Brief Announcement: Cache-Oblivious Scheduling of Streaming Applications

Kunal Agrawal
Washington Univ. in St. Louis
kunal@cse.wustl.edu

Jeremy T. Fineman
Georgetown University
jfineman@cs.georgetown.edu

ABSTRACT

This paper considers the problem of cache-obliviously scheduling streaming pipelines on uniprocessors with the goal of minimizing cache misses. Our recursive algorithm is not parameterized by cache size, yet it achieves the asymptotically minimum number of cache misses with constant factor memory augmentation.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: General

Keywords

Cache-Oblivious Algorithms; Caching; Partitioning; Pipelines; Scheduling; Streaming; Synchronous Data Flow.

1. INTRODUCTION

As parallel processors such as multicores have become more prevalent, there has been an increasing interest in parallel programming paradigms such as ***streaming***. Examples include academic projects like StreamIt [14] and StreamC/KernelC [8], community-based open-source projects like GNU Radio [7], and commercial products including Simulink® [11] and LabVIEW [10]. Streaming is used to express high-throughput applications such as audio and video processing, biological sequence or astrophysics data analysis, and financial modeling. In general a streaming application can be represented by a graph; in this paper, we restrict our attention to ***streaming pipelines*** — a chain topology commonly used in steaming applications.

Streaming Pipelines: A streaming pipeline consists of a sequence of n ***computational modules***, $1, 2, \ldots, n$, and each module i has exactly one ***incoming channel*** (from the previous module $i-1$) and one ***outgoing channel*** (to the subsequent module $i+1$). The modules send data, in the form of ***messages*** or ***data items*** to each other via these channels. We assume that the incoming channel into module 1 (the first module) streams an infinite amount of data into the pipeline and the outgoing channel from module n streams it out.

Each module i has an associated state; we denote the size of this ***state*** by $s(i)$. In order to execute, or ***fire*** a module i, the entire state of that module must be loaded into the cache. A standard (discrete) model is that when the module fires, it consumes $in(i)$ data items from its incoming channel, performs some computation, and then produces $out(i)$ data items on its outgoing channel, where $in(i)$ and $out(i)$ are static parameters of the module. This paper considers a slightly easier continuous model — when a module fires, it may consume any amount x of available data from its incoming channel, then produce $x \times in(i)/out(i)$ data on its outgoing channel. The key distinction in the continuous case is that it may produce a fractional amount of data, which alleviates any bottlenecks in the pipeline caused by integrality.

Cache-Efficient Scheduling: Cache efficiency is an important determinant of performance, and many cache-efficient algorithms have been studied (see [1, 4, 6] for a sample). In this paper, we adopt the ideal-cache model [6], which is an extension of a classic two-level memory model [1]. In this model, there is a fast cache of size M connected to slower storage, and each load (cache miss) moves a contiguous chunk of data, or ***block***, of size B into cache.

Streaming applications exhibit two kinds of cache misses that can be controlled using intelligent scheduling. First, since modules access their state when they fire, it is advantageous to execute the same module many times once its state has been loaded. Second, it is advantageous to execute consecutive modules, say i and $i+1$ in quick succession in order to keep the data produced by module i on its output channel in cache until module $i+1$ consumes it. Since these heuristics are contradictory, we must balance concerns intelligently to design a good scheduling algorithm. While there has been extensive research, both theoretical and empirical, on scheduling streaming pipelines to optimize throughput and/or latency [9, 2, 5], most prior work on cache-efficiency has been empirical [13, 12].

In previous work [3], we show that if the size of the cache M is known in advance, then one can create a partitioned schedule that provides asymptotically optimal cache performance given constant factor cache augmentation. That is, if the optimal schedule, given a cache of size M, has X cache misses, then this partitioned schedule has $O(X)$ cache misses, given a cache of size $O(M)$. In the present paper, we extend a similar result to cache-oblivious [6] scheduling, albeit in a simpler continuous model. Specifically, without parameterizing by cache size M or block size B, we design a schedule that is asymptotically optimal in the number of cache misses given a constant-factor memory augmentation.

Assumptions and Definitions: We use the term "gain" to describe the amplification of messages along the pipeline. In particular, for an edge from module i to $i+1$, the ***gain*** of the edge is defined as the number of messages produced along the edge for each input consumed by the first module in the pipeline. Therefore $gain(i) = \prod_{j=1}^{i}(out(j)/in(j))$.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
ACM 978-1-4503-2821-0/14/06.
http://dx.doi.org/10.1145/2612669.2612707.

In addition, we define some terms for a given partition of the pipeline from module i to module j, denoted by $\langle i, j\rangle$. The total size of the pipeline partition $\langle i, j\rangle$ is denoted by $total(i, j) = \sum_{k=i}^{j} s(k)$. Similarly, the largest module within the partition is $max(i, j) = \max_{k=i}^{j} s(k)$. Throughout the paper, we assume that the size of each module is at most $M/6$; that is, $max(1, n) \leq M/6$. Even the optimal algorithm requires that $max(1, n) \leq M$ to allow a module to be in cache when fired; we allow a factor of 6 augmentation over optimal.

2. CACHE-OBLIVIOUS PARTITIONING AND THE SCHEDULING ALGORITHM

This section describes the cache-oblivious scheduling algorithm for streaming pipelines. As with many cache-oblivious algorithms, this scheduling algorithm is recursive, meaning that contiguous subpipelines are scheduled recursively with buffers in between to accommodate messages. One of the challenges is controlling the amount of space used by buffers.

At the highest level, consider the entire pipeline $\langle 1, n\rangle$. Define $X = total(1, n)$ and $BASE = 6max(1, n)$. Here, $BASE$ specifies a base case for the recursion (even though M is unknown to the algorithm, the model assumes $M \geq 6max(1, n)$, as stated in Section 1). Without loss of generality, we assume the inputs to the entire pipeline arrive on a size-X input buffer B_1, and the output edge from the last module has an infinite-size buffer B_{n+1}.

Schedule: The scheduling is performed by recursively calling FIRE, specified in Figure 1. The call FIRE(i, k, B_i, B_{k+1}) corresponds to repeatedly executing the sub-pipeline $\langle i, k\rangle$ until (1) all of the inputs are removed from B_i, and (2) all buffers $B_i, \ldots, B_k$ are empty, i.e., all messages have moved to the output. The algorithm is carefully constructed so that B_{k+1} has enough capacity to store all outputs. The top-level call is FIRE$(1, n, B_1, B_{n+1})$, which may be repeated whenever X inputs become available to the pipeline.

The main idea of the algorithm is to recursively cut the pipeline into pieces that have nearly equal state, insert a buffer of size roughly their state between them, and then recursively schedule each of those pieces. For conciseness, this partitioning is specified in Figure 1, but it could be performed statically *a priori*.

The following lemma states that the preconditions and postconditions hold. The important feature here is that emptying internal buffers allows us to reclaim space (last line in Figure 1), and hence we can relate the (dynamically allocated) space used by the algorithm to the state in the subsequent lemma. This in turn leads to the corollary as long as the algorithm is careful about allocating space.

LEMMA 1. *Any recursive call made to* FIRE(i, k, B_i, B_{k+1}) *moves all x inputs from B_i to B_{k+1} without 1) overflowing the buffer B_{k+1} or 2) leaving any data in internal buffers.*

PROOF. To prove (2), assume inductively that the statement holds for lower levels in the recursion. (It trivially holds for the base case.) Observe that data moved to the middle buffer B_j is drained by inductive assumption in FIRE(j, k, B_j, B_{k+1}); hence all internal buffers are empty. The input buffer is also drained by construction.

For (1), induct starting from the top of the recursion, where it holds trivially since B_{n+1} has infinite size. It holds by construction for the next level of recursion as the number of times the first subpipeline fires is bounded by the size of B_j (i.e., case 1). □

LEMMA 2. *Let $t = total(i, k)$. During* FIRE(i, k, B_i, B_{k+1}) *the following are true: 1) At any point, the total recursively allocated buffer space that has not been freed is $\Theta(t)$, and 2) the input buffer B_i has size $\Theta(t)$.*

```
FIRE(i,k,B_i,B_{k+1})          //execute ⟨i,k⟩ = i, i+1, ..., k
    // let x be the number of inputs on B_i
    // Precondition: B_{k+1} has capacity for all inputs on B_i, i.e.,
    //       i.e., at least x gain(k)/gain(i-1)
    // Postcondition: the input and all internal buffers are empty
 1  let t = total(i,k)
 2  if t < BASE
 3          fire all modules in order until the input buffer is empty.
 4  find the module j having minimum gain edge from j-1 to j
       such that min{total(i,j-1), total(j,k)} ≥ t/3
 5  create buffers B'_i and B_j with size t
 6  if gain(j-1) ≥ gain(i-1)          //(case 1)
 7          repeat ⌈x*(gain(j-1)/gain(i-1))/t⌉ times
 8                  move t/(gain(j-1)/gain(i-1)) items from B_i to B'_i
 9                  FIRE(i, j-1, B'_i, B_j)
10                  // there is t data is buffered at B_j
11                  FIRE(j,k,B_j,B_{k+1})
12  else  // gain(j+1) < gain(i)          (case 2)
13          repeat ⌈x/t⌉ times
14                  move t inputs from B_i to B'_i
15                  FIRE(i, j-1, B'_i, B_j)
16                  // there is < t data buffered at B_{j+1}
17                  FIRE(j,k,B_j,B_{k+1})
18  deallocate or free buffers B'_i and B_{j+1}
```

Figure 1: Recursive pseudocode for the schedule.

PROOF. Since each partition has size at least $t/3$, the total state size of either subpipeline is at least $t/3$ and at most $2t/3$. Since each recursive call is given a buffer of size t, (2) holds by construction at each level of recursion. To prove (1), observe that every recursive call frees any space it allocates, and hence the outstanding space corresponds to the recurrence $S(t) \leq S(2t/3) + 2t$, where $2t$ is the space allocated to B'_i and B_j. This recurrence solves to $\Theta(t)$. □

COROLLARY 3. *For any partition $\langle i, k\rangle$ if total state satisfying $total(i, k) \leq M$, then* FIRE(i, k, B_i, B_{k+1}) *incurs at most $O(M/B)$ cache misses in addition to misses incurred writing the outputs to B_{k+1}.*

PROOF. By Lemma 2, then entire subcomputation with the exception of the output fits in $\Theta(M)$ space, and hence we need only load this state and buffer space once. □

Finally, we prove the main theorem, which bounds the cache cost with respect to gains of edges crossing the recursive partition.

THEOREM 4. *Let $P_1, P_2, \ldots, P_2$ be the maximal recursive subpipelines, passed to* FIRE *calls, such that $total(P_\ell) \leq M$. Let C be the set of edges crossing between these sub-pipelines. Then* FIRE$(1, n, B_1, B_{n+1})$ *incurs a total of $O((X/B)\sum_{(k,k+1)\in C} gain(k))$ cache misses, where X is the number of inputs processed per* FIRE.

PROOF. From Corollary 3, we know that each time P_ℓ is FIREd, the cost of firing it is $O(M/B)$ plus the cost of the outputs. (The cost of outputs trivially bounded by the sum of the gains.) The goal is thus to show that we can charge this $O(M/B)$ cost against the gain of an edge in C. In particular, let FIRE(i, k, B_i, B_{k+1}) be the nearest ancestor call (i.e., $P_\ell \subseteq \langle i, k\rangle$) such that the loop (case 1 or case 2) has more than 1 repetition (or the root if no such ancestor exists). Let $e \in \{(i-1, i), (j-1, j), (k, k+1)\} \subseteq C$ be the edge with maximum gain from among the input/middle/output edges. We will charge firing P_ℓ against e.

Note that by assumption, any sub-pipeline P_ℓ charged to e only fires once per iteration of the repeat loop (it could only be more than once if a nearer recursive call had multiple iterations). Thus, if there are q iterations, then there is at most $q\sum_{P_\ell \in \langle i,k\rangle} O(M/B) = O(q \cdot total(i,k))$ work charged to e. Moreover, there must be at least $\Omega(q \cdot total(i,k))$ data moving through edge e to cause $q \geq 1$ iterations (due to bounded buffer size in case 1 or 2). Thus, the cost charged per data on e is at most $O(1/B)$, or $O(X\,gain(e)/B)$ in total. Summing across all edges in C completes the proof. □

We now argue that the cache-oblivious schedule shown in Figure 1 is asymptotically optimal given constant factor memory augmentation.

LEMMA 5. *If the optimal algorithm for scheduling pipelines has $\mathcal{M}$ memory and incurs Q cache misses, then the recursive schedule in Figure 1 incurs $O(Q)$ cache misses given $O(\mathcal{M})$ memory.*

PROOF. Our previous lower bound [3] argues the following: Consider any arbitrary partition of the pipeline into contiguous segments $S_1, S_2, \ldots, S_k$ such that each segment has total size at least $2\mathcal{M}$, where $\mathcal{M}$ is the memory afforded to the (optimal) algorithm. Define $gm_i = \min_{e \in S_i} gain(e)$ to be the smallest gain edge within segment S_i. On processing X inputs (for large enough X), the optimal algorithm incurs $\Omega((X/B)\sum_{i=1}^{k} gm_i)$ cache misses. In other words, the optimal algorithm has to "pay for" the minimum-gain edge of any segment with size $2\mathcal{M}$. While our previous statement of the lower bound [3] is with respect to the discrete model, the proof does not leverage discreteness, and hence the bound applies to the continuous model as well.

Our upper bound (Theorem 4) states that the cache-oblivious algorithm need only "pay for" edges crossing certain partitions. It remains only to show that these edges also contribute to the lower bound. Specifically, a size-$t > M$ segment is subdivided by splitting it at the minimum-gain edge in its middle size-$t/3$ subsegment (Figure 1, Line 4). As long as $t/3 > 2\mathcal{M}$, the optimal algorithm must also pay for this edge. Setting $\mathcal{M} = M/6$ implies that all of these crossing edges must also be paid for by the optimal algorithm with $\mathcal{M}$ memory. In conclusion, the number of cache misses incurred by the cache-oblivious scheduler is at most a constant factor more than the optimal algorithms cache misses if the cache-oblivious scheduler has constant factor more memory. □

3. CONCLUSIONS

We have presented a cache-oblivious scheduling algorithm for streaming pipelines based on recursive partitioning of the pipeline. This algorithm is asymptotically optimal given constant factor memory augmentation under the continuous assumption — that is, when each module of the pipeline can consume and produce fractional items. Under the more realistic non-continuous assumption — when a module can only fire when it can consume integral number of items equal to its input rate — the problem of cache-oblivious scheduling remains open. In addition, the problem of cache-oblivious scheduling of streaming applications which can be described using directed acyclic graphs (instead of simple linear chains) also remains open.

Acknowledgements

This research was supported by NSF grants CCF-1218017, CCF-1150036, and CCF-1218188.

4. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] K. Agrawal, A. Benoit, and Y. Robert. Mapping linear workflows with computation/communication overlap. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 195–202, Washington, DC, USA, 2008. IEEE Computer Society.

[3] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo. Cache-conscious scheduling of streaming applications. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 236–245, June 2012.

[4] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[5] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation mapping stream programs onto multicore architectures. *ACM SIGPLAN Notices*, 46:357–368, 2011.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, January 2012.

[7] GNU Radio, 2001. Software, `gnuradio.org`.

[8] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. Concurrent VLSI Architecture Tech Report 122, Stanford University, Computer Systems Laboratory, March 2002.

[9] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. *ACM SIGPLAN Notices*, 38(7):103–112, 2003.

[10] LabVIEW, 2011. Software, `www.ni.com/labview`.

[11] Mathworks. *Simulink User's Guide*, 2011. Release 2011b.

[12] 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 *Proceeings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 300–305, 2008.

[13] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, 40(7):115–126, 2005.

[14] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 179–196, London, UK, 2002. Springer-Verlag.