ gain(v)/\ gain(e)$ times. By definition, crossing ancestors are read by p, so each crossing ancestor must either already be resident in p's cache before the subschedule, or it must be loaded into p's cache during the subschedule. Since at most M crossing ancestors can be resident at the start of the subschedule, the remaining M crossing ancestors incur at least M/B cache misses. ■

The following corollary combines Lemma 1 across all processors. The nuance here that necessitates the new lower bound, as opposed to applying Agrawal et al.'s bound [11] as a black box, is that P processors have PM cache in total instead of the M cache for the uniprocessor case.

Corollary 2: Consider a streaming pipeline. Let $s = \langle u, v\rangle$ be any segment with total size at least $2M$, and let $e = gainMin(s)$ be its gain-minimizing edge. Any subschedule that fires v at least $6PM\ gain(v)/\ gain(e)$ times in total across P processors must incur $\Omega(PM/B)$ cache misses. In other words, $\Omega((1/B)\ gain(e)/\ gain(v))$ is a lower bound on the amortized cost of firing v.

Proof: From Lemma 1, if a particular processor p fires v a total of $f_p\ \lceil 2M\ gain(v)/\ gain(e)\rceil$ times, then it incurs at least $\lfloor f_p\rfloor\ M/B$ cache misses. It remains to prove that $\sum_p \lfloor f_p\rfloor \geq P$, and hence the total number of cache miss is $\sum_p \lfloor f_p\rfloor\ M/B \geq PM/B$.

Since e is the gain-minimizing edge and $in(v) \leq M$, we have $gain(e) \leq gain(v) \cdot in(v)$ and thus $gain(v)/\ gain(e) \geq 1/\ in(v) \geq 1/M$. It follows that $2M\ gain(v)/\ gain(e) \geq 2$,

and hence $\lceil 2M\ gain(v)/\ gain(e) \rceil \leq 3M\ gain(v)/\ gain(e)$. Since there are at least $6PM\ gain(v)/\ gain(e)$ firings in total, we have $\sum_p f_p \geq 2P$ and hence $\sum_p \lfloor f_p \rfloor \geq P$. ■

The following theorem combines the preceding corollary across the entire pipeline. Note that the theorem identifies which edges must be paid for with respect to an arbitrary segmentation of the pipeline (defined as "bandwidth" in [11]). Nevertheless, the bound holds for *any* schedule, even a non-partitioned and non-static schedule.

Theorem 3: Consider a pipeline graph in which $S = \{\langle u_i, v_i \rangle\}$ is any collection of disjoint segments such that each segment has total size at least $2M$. Then for sufficiently large T, any parallel schedule of the pipeline that fires the sink node t at least $T \cdot gain(t)$ times must incur at least $\Omega((T/B) \sum_{s \in S} gain(gainMin(s)))$ cache misses in total.

Proof: Observe that if t fires $T\,gain(t)$ times, then v_i fires $T\,gain(v_i)$ times. Thus if $T \geq 6PM/\ gain(gainMin(s))$ for $s \in S$, then we can apply Corollary 2 to s to get a cache-miss bound of $\Omega((T/B)\ gain(gainMin(s)))$. In addition, each application of Lemma 1 and Corollary 2 only counts misses for messages/state within each segment. Therefore, there is no double counting of cache misses. ■

B. Lower Bound on Time

Theorem 3 implies that the number of parallel I/Os is at least $\Omega((T/(PB)) \sum_{s \in S} gain(gainMin(s))$, since at most P misses occur in any parallel I/O. This provides a lower bound on the running time of any schedule. We now argue that for *static schedules*, the gain-minimizing edge with the largest gain also provides a lower bound.

Theorem 4: Consider a pipeline graph in which $S = \{\langle u_i, v_i \rangle\}$ is any collection of disjoint segments such that each segment has total size at least $2M$, and let t be the sink node. After t is fired at least $T \cdot gain(t)$ times for sufficiently large T, the running time is,

- $\Omega((T/(PB)) \sum_{s \in S} gain(gainMin(s)))$ parallel I/Os for any schedule, and
- $\Omega((T/B) \max_{s \in S} gain(gainMin(s)))$ parallel I/Os for any *static* schedule.

Proof: The first statement follows from Theorem 3. For the second statement, consider segment $s' = \langle u, v \rangle$ which has the largest gain-minimizing edge. Lemma 1 shows that for this segment, if the module v stays on one processor p, then p must incur an amortized $\Omega((1/B)\ gain(gainMin(s'))/\ gain(v))$ cache misses each time v is fired, each of which must be part of a different parallel I/O. ■

Somewhat surprisingly, the second lower bound deriving from the maximum *does not* hold for general schedulers, only for static ones. We demonstrate by example: Suppose that the pipeline consists of $2kM/B$ modules, each of size B, with all gains equal to 1. Let $P = kM^2/B$ be the number of processors. For conciseness only, the following description assumes each processor has one extra block, i.e., $M+B$ cache size. Consider the following schedule: For each module u, assign a M/B processors, denoted P_u, to module u. Place cyclic buffers of size M^2 on each edge. Warm-up the schedule to get $M^2/2$ messages in each buffer, which is invariant between rounds in the remaining schedule. The schedule proceeds in rounds, in parallel on each module, consisting of a load step followed by an execute step.

Load step for u: foreach processor from P_u in parallel, load a distinct $M/(2B)$ full blocks from u's input buffer and also preload the corresponding $M/(2B)$ empty blocks from u's output buffer. These loads occur in parallel, so the total number of parallel I/Os is M/B.

Execute step for u: foreach processor from P_u but now sequentially (since u can not be fired on multiple processors at once), load u's state and fire it $M/2$ times. In the execute step, loading u incurs one cache miss on each processor for M/B parallel I/Os total, but the input and output buffers are already in cache, so firing u incurs no other I/Os.

Putting it together, a single round takes $2M/B$ parallel I/Os but fires each module $M^2/(2B)$ times, for a total of $O(1/M)$ per firing. Thus if the sink fires T times, then the total time is $O(T/M)$.

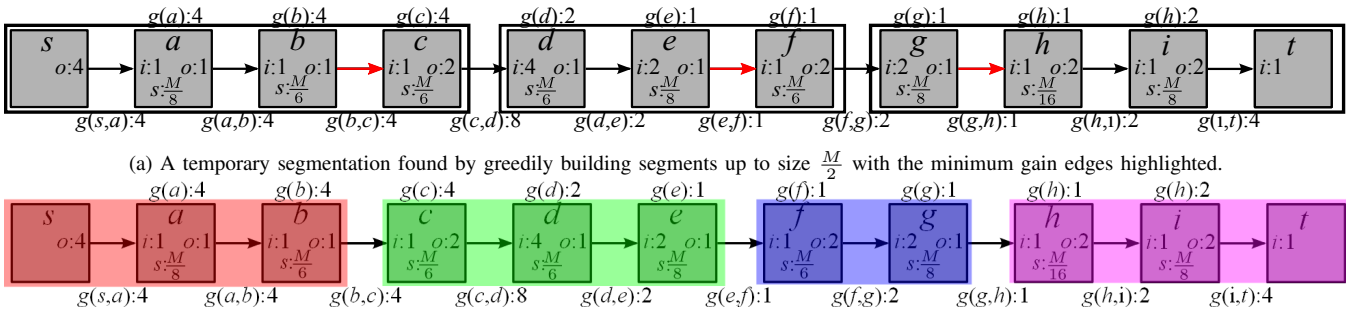
This upper bound gives an example where the maximum-based lower bound (second condition in Theorem 4) does not hold. That bound would say that the total time is $\Omega(T/B)$ which is much larger than $O(T/M)$. This discrepancy arises from the fact that the example schedule moves modules across processors. The first condition of Theorem 4 still holds, since it only states that the lower bound on time is $\Omega(Tk/(PB)) = \Omega(T/M^2)$. Note that, setting $k = \Theta(B)$, this example only achieves a runtime of $O(Tk/(\sqrt{PB})$ parallel I/Os, not $O(Tk/(PB))$. There is a tradeoff here; the example is able to beat the $\Omega(T/B)$ lower bound by increasing the total number of misses by a factor of M, which in turn increases running time arising from the total-work bound. Therefore, we have shown that a non-static schedule can do better than a static schedule, however, at the cost of increasing the total number of cache misses.

IV. Upper Bound on Scheduling Pipelines on Multiple Processors

This section describes our cache-based partitioning algorithm, called *seg_cache*, for scheduling pipelines on multiple processors. For correctness, this algorithm assumes that the maximum state size of any module is at most $M/6$.

A. The seg_cache Algorithm

The algorithm produces a partitioning by first dividing the pipeline into temporary segments $S = \{s_1, s_2, \ldots, s_k\}$, where each segment $s_i = \langle u_i, v_i \rangle$ for $i < k$ has total state between $M/3$ and $M/2$ — the last segment s_k may be smaller. This temporary segmentation can be selected greedily: Initially create one segment s_1 as the current



(a) A temporary segmentation found by greedily building segments up to size $\frac{M}{2}$ with the minimum gain edges highlighted.

(b) The final segmentation using the cross edges found from the temporary segmentation.

Figure 2: The final segmentation using *seg_cache* derived by cutting the minimum gain edge of each temporary segment. Note that each segment has size at most M.

segment. Iterate over the modules in order. Add the current module to the current segment. If the total state of the current segment s_i exceeds $M/3$, create a new segment s_{i+1} and set that to be the current segment. Under the assumption that modules have state at most $M/6$, this process produces segments with total state at most $M/3 + M/6 = M/2$. Figure 2a shows this temporary segmentation for the pipeline in Figure 1.

Then select the minimum gain edge $gainMin(s_i)$ within each segment $s_1, \ldots, s_{k-1}$, with the exception of the last segment s_k. We call these edges the ***cross edges***, as they are the edges crossing between our final segments. That is, our final segmentation $R = \{\langle x_i, y_i \rangle\}$ is the one induced by cutting the selected cross edges. Since each final segment spans at most two temporary segments, each of size at most $M/2$, each segment in R has total state at most M. Figure 2b shows this final segmentation — note that we cut the gain minimizing edge in each of the temporary segments. If there are two edges with the same gain, we can choose arbitrarily.

We next load balance the set of segments R across processors as follows. We define the ***load*** of a segment $r = \langle x_i, y_i \rangle$ as $load(r) = gain(x_i)\, in(x_i) + gain(y_i)\, out(y_i)$, i.e., the sum of the gains of the incoming edge and the outgoing edge. Our load-balancing also employs a simple greedy strategy. Specifically, calculate the total load $L = \sum_{r \in R} load(r)$. The average load across P processors would thus be L/P. In pipeline order, greedily place segments from R on the current processor until the total load on the processors exceeds L/P, and then move to the next processor. The in-order aspect of this greedy load balancing guarantees that each processor is assigned a contiguous set of segments. Therefore, for the example shown in Figure 2b, if we had 2 processors, the first segment on the first processor and the last three segments on the second processor.

It remains to define buffers on cross edges and the actual schedule. To simplify the description and analysis, we describe a version that schedules in synchronized rounds with a large period and correspondingly large buffers. To calculate the buffers on cross edges and the period, we choose the value $X = M \prod_{\text{modules } i} in(i) \cdot out(i)$. In this way, for every edge e (and in particular for cross edges), we have that $X\, gain(e)$ is an integer larger than M. We add buffers of size $X\, gain(c)$ to every cross edge c. Our actual implementation (which is also synchronous) does not employ buffers this large and the buffer sizes can be further reduced by scheduling asynchronously.

The schedule itself is periodic and divided into rounds, with synchronization between rounds. In each round, each processor loads each of its segments exactly once in pipeline order; once a segment is loaded, the contained modules are fired many times. Specifically, for $\langle x_i, y_i \rangle$, if the input buffer is full — that is, there are $X\, gain(x_i)\, in(x_i)$ messages available on the incoming edge — the segment is ready and is run.[1] To run the segment $\langle x_i, y_i \rangle$, execute the latest module within the segment with enough inputs available to fire. Repeat until y_i has fired $X\, gain(y_i)$ times. By careful construction of X, all modules $j \in \langle x_i, y_i \rangle$ have fired $X\, gain(j)$ times, and hence there are no messages on any internal edges, and the subsequent segment becomes ready.

B. *Bounding* seg_cache*'s Performance*

We now prove an upper bound on the number of cache misses incurred on each processor with respect to its load, which is defined as follows. Let R_p be the set of segments assigned to p by *seg_cache*. Then the ***load of processor p*** is defined as $procLoad(p) = \sum_{r \in R_p} load(r)$.

Lemma 5: In each round of *seg_cache*, the total number of cache misses incurred by p is $O((X/B)\, procLoad(p))$.

Proof: When executing a segment $r = \langle x, y \rangle$, its entire state is loaded just once, since it fits in cache. (By assumption, the state includes a small buffers on internal edges to accommodate varying input/output rates.) Thus when running r, the only cache misses are from loading the state initially, and reading/writing messages from/to the incoming/outgoing cross edges. The state load costs $O(M/B)$. The cost on cross edges is $O(X\, gain(x)\, in(x)/B)$ and $O(X\, gain(y)\, out(y)/B)$, respectively, which sum to

[1] Otherwise, wait until the next round. This waiting occurs only in the earliest rounds until a steady state is reached — the ith processor first becomes ready in the ith round.

$O((X/B)\,load(r))$. Since $X\,gain(e) \geq M$ for all edges e, $O((X/B)\,load(r))$ dominates. Summing over all segments r assigned to p completes the proof. ■

We obtain the following corollary on time by taking the max of Lemma 5 across all processors.

Corollary 6: The duration of each round is $O((X/B)\max_{\text{proc. } p} procLoad(p))$ parallel I/Os. ■

We now upper-bound the running time of *seg_cache*.

Lemma 7: For sufficiently large T, the time required for *seg_cache* to fire the sink node t a total of $T\,gain(t)$ times is $O((T/B)\max_{\text{proc. } p} procLoad(p))$ parallel I/Os.

Proof: If a segment fires X times its gain times each time it becomes ready, then a simple inductive argument shows that all the segments on processor i first become ready in round i. Moreover, once ready, they continue to become ready again in each subsequent round. Thus, after Z rounds, the last processor's segments run during at least $Z - P$ of them, and hence t fires at least $(Z - P)X\,gain(t)$ times. From Corollary 6, the total time required for Z rounds is $O(Z \cdot (X/B)\max_{\text{proc. } p} procLoad(p))$. As long as $T \geq PX$, choosing $Z = \lceil 2T/X \rceil$ completes the proof. ■

The following theorem states that *seg_cache* is asymptotically optimal, when given a constant factor memory augmentation. Note that in the theorem statement, $procLoad(p)$ is a metric of the pipeline — we are not just saying that *seg_cache* is optimal for this particular static schedule, but rather that no other static schedule is much better.

Theorem 8: Let $procLoad(p)$ denote the processor loads arising from an execution of *seg_cache* on the pipeline. Then for sufficiently large T, every static schedule on a machine with cache size of $M/6$ requires a runtime of $\Omega((T/B)\max_{proc.\ p} procLoad(p))$ parallel I/Os to fire the sink node t a total of $T\,gain(t)$ times.

Proof: Consider the temporary segmentation S chosen by *seg_cache*, and let C be the set of gain-minimizing cross edges selected. Each of the segments (except the last which contributes no cross edge) has size at least $2(M/6)$, and thus applying Theorem 4 for a machine with $M/6$ allows us to conclude that $\Omega((T/(PB))\sum_{c\in C} gain(c) + (T/B)\max_{c\in C} gain(c))$ is a lower bound on the runtime of any static scheduler.

We next show $(1/P)\sum_{c\in C} gain(c) + \max_{c\in C} gain(c)) \geq (1/2)\max_{proc.\ p} procLoad(p)$, which completes the proof. Cross edges contributes to the load of two segments, so the total load is $L = 2\sum_{c\in C} gain(c)$. The *seg_cache* algorithm adds at most one segment to a processor after it reaches L/P load. Each segment borders two cross edges, so the load of any segment is at most $2\max_{c\in C} gain(c)$. Thus the maximum load on any processor is $\max_{proc.\ p} procLoad(p) \leq (2/P)\sum_{c\in C} gain(c) + 2\max_{c\in C} gain(c)$. ■

V. Experiments

We now experimentally compare the running time of our strategy against several other static policies when executing randomly generated pipelines drawn from distributions with varying characteristics. The experiments indicate that under many conditions, cache-based segmentation is able to improve the overall running time, while under other conditions, computation time based partitioning performs better. We also investigate a heuristic that combines computation and cache misses as a basis of partitioning, and find that it often provides the best of both worlds.

A. Scheduling Policies

We evaluated several static schedulers. Each schedule is characterized by 3 features: (1) The segmentation. We describe all of our schedules in terms of segments — a set of consecutive modules restricted to the same processor. Note that a segment may contain anything between one module and all the modules. How segments are chosen varies by scheduling policy. (2) The processor assignment. Each schedule defines how the segments are (statically) assigned to processors. Multiple segments may be assigned to the same processor. (3) Cross-edge buffer allocation. Each scheduling policy also defines the size of buffers to place on cross edges — the edges that go between segments.

All the other details for all policies are similar. For any internal edge e from module u to module v of the same segment the buffer of size $2\text{lcm}(out(u), in(v))$ — that is the least common multiple of the number of items produced and consumed on the edge — is allocated.[2] For all policies except *seg_cache*, the cross edge buffers are also assigned in the same manner. In all the computation-based policies (*seg_runtime*, *bin_full* and *bin_empty*), before load-balancing, we normalize the modules computation cost based on the gain of the module. The scheduling policies compared in this section are defined as follows:

1) *seg_cache*: Segmentation based on cache (our policy) as described in Section IV. Each cross edge e has a buffer of $(M/2)\,gain(e)$.
2) *random_assignment*: Segments consists of single modules and are assigned to processors at random.
3) *seg_random*: Random segmentation. A single segment is created per processor by randomly picking $p - 1$ edges to be cross edges.
4) *seg_runtime*: Segmentation based on computation cost. In this policy a single segment is created per processor minimizing the maximum number of compute steps on a single segment by using a greedy load-balancing policy analogous to that of *seg_cache*.
5) *bin_full*: Fullest first binpacking on computation. Assign modules to a segment until their cumulative computation is just smaller than $1/P$ fraction of the total computation, creating P large segments, one for each processor. The remaining modules are in their

[2]This internal buffer could be sized $out(i) + in(i + 1)$, but the larger quantity allows for easier scheduling and does not affect the performance.

own segments and assigned greedily to the processor with the smallest overall compute cost. This policy often results in a segmentation nearly identical to *seg_runtime*.

6) *bin_emptiest*: Emptiest first binpacking on computation. Each segment consists of a single module. We greedily assign segments to the processors with the fewest total compute steps.

B. Pipeline Execution

We built an execution engine that takes in a pipeline and the static-schedule description and executes the pipeline according to that schedule. The execution engine runs exactly one thread on each processor. The ***master thread*** begins execution and is responsible for parsing the schedule and spawning all the remaining ***worker threads*** and pinning them on their designated processors. These worker threads are responsible for executing the segments assigned to respective processors. The master thread executes the source node, generating all the subsequent messages to be processed, and the sink node, consuming all the messages after processing.

In a loop, each worker thread finds a segment that is ***ready*** — the buffer on the input cross edge is at least half-full and the buffer and the output cross edge is at least half-empty. This segment is then loaded into memory. At this point, this segment is executed until either the buffer on the input cross edge is empty or the buffer on the output cross edge is full. If there are multiple ready segments, the worker thread always executes the last (in pipeline order) ready segment. Once a ready segment is picked for execution, it is executed until either the buffer on its incoming cross edge is empty or the buffer on its outgoing cross edge is full. When a segment is executed, its internal modules are fired one at a time. Any schedule of firing its internal modules is valid — in our execution engine, the last ready module (in order of the pipeline) is always fired.

The internal-edge buffers between modules within a segment are implemented as simple FIFO queues. No synchronization is required on internal edges because all modules within a segment are consecutive and are scheduled on the same processor. Neighboring segments can, however, be scheduled on different processors, and we thus use thread-safe single-producer single-consumer lock free ring buffers for communication through cross edges between segments

C. Pipeline Generation

Our experimental evaluation consists of running each schedule on randomly generated pipelines. We fixed the total number of modules to 140 in order to allow for sufficient differences in the schedules generated. For each pipeline, we randomly generate three sets of parameters: (1) gain of each edge, (2) state size of each module, and (3) computation requirement of each module. The gain of each edge is picked randomly between $[1, M/16]$, where M is the size of the cache. The upper limit is picked so that the maximum gain is pretty large, but still allows all non-cross edge buffers to fit in memory. The state size of each module is picked randomly between $[1KB, 32KB]$, again in order to allow the each module to individually fit in cache. Finally, the module computation times are selected from $[0.5\mu s, 50\mu s]$. We used two kinds of distributions for generating our parameters — namely ***uniform distribution*** and ***zipf's distribution***, [13]. Zipf's is a heavy tailed distribution with a probability density is given by $P(x) = \frac{x^{-p}}{\zeta(p)}$. Here ζ is the Riemann zeta function and p is a positive parameter which controls how heavy the tail is. For our experiments we use $p = 1.5$

D. Experimental Settings

All of our experiments were run on Intel Eight Core Xeon E5-4620 2.2Ghz processors with 8-way set associative L2 caches of $M = 256$KB and block size $B = 64$ bytes. Each message is the size of a word (4 bytes) and the source node generates $16384M$ ($16384 \times 1024 = 16777216$) messages in total during each execution. This large number ensures that the overhead of loading and clearing the pipeline is low compared to the total execution time — therefore the total execution time is a reasonable approximation of the steady-state throughput.

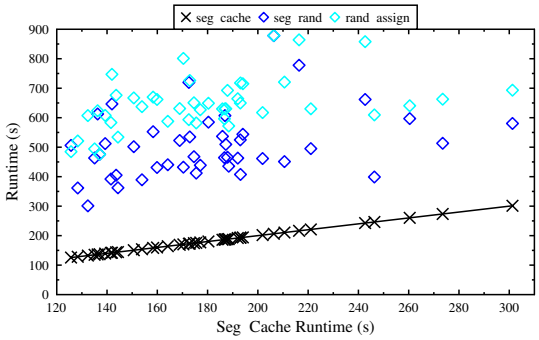
Each chart consists of a set of trials using the same parameters to randomly generate the pipelines. In all charts, the y-axis is running time — lower is better. A vertically aligned set of points plot the running times of each scheduler on the same pipeline. The x-axis is the running time of *seg_cache*. By construction, the running time of *seg_cache* fits a line, which is shown. If most of the points for a particular strategy fall above the line, then that strategy is generally worse than cache-based segmentation; if the points generally fall below the line, then it is better than cache-based segmentation.

E. Disabled Computation

We first conduct a simple experiment to see if cache performance has any effect on runtime. We set the computation time for all modules to 0 and compare *seg_random* and *rand_assign* scheduling policies against *seg_cache* (the other policies depend on computation time).

Figure 3a shows the results for pipelines that have been generated using a zipf distribution for the gains and a uniform distribution for state size. We see that *seg_random* generally requires at least twice as much time to complete as *seg_cache*. While *seg_random* is able to take advantage of the temporal locality arising from segmentation in general, a random selection does not properly reduce the cost on cross edges, resulting in some bottlenecked processor. Since *rand_assign* lacks this temporal locality, it performs even worse. In Figure 3b we see that when a uniform distribution is used for gain selection *seg_random*'s performance is much closer to *seg_cache*'s. This is due to the fact that *seg_cache*

(a) Zipf gain, uniform state, uniform computation



(b) Uniform gain, uniform state, uniform computation

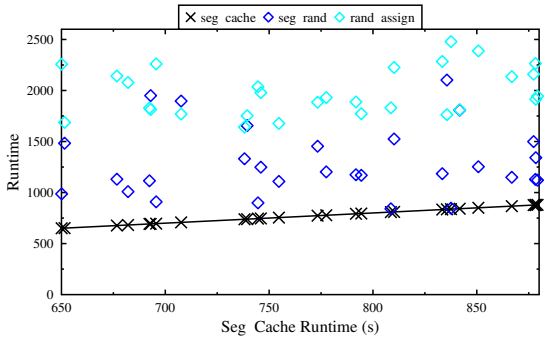


Figure 3: Normalized results for *seg_rand* and *rand_assign* against *seg_cache* with computation simulation disabled. We see that *seg_cache* works better with zipf gain than with uniform gain.

concentrates on cutting low gain edges; with uniform distribution, there is little difference between a "good edge" (low gain edge) and a random edge. Therefore, the primary advantage of *seg_cache* is reduced compared to *seg_random*. Due to the lack of temporal locality, *random_assign* still performs much worse. In subsequent experiments, we will omit the *random_assign* policy since it almost always much worse than the others.

This experiment provides evidence that cache performance of scheduling policy can play a role in determining the running time. In addition, a uniform distribution on the gains decreases the effectiveness of a cache-based segmentation due to the increased number of expensive edges which makes it difficult to pick "cheap" cross edges. Zipf's like heavy-tailed distributions commonly occur in practice in many circumstances and some studies [14]–[16] claim that the sizes of Unix jobs follow a heavy-tailed distribution. Therefore, it seemed reasonable to try these distributions in our experiments.

F. Randomly Generated Computation Times

The remainder of the experiments will consider computation times. Figure 4 shows the results when state sizes are selected uniformly at random, but the distribution of the gain and computation varies. As in the case where there is no computation, we again see, in Figures 4a and 4b, that uniform gains are not particularly good for *seg_cache* and computation-based segmentation *seg_runt* performs as well or better. The reason is similar as in the case of disabled computation — when the gains are uniformly distributed, *seg_cache* has very little ability to pick better cross edges (smaller gain cross edges) than other policies.

We see another interesting effect here — *seg_runt* is better with respect to *seg_cache* when computation is distributed uniformly (Figure 4a), while it is comparable to *seg_cache* when computation is generated using the zipf's distribution (Figure 4b). This is due to the difference in distributions: Our uniform distribution has a higher mean than the zipf distribution; therefore, the overall runtime is larger with the uniform distribution than it is with the zipf's distribution. Therefore, computation cost is more likely to dominate the

(a) Zipf's Gain, Uniform State, Correlated Computation

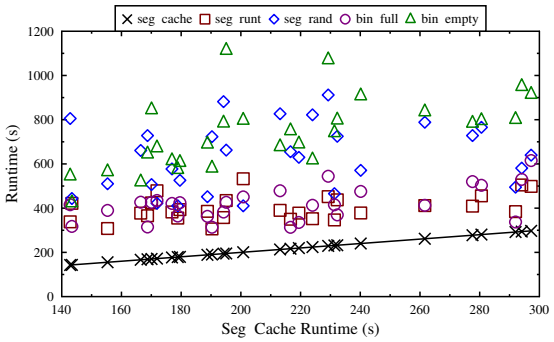


Figure 5: *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* for correlated state size and computation time.

performance with uniform distribution on computation and hence, a policy that load-balances computation performs better. On the other hand, under the zipf's distribution, the computation of most modules is small; therefore, the overall computation is small and the cache effects can become more prominent and *seg_cache* can start to have some advantage.

In comparison, (just as with disable computation) when the gains are zipf's distributed, in Figures 4c and 4d, we see that *seg_cache* generally outperforms the other policies. In particular, even when the computation is uniformly distributed (Figure 4c) the ability to pick good cross edges allows *seg_cache* to perform better than other policies. Again we see in Figure 4d that when computation time is selected with a zipf distribution, cache performance plays an even higher role and *seg_cache* performs the best.

We ran other combinations of distributions (omitted due to space constraints) and these general trends seem to hold. The state-size distribution generally does not seem to have an appreciable effect on the relative performance of policies.

G. Correlated Computation Time

Thus far our experiments have selected all gains, state sizes and computation times independently. It is, however, often the case in practice that state size and computation time are correlated. Figure 5 shows the results when we pick the state s using the uniform distribution, and then the computation time of the module is its $(s/\ maxState) \times maxComp$

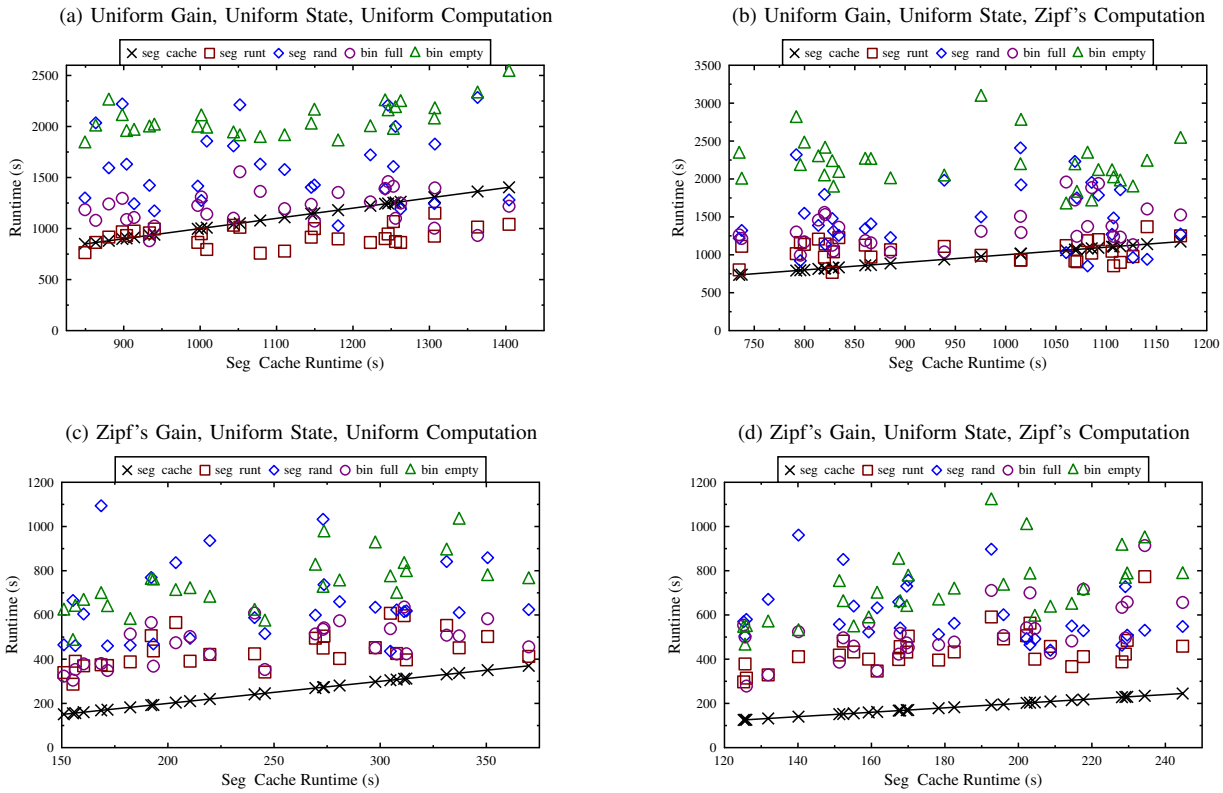


Figure 4: Normalized results for *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* with computation time enabled. We see that *seg_cache* performs the best when both the computation time and the gains have a zipf's distribution. This is due to the fact that (1) zipf's distribution on computation time reduces the total computation time of the pipeline, making it more likely that the runtime is dominated by cache misses and (2) zipf's distribution on gains allows *seg_cache* more leeway to pick better cross edges.

where $maxState = 32\text{KB}$ and $maxComp = 50\mu s$. Again we see that *seg_cache* outperforms all scheduling policies in the majority of cases. We see that the results are quite similar to Figure 4c, which is not surprising since again state size and computation are uniformly distributed. In fact, the distribution on computation times is not identical; the range of computation times for the un-correlated experiment shown (Figure 4c) is $[0.5\mu s, 50\mu s]$ while for the correlated experiment it is $[1.56\mu s, 50\mu s]$. Therefore, the overall computation time of these correlated pipelines generally larger and therefore, we would expect *seg_cache* to perform worse. However, since *seg_cache* balances the state sizes in its segmentation, due to the correlation, in this case, it appears to also manage to balance the computation times to a certain extent, leading it to perform better than other strategies.

H. Segmentation Based on Computation Time and Cache

These experiments indicate that segmentation is good, since either cache-based segmentation (*seg_cache*) or computation-based segmentation (*seg_runt*) generally dominates the other policies. Therefore, we experimented with a new algorithm, *seg_both*: a segmentation strategy that considers both computation and cache misses. For this policy, we first estimate the time for a cache miss (we found it to be about 3 nanoseconds). We then calculate the normalized load for all possible segments $\langle x, y \rangle$ that fit in cache by including both the time spent incurring cache misses and the time spent on modules' computation. We then greedily assign contiguous segments to processors so as to minimize the maximum load per core via a binary search algorithm. Since this algorithm considers cache misses, the buffers on cross edges are assigned in the same manner as *seg_cache*.

Figure 6 shows the results for this policy for both the case of uniform gain, uniform computation and for zipf gain, zipf's computation. If we compare Figure 6a to Figure 4a, we see that in the case where computation-based segmentation is better than cache-based segmentation, this combined segmentation policy beats both *seg_cache* and *seg_runt*. On the other hand, comparing Figure 6b to Figure 4d indicates that when cache-based segmentation is better, this combined segmentation does almost as well as *seg_cache*. Therefore, this policy seems to provide the best of both worlds.

I. Fixed State and Computation

In order to better understand these scheduling policies, we conducted experiments where the module state sizes and computation times have been fixed and only the gains vary. Figure 7 shows three experiments, using 'small','medium' or 'large' values, where $state \in \{\frac{M}{1024}, \frac{M}{16}, \frac{M}{2}\}$ and $computation \in \{1, 10, 50\}\mu s$. In all

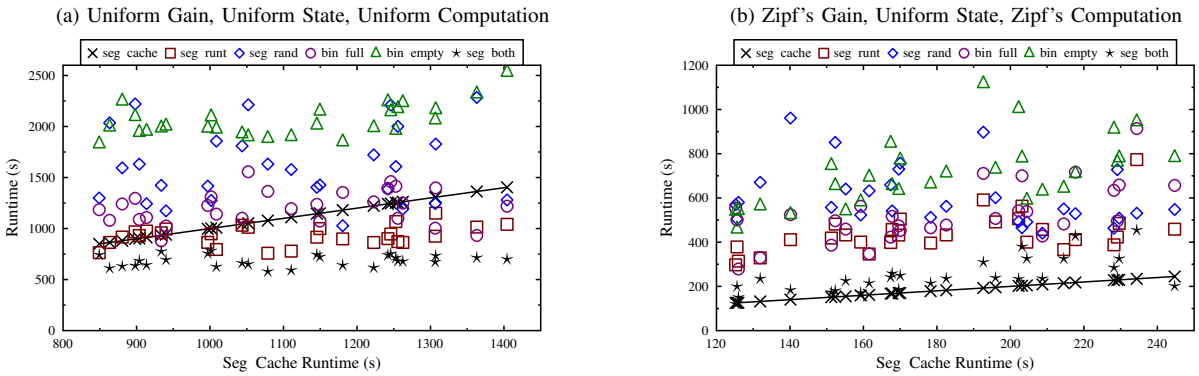


Figure 6: Segmentation based on both computation time and cache. We see that this form of segmentation does a better job of load-balancing computations when computation-based strategies dominate, but also gets most of the advantage of cache-based segmentation when *seg_cache* dominates.

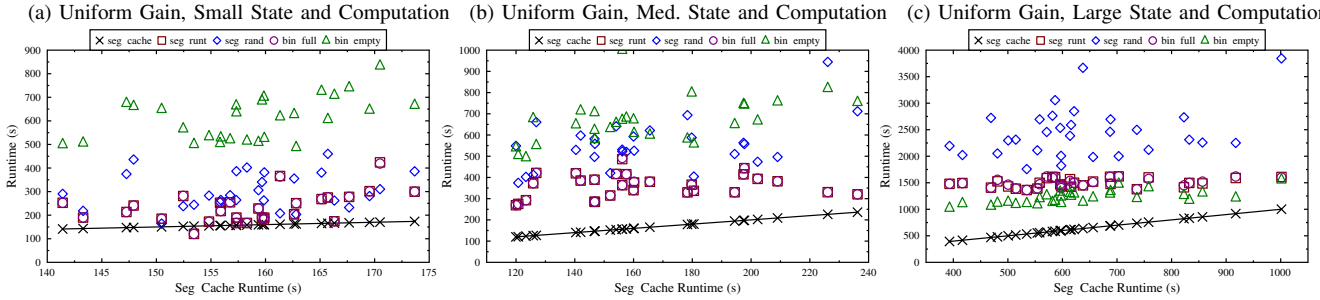


Figure 7: Normalized results *seg_runtime*, *seg_random*, *bin_fullest* and *bin_emptiest* against *seg_cache* when state size and computation times have been fixed and gains are selected with a zipf distribution.

experiments, *seg_cache* performs as well as or better than the other policies, since balancing the state sizes automatically balances the computation as well.

In Figure 7a both *bin_fullest* and *seg_runtime* are able to perform nearly as well as *seg_cache*. Note that the total state of the entire pipeline is small. Since both these policies create approximately 1 segment per processor and these segments fit in cache. Therefore, these policies get most of the advantage of *seg_cache* since they never have to pay to reload the state and only pay for 2 cross edges per processor. *bin_empty* is significantly worse since it has too many cross edges. As we increase the state sizes, the modules assigned to each processor no longer fit entirely in cache and the cost of loading segments starts mattering.

Another interesting trend is that *bin_empty* starts performing better as the computation and state size increase. The reason for this is subtle. Since *seg_runt* and *bin_full* essentially perform segmentation based on computation cost, they put large contiguous segments on each processor as the state size and computation increases. At the largest size, each processor's cache can only fit 1 or 2 modules at a time. Therefore, both these policies load a single module, execute it, and then load the next module, and so on, paying a large cache cost for each firing. On the other hand, *bin_empty* distributes contiguous modules across processors with each segment consisting of a single module — therefore, for large state sizes, it is essentially doing what *seg_cache* does, but without the advantage of large buffers on cross edges. This policy potentially allows it to execute each module many times before loading the next module.

VI. Related Works

Most existing work on scheduling streaming computations computation costs of the modules only and tries to balance to computation across processors in order to maximize throughput. Bokhari solved the throughput optimization problem for pipeline mapping by finding a minimum bottleneck path in a layered graph that contains all information about application modules [17]. Hansen et al. later improved Bokhari's solution using dynamic programming [18]. Subsequently, many efficient algorithms have been proposed, both for homogenous processors [19]–[21] and heterogenous processors [22]. Researchers have also considered distributed memory communication models of various kinds to understand the effect of communication on throughput of pipeline computations [23], [24]. Researcher have also considered the problem of maximizing throughput and minimizing latency as a bi-criteria scheduling problem [25], [26]. In addition, replication of state-less modules in order to improve throughput has also been studied [27]–[30]. None of this research explicitly addresses the problem of minimizing the number of cache misses.

Heuristic cache-aware scheduling of streaming programs on both single processors and multiprocessors has been

studied by several research groups [31]–[33]. Since all of these algorithms are based on heuristics, they do not guarantee optimality with respect to the number of cache misses. However, their empirical results support our claim that optimizing the number of cache misses can lead to significant improvement in performance. The only prior theoretical work that considers the number of cache misses for streaming applications considers only single processor schedules in both the cache-aware [11] and cache-oblivious [34] setting.

VII. Conclusion and Future Works

This paper has studied cache-conscious scheduling of streaming pipelines on machines with private caches. In most of the experimental benchmarks, scheduling based solely on the modeled cache effects is indeed significantly more effective than the more common load-balancing of computation costs. In the case that computation cost dominates the workload, a mixed strategy that takes into account both cache misses and computation cost outperforms one designed to balance the computation. We conclude that the cost of cache misses should not be ignored when designing scheduling strategies for streaming pipelines.

From a theoretical perspective, this paper has developed matching upper and lower bounds in the parallel external memory model. One natural question is how the results extend to a model including computation cost, or more generally on multiple levels of private caches. We were also surprised to discover that the lower bound does not extend to non-static schedules. A non-static schedule violates the conventional practice of keeping a module stationary and sending messages across processors, instead keeping the messages somewhat stationary and sending modules across processors. It would be interesting study non-static schedulers more fully, and to see how they perform in practice.

Finally, this paper has only considered private caches. It would also be interesting to explore streaming pipelines on shared caches and mixed cache hierarchies.

References

[1] B. Khailany, W. Dally, S. Rixner, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, and A. Chang, "Imagine: Media processing with streams," *IEEE Micro*, pp. 35–46, March/April 2001.

[2] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart, "Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer," in *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2006, pp. 59–66.

[3] Y. Liu, N. Vijayakumar, and B. Plale, "Stream processing in data-driven computational science," in *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, 2006, pp. 160–167.

[4] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data streams: a review," *SIGMOD Rec.*, vol. 34, no. 2, pp. 18–26, June 2005.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[6] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, Aug. 2003.

[7] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[8] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, September 1988.

[9] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, Jul. 2005, pp. 228–237.

[10] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2008, pp. 501–510.

[11] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo, "Cache-conscious scheduling of streaming applications," in *Proceedings of the 24th ACM Symposium on Parallelism in algorithms and architectures*, 2012, pp. 236–245.

[12] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 2008, pp. 197–206.

[13] G. K. Zipf, *Selected studies of the principle of relative frequency in language*. Cambridge, Mass. : Harvard University Press, 1932.

[14] W. Leland and T. J. Ott, "Load-balancing heuristics and process behavior," in *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation*. New York, NY, USA: ACM Press, 1986, pp. 54–69.

[15] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 253–285, 1997.

[16] M. Harchol-Balter, "The effect of heavy-tailed job size. distributions on computer system design," in *Proceedings of ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics*, 1999.

[17] S. H. Bokhari, "Partitioning problems in parallel, pipeline, and distributed computing," *IEEE Trans. on Computers*, vol. 37, no. 1, pp. 48–57, Jan. 1988.

[18] P. Hansen and K.-W. Lih, "Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. on Computers*, vol. 41, no. 6, pp. 769 –771, Jun. 1992.

[19] H.-A. Choi and B. Narahari, "Algorithms for mapping and partitioning chain structured parallel computations," in *Pro-*

ceedings of the International Conference on Parallel Processing, 1991, pp. 625–628.

[20] B. Olstad and F. Manne, "Efficient partitioning of sequences," *IEEE Transactions on Computers*, vol. 44, no. 11, pp. 1322–1326, 1995.

[21] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, 2004.

[22] A. Benoit and Y. Robert, "Mapping pipeline skeletons onto heterogeneous platforms," *Journal of Parallel and Distributed Computing*, vol. 68, no. 6, pp. 790–808, 2008.

[23] K. Agrawal, A. Benoit, F. Dufossé, and Y. Robert, "Mapping filtering streaming applications with communication costs," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, 2009, pp. 19–28.

[24] K. Agrawal, A. Benoit, F. Dufoss, and Y. Robert, "Mapping filtering streaming applications," *Algorithmica*, pp. 1–51, September 2010.

[25] A. Benoit and Y. Robert, "Complexity results for throughput and latency optimization of replicated and data-parallel workflows," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, Sept 2007, pp. 497–506.

[26] A. Benoit, M. Hakem, and Y. Robert, "Optimizing the latency of streaming applications under throughput and reliability constraints," in *International Conference on Parallel Processing*, Sept 2009, pp. 325–332.

[27] J. Subhlok and G. Vondran, "Optimal mapping of sequences of data parallel tasks," in *Proceedings of 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995, pp. 134–143.

[28] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 114–124.

[29] D. Cordes, A. Heinig, P. Marwedel, and A. Mallik, "Automatic extraction of pipeline parallelism for embedded software using linear programming," in *Proceedings of IEEE 17th Interenational Conference on Parallel and Distributed Systems*, 2011, pp. 699–706.

[30] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Adding data parallelism to streaming pipelines for throughput optimization," in *Proceedings of the 20th International Conference on High Performance Computing*, 2013, pp. 20–29.

[31] S. Kohli, "Cache aware scheduling for synchronous dataflow programs," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M04/3, 2004.

[32] A. Moonen, M. Bekooij, R. Van Den Berg, and J. Van Meerbergen, "Cache aware mapping of streaming applications on a multiprocessor system-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2008, pp. 300–305.

[33] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, "Cache aware optimization of stream programs," *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 115–126, 2005.

[34] K. Agrawal and J. Fineman, "Brief announcement: Cache-oblivious scheduling of streaming pipelines," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2014, pp. 79–81.