Buffer-space Efficient and Deadlock-free Scheduling of Stream Applications on Multi-core Architectures

Jongsoo Park
Computer Systems Laboratory
Stanford University, California, USA
jongsoo@cva.stanford.edu

William J. Dally
Computer Systems Laboratory
Stanford University, California, USA
dally@cva.stanford.edu

ABSTRACT

We present a scheduling algorithm of stream programs for multi-core architectures called *team scheduling*. Compared to previous multi-core stream scheduling algorithms, team scheduling achieves 1) similar synchronization overhead, 2) coverage of a larger class of applications, 3) better control over buffer space, 4) deadlock-free feedback loops, and 5) lower latency. We compare team scheduling to the latest stream scheduling algorithm, SGMS, by evaluating 14 applications on a multi-core architecture with 16 cores. Team scheduling successfully targets applications that cannot be validly scheduled by SGMS due to excessive buffer requirement or deadlocks in feedback loops (e.g., GSM and W-CDMA). For applications that can be validly scheduled by SGMS, team scheduling shows on average 37% higher throughput within the same buffer space constraints.

Categories and Subject Descriptors

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

General Terms

Algorithms, Languages, Performance

Keywords

Compiler and Tools for Concurrent Programming, Stream Programming, Green Computing and Power-Efficient Architectures, Multi-core Architectures

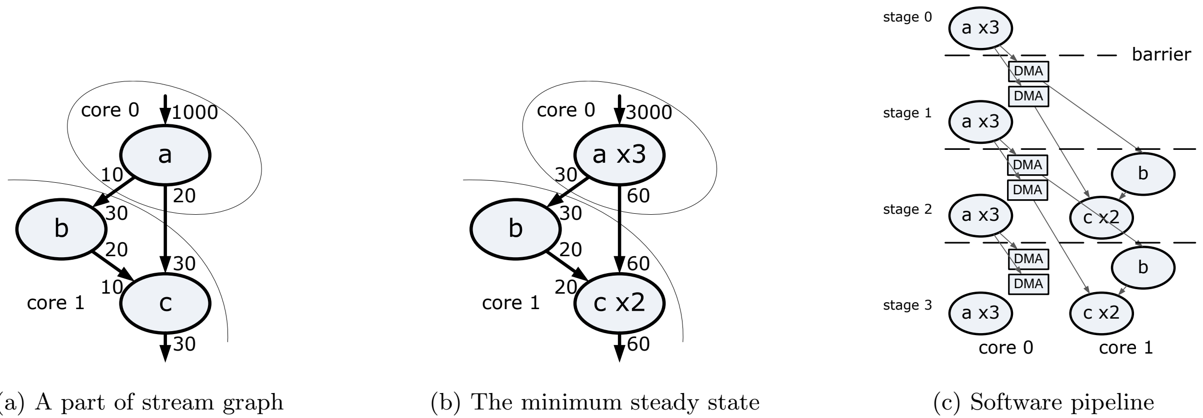
1. INTRODUCTION

In order to support the ever-increasing computation requirements of mobile devices while achieving a high level of energy efficiency, there has been growing interest in stream architectures [18, 14] and programming systems [35, 28, 9]. In a stream programming language, an application is abstracted as a *stream graph* [20] whose nodes are actors and whose edges are streams. Computation is described by actors that are fired when enough input stream tokens arrive (e.g., in Figure 1(a), actor b represents a computation that consumes 30 tokens and produces 20 tokens per firing). Unidirectional data flow between actors is described as streams, providing two benefits to the compiler. First, the compiler can easily detect parallelism by analyzing data dependences exposed as streams. Second, the compiler can efficiently orchestrate data movement by explicitly managing local memories and keep data coherent by maintaining queues, without relying on expensive hardware cache coherency protocols. By exploiting these benefits, stream compilers have shown a consistent speedup even for applications with low computation-to-communication ratios. For example, a StreamIt compiler shows an average of 14.8× speedup on a 16-core Cell [14] platform [20].

Static parts of stream programs, in which the number of tokens consumed and produced per actor firing are compile-time constants, follow the model of computation called *synchronous data flow* (SDF). SDF provides a theoretical background by which we can dramatically reduce synchronization overhead and buffer requirements. Lee and Messerschmitt [25] present an algorithm that constructs single-core static schedules with bounded buffer requirement and no synchronization overhead. Bhattacharyya et al. [4] present an algorithm that significantly reduces the buffer requirement of single-core static schedules. For multi-core architectures, [25, 24, 5, 31] present scheduling algorithms based on *homogeneous* SDF *graph* (HSDFG), a graph in which every actor consumes and produces only one token from each of its inputs and outputs [24]. However, constructing an HSDFG from an equivalent SDF graph can take an exponential amount of time [31], and their algorithms do not fully exploit pipeline parallelism [34]. These issues are resolved by Stream Graph Modulo Scheduling (SGMS) implemented in a StreamIt compiler [20].

SGMS applies software pipelining [22, 32] to the entire stream graph and synchronizes steady states of the pipeline with barriers. Consider a part of a stream graph shown in Figure 1(a). The partitioning phase that precedes scheduling has assigned actor a to core 0 and actors b and c to core 1. Numbers at each edge denote the number of stream tokens that are consumed or produced per actor firing. SGMS first finds the *minimum steady state* [16] in which the number of

This is the author's version of the work. Not for redistribution
SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.
Copyright 2010 ACM 978-1-4503-0079-7/10/06 ...$10.00.

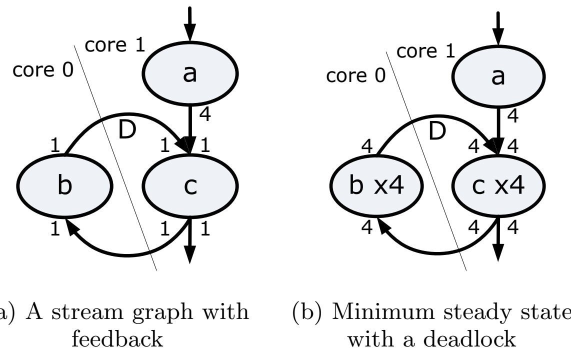


(a) A part of stream graph (b) The minimum steady state (c) Software pipeline

Figure 1: An example of Stream Graph Modulo Scheduling (SGMS). We assume that actor a is assigned to core 0 and actors b and c are assigned to core 1 in the partitioning phase that precedes scheduling. Numbers at each edge denote the number of stream tokens that are consumed or produced per actor firing. For example, actor b consumes 30 tokens and produces 20 tokens per firing. DMA denotes direct memory access.

produced tokens and consumed tokens are balanced at each edge with the minimum number of actor firings. For example, a must be fired three times per b's firing to produce 30 tokens required by b as shown in Figure 1(b). An efficient algorithm to find such a minimum steady state is described in [25]. After finding the steady state, SGMS constructs a software pipeline as shown in Figure 1(c). By starting execution of a producer actor and its consumer actor[1] at different *stages* [32] (a starts at stage 0 while its consumer, b, starts at stage 2), SGMS eliminates intra-stage dependencies so that processor cores do not need to synchronize with each other within a steady state. An actor periodically writes tokens to its output buffer, whose data is DMA-transferred at the next stage. Barriers between each stage guarantee that, whenever an actor fires, the input tokens required by the actor are already in place.

SGMS has the advantage of low synchronization overhead (one barrier per steady state), but has the following three drawbacks. First, SGMS requires information that may not be available at compile time. For example, the number of tokens to be produced can vary at run-time for certain streams. We call these *variable-rate streams* (e.g., the output of the Huffman encoder in JPEG). Second, SGMS has little control over buffer space; the minimum buffer space for each stream[2] is imposed by the minimum steady state. For example, in the steady state shown in Figure 1(b), we require buffer space that accommodates at least 3000 tokens at the incoming stream of actor a. We cannot reduce this buffer requirement because the minimum number of a firings between barriers is set to 3 by the steady state. If each core has a 2K-word local memory and the unit token size of a's incoming stream is 1 word, a remote memory must be accessed to further buffer the tokens. This leads to higher energy consumption and less predictable execution time, which makes guaranteeing load balance and real-time constraints at compile-time a challenge. Our evaluation and [27] show



(a) A stream graph with feedback (b) Minimum steady state with a deadlock

Figure 2: A deadlock in a feedback path caused by SGMS. (a) An example stream graph. "D" at edge (b, c) denotes a single initial token that makes the stream graph deadlock-free. (b) The minimum steady state used by SGMS in which c never fires.

that buffer space requirements can grow exponentially in the minimum steady state of real-life applications such as W-CDMA. Third, SGMS does not handle feedback loops satisfactorily. In [20], the authors mention that a feedback loop is naïvely handled by fusing the entire loop into a single actor, which results in complete serialization of the loop. If we do not fuse feedback loops into single actors to avoid serialization, SGMS is prone to deadlock. Consider a feedback path $c \rightarrow b \rightarrow c$ shown in Figure 2(a). This feedback path is deadlock-free due to an initial token at edge (b, c) denoted as D. The value of the initial token is specified by the programmer and adds a unit delay at (b, c), thus the use of the symbol 'D' commonly found in signal processing [29]. However, in the steady state shown in Figure 2(b), actor c cannot be fired because it never receives enough input tokens. The compiler cannot create additional initial tokens because doing so changes the semantic of the application. Our evaluation (Section 3) shows that a similar deadlock occurs in a real-life application, GSM.

In this paper, we present an alternative algorithm called *team scheduling* that addresses the drawbacks of SGMS, while maintaining a similar synchronization overhead. Team schedul-

[1] More specifically, its consumer actor at a different core since SGMS starts producer and consumer actors at the same stage if they are assigned to the same core.

[2] More specifically inter-core stream: the buffer requirement for intra-core streams depends on how to schedule actors assigned to a single-core, which is described in [4, 16].

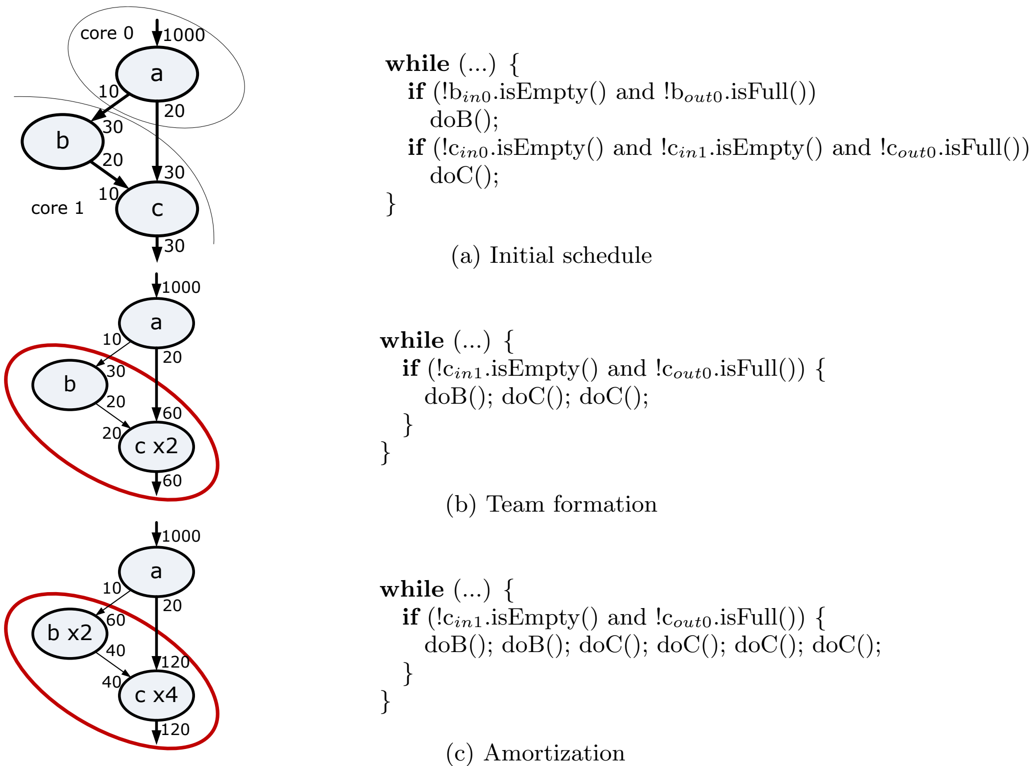


Figure 3: An example team scheduling and its generated code for core 1. (a) An initial schedule (b) Form team $\{b, c\}$ and construct its static schedule, which eliminates synchronization between b and c. Section 2.1 describes why synchronization between a and b can also be eliminated. (c) Amortize the team $\{b, c\}$ by a factor of 2.

ing starts with a simple initial schedule as shown in Figure 3(a). Actor firings are pair-wise synchronized through queue empty and full checks. This initial schedule involves high synchronization overhead (i.e., frequent queue empty and full checks). Nevertheless, this is a correct schedule for a wide range of applications including the ones that cannot be validly scheduled by SGMS. Moreover, the synchronization overhead can be minimized with aggregation and amortization of actors as follows. We assume that the partitioning phase precedes scheduling similar to SGMS, where actor-to-core mapping is predetermined when reaching the scheduling phase.

First, we selectively aggregate actors that are assigned to the same core, and form a *team* in which actors are statically scheduled. By statically scheduling actors in a team, we eliminate intra-team synchronizations. For example, in Figure 3(b), we form a team by aggregating actors b and c, and eliminate synchronization between them (b_{out0}.isFull() and c_{in0}.isEmpty() checks are removed). We can also eliminate inter-team synchronization such as the one between a and b (explained in Section 2.1). In order to construct a static schedule of team $\{b, c\}$, we find its steady state — fire b once and c twice. This is in contrast to SGMS, which must use a steady state of the *entire stream graph*; in team scheduling, the unit of steady state construction is a team whose formation is under the compiler's control. We continue team formation as long as it does not violate constraints such as maximum buffer space per core.

Second, we selectively amortize communication overhead of teams by increasing the number of actor firings per synchronization. Each amortized actor accumulates its output tokens in its local buffer and transfers the accumulated tokens in bulk to the consumer's local memory. For example, in Figure 3(c), we amortize team $\{b, c\}$ so that its actors fire twice as often as they do in the minimum steady state of the team. Actor c accumulates 120 tokens in its local buffer and transfers them at once. Amortizing communication overhead is an important optimization scheme, especially for actors with a low computation-to-communication ratio: the locality of memory access is improved, and the fixed cost associated with each data transfer initiation is amortized. In Section 3, we show up to 2.1× speedup from amortization. The same optimization can be done in SGMS but with limited flexibility: minimum amortization factors are predetermined by the minimum steady state; and if we want to amortize an actor by 2, we must amortize all the other actors by 2 as well. Note that the flexibility in choosing amortization factor is crucial for finding the right trade-off between synchronization overhead and buffer requirement. For example, in Figure 3(c), team scheduling is able to selectively amortize b and c without excessively increasing the buffer requirement. On the other hand, SGMS incurs a large buffer requirement increase in order to amortize b and c because it must amortize a as well. As we do in team formation, we continue amortization as long as it does not violate constraints such as maximum buffer space per core.

The contributions of this paper are as follows: 1) We present an algorithm for scheduling stream programs on multi-core architectures that has better control over buffer space, lower latency, and better support for variable-rate streams than SGMS. 2) We present a method to compute minimum queue lengths to avoid deadlock or serialization that can be introduced by team scheduling. 3) We evaluate team scheduling with 14 stream applications on ELM [1], a multi-core architecture for energy-efficient embedded computing. Our evaluation shows that team scheduling achieves a similar throughput to that of SGMS with lower latency and smaller buffer requirement. Our evaluation also shows that team scheduling has better control over buffer space: when we set maximum buffer space per core as a constraint, team scheduling consistently satisfies the constraint while SGMS does not.

The remainder of this paper is organized as follows: Section 2 describes the details of team scheduling. Section 3 presents simulation results comparing team scheduling with SGMS. Section 4 reviews related work and Section 5 summarizes our work.

2. TEAM SCHEDULING

This section describes details of the team scheduling algorithm. Pseudo-code for the algorithm is presented in Appendix E.

2.1 Team Formation

We start from an initial schedule in which each actor forms a separate team. For example, in Figure 3(a), actors a, b, and c each form teams on their own. In a pair-wise manner, we merge teams in the same core starting with the pair that leads to the highest gain. We compute the gain as synchronization reduction divided by additional buffer requirement resulting from team merge. This is a greedy heuristic chosen to maximize synchronization overhead reduction (i.e., the reduction of queue empty or full checks) per additional buffer space requirement. We maintain a *team graph* that represents the connectivity of teams. The team graph is initially identical to the stream graph, and then we contract the corresponding nodes for each team merge.

We adhere to the following four constraints when merging. First, we do not merge across variable-rate streams. Second, we do not merge teams if doing so exceeds the buffer limit per core. Suppose that actors a, b, and c are all assigned to the same core in Figure 3. If each core has a 2K-word local memory and the unit token size of a's incoming stream is 1 word, we avoid merging a with any other actor. A detailed method of computing buffer sizes is described in Section 2.3. Third, we must not introduce a cycle to the team graph since it may result in deadlocks. Suppose again that actors a, b, and c are all assigned to the same core in Figure 3. We avoid merging a with c because it forms a cycle $b \rightarrow \{a, c\} \rightarrow b$. Fourth, a merge must not introduce any deadlock in an existing cycle (i.e., in a feedback loop). We can check for such deadlocks by inspecting *precedence expansion graphs* (PEG) [25] of each cycle containing the merged team in the team graph. If every PEG is acyclic, it is guaranteed that the team merge does not introduce any deadlock. This is because a PEG has a cycle if and only if the corresponding stream subgraph has a deadlock [25]. Suppose that we are about to merge a and c in Figure 2(a). If we construct a PEG of the cycle $b \rightarrow c \rightarrow b$ after the merge, we see a cycle in the PEG since merging a and c introduces a deadlock. For the details of PEG construction, refer to [25, 24, 31]. A PEG can grow exponentially when the number of actor firings in the minimum steady state of a team or the number of cycles in the stream graph is exponential. In this case, we use a heuristic described in [31] that conservatively but quickly checks for deadlocks.

After merging a team, we construct a static schedule of the actors within the team. There are several ways of constructing such a single-core schedule [4, 16], but we find that *loose interdependence scheduling framework* (LISF) [3] works well in our evaluation (Section 3). For most applications, LISF finds a *single appearance schedule* in which each actor lexically appears only once, resulting in a minimal code size [3]. For example, b $2c$ (fire b once, then fire c twice) is a single appearance schedule of team $\{b, c\}$ shown in Figure 3(b). Other single-core scheduling methods such as *push schedule* or *phased schedule* save significant buffer space at the expense of marginal increase in code size when applied to the entire stream graph [16]. However, when we target multi-cores, applications are partitioned into small pieces, and applying either scheduling method to each piece shows little buffer space saving (on average 8% for 16 cores).

By constructing a static schedule of a team, we eliminate intra-team synchronizations such as the one at edge (b, c) in Figure 3(b). We can also eliminate certain inter-team synchronizations such as the one at edge (a, b). This is possible because the production-to-consumption ratios of streams between a given team pair (with no variable-rate streams) are constant (proof shown in Appendix A). For example, at $(a, \{b, c\})$ in Figure 3(b), production-to-consumption ratios are $\frac{10}{30} = \frac{20}{60}$. To generalize, consider the streams from team T to team U denoted as S_{TU} (in Figure 3(b), $S_{TU} = \{(a, b), (a, c)\}$ when $T = \{a\}$ and $U = \{b, c\}$). Let s_1 be the stream in S_{TU} that is enqueued last in T's static schedule (in Figure 3(b), s_1 is (a, c) if T enqueues tokens to (a, b) before (a, c)). Let s_2 be the stream in S_{TU} that is dequeued last in U's static schedule (in Figure 3(b), s_2 is (a, c)). Then among the conditions with respect to S_{TU} that we need to check before firing U, we can eliminate everything except the check for whether s_1 is not empty (in Figure 3(b), checking whether the queue at (a, b) is not empty is redundant when s_1 is (a, c)). Similarly, among the conditions with respect to S_{TU} that we need to check before firing T, we can eliminate everything except the check for whether s_2 is not full. More details of inter-team synchronization elimination are described in Appendix B.

2.2 Amortization

After team formation, we amortize communication cost of teams starting from the one that leads to the highest synchronization reduction per additional buffer requirement. As in the team formation procedure, we do not amortize a team if doing so exceeds buffer space limit or introduces deadlock in a feedback path.

We define amortization as follows: The *minimum repetition vector* [25] $\overrightarrow{q_G}$ of stream subgraph G is a vector such that $\overrightarrow{q_G}(a)$ is the number of a firings in the minimum steady state of G. For example, the minimum repetition vector of the stream graph shown in Figure 1(a) is $(3, 1, 2)$ where we index the vector in the order of a, b, and c; the minimum repetition vector of team $\{b, c\}$ in Figure 3(b) is $(1, 2)$. For each stream subgraph G that is statically scheduled (the en-

tire stream graph in SGMS or a team in team scheduling), we define the *repetition vector* $\overrightarrow{r_G}$ such that $\overrightarrow{r_G}(a)$ is the number of a firings in the current static schedule of G. We call $\overrightarrow{r_G}(a)$ the *repetition* of actor a. For example, the repetition vector of team $\{b, c\}$ in Figure 3(b) is the same as its minimum repetition vector, $(1, 2)$, because the team has not been amortized. In Figure 3(c), the repetition vector of team $\{b, c\}$ is $(2, 4)$. In this paper, *amortization* of stream subgraph G by a factor of k means multiplying G's repetition vector by k. For example, in Figure 3(c), amortization of team $\{b, c\}$ by a factor of 2 has updated its repetition vector from $(1, 2)$ to $(2, 4)$.

Note that in SGMS the repetition vector is identical to the minimum repetition vector of the stream graph before any amortization. If we amortize a schedule by a factor of 2, we multiply the repetition of *every* actor by 2. In team scheduling, each team has its own repetition vector, and each team is amortized separately.

We use the following method of selecting amortization factors: Suppose that we are about to amortize team T in stream graph G. If $\exists$ an integer $k > 1$ such that $\forall a \in T$, $\overrightarrow{q_G}(a) = k \cdot \overrightarrow{r_T}(a)$, we amortize T by the smallest integer bigger than 1 that divides k. For example, for team $\{a\}$ in Figure 3(b), $\overrightarrow{q_G}(a) = 3$ and $\overrightarrow{r_{\{a\}}}(a) = 1$, thus $k = 3$. Otherwise, we amortize T by a factor of 2. We use this method in order to first amortize T up to the minimum steady state of the entire graph and to additionally amortize T by a factor of 2 thereafter.

2.3 Sizing Queues to Avoid Deadlocks

When each team is amortized separately, deadlock or serialization due to insufficient queue capacity can occur as shown in Figure 4. In Figure 4(a), after firing a 6 times, the queue at (a, c) is full and "$b \times 3$" does not have enough input tokens to be fired, resulting in a deadlock (assume that each actor is assigned to different cores). Note that this is a different kind of deadlock from the ones that occur in the feedback loops shown in Figure 2(b). In Figure 2(b), deadlock is inherent in the stream graph: we cannot avoid deadlock no matter how large a queue we use for each stream. To avoid the deadlock shown in Figure 4(a), we need to increase the length of queue at (a, c) to 180. However, this still is not large enough to support serialization-free execution during the latency along path $a \to b \to c$ as shown in Figure 4(b). To avoid such serialization, the queue length must be at least 400. This section presents a method that computes the minimum queue length needed to avoid deadlock and serialization.

Before team merge and amortization, we first determine the queue lengths of streams along feedback loops. We can bound the queue length of a feedback stream s as (see [5])

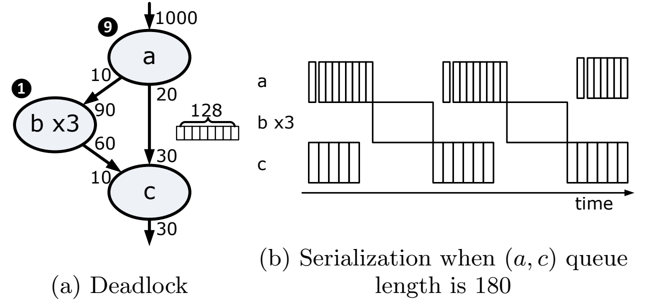
$$\min_{\text{cycle } C \text{ containing } s} (\text{sum of delays along } C).$$

When we compute queue lengths of other streams, we use the *acyclic team graph* which is constructed by removing a stream with non-zero delay from each cycle in the team graph.

After merging or amortizing a team, for each stream s that is adjacent to the team, we set the queue length of s to

$$2(prod(s) + cons(s) - gcd(prod(s), cons(s))),$$

where $prod(s)$ is the number of tokens produced per s's producer firing and $cons(s)$ is the number of tokens consumed



(a) Deadlock

(b) Serialization when (a, c) queue length is 180

Figure 4: An example of deadlock and serialization from insufficient queue capacity. In (a), b is amortized by a factor of 3. ❾ denotes that actor a must be fired at least 9 times to provide enough input tokens for c firing. Assume that the queue at (a, c) can buffer 128 tokens. A deadlock occurs after firing a 6 times. (b) shows a steady state execution when the queue length of (a, c) is 180.

per s's consumer firing. Appendix C shows that this prevents serialization between a producer and consumer pair.

After sizing queues only based on the information associated with their producers and consumers, we consider global information. First, we find split-join patterns that contain a team that has been merged or amortized. A team is a *splitter* if it has multiple successors, while a team is a *joiner* if it has multiple predecessors. We define the *split-join pattern* of S and J, G_{SJ}, as the teams that are reachable from S and reachable to J. In Figure 4(a), assume that a, b, and c each form teams of their own, then $\{b\}$ belongs to $G_{\{a\}\{c\}}$. Second, we compute $x_J(U)$ for each $U \in G_{SJ}$, the minimum number of U firings to fire J. In Figure 4(a), $x_{\{c\}}(\{a\}) = 9$. This can be computed by an algorithm similar to backward data-flow analysis [15] whose details are shown in Appendix D. Third, we find the longest latency path from S to J with the latency defined as follows: Let $q(T)$ be the minimum repetition of T that can be computed by $\frac{\vec{q}(d)}{\overrightarrow{r_T}(d)}$ for any $d \in T$. In Figure 4(a), $q(\{b\}) = \frac{1}{3}$. Let $l(T, U) = \frac{x_J(T)}{q(T)}$ be the latency of edge (T, U), which represents the latency introduced at (T, U), normalized to the period of repeating the minimum steady state of the application. In Figure 4(a), $l(\{a\}, \{b\}) = \frac{9}{3}$, $l(\{b\}, \{c\}) = \frac{1}{1/3}$, and $l(\{a\}, \{c\}) = \frac{9}{3}$. Therefore, the longest latency path is $\{a\} \to \{b\} \to \{c\}$ with normalized latency 6, which means that buffers along each path from $\{a\}$ to $\{c\}$ should support serialization-free execution during the latency equivalent to the time for repeating the minimum steady state 6 times. Denote the number of S firings during the longest latency as $y_{SU} = \lceil q(S) \cdot (\text{longest latency from } S \text{ to } J) \rceil$. In Figure 4(a), $y_{\{a\}\{c\}} = 3 \cdot 6 = 18$. Fourth, we simulate firing actors in the split-join pattern until S is fired y_{SU} times. This can be done by an algorithm similar to forward data-flow analysis. In the simulation, we do not fire J and we set queue lengths of J's incoming streams to infinity. Let $z_{SJ}(s)$ be the number of tokens in J's incoming stream s after the simulation, which is the minimum number of tokens to start a steady state with respect to s's producer and J, while supporting serialization-free execution during the longest latency from S to J. During the steady state, the buffer at s requires $prod(s) + cons(s) - gcd(prod(s), cons(d))$ additional space

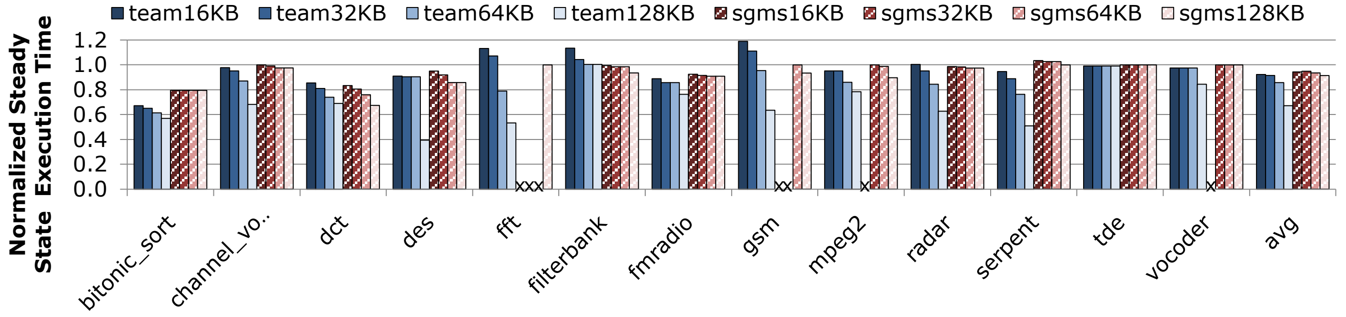


Figure 5: Steady state execution time when team scheduling and SGMS amortize actors within different buffer space constraints. Execution times are normalized to those of SGMS without any amortization.

(Appendix C). Therefore, we increase the queue length of s to $z_{SJ}(s) + prod(s) + cons(s) - gcd(prod(s), cons(d))$. In Figure 4(a), $\{a\}$ is fired 18 times during the simulation and leaves 360 tokens at (a, c) $(z_{\{a\}\{c\}}((a, c)) = 360)$. We increase the queue length of (a, c) to 360 + 20 + 30 - 10 = 400.

3. RESULTS

This section describes the experimental setup for our algorithm evaluation and the analysis of results.

3.1 Experimental Setup

We use the same set of StreamIt benchmark applications that were used for SGMS evaluation [20] plus GSM encoder and W-CDMA searcher [36]. We have ported the StreamIt applications into a language called Elk [8] that extends StreamIt. Elk supports multiple input/output streams and variable-rate streams, though language difference is not essential in our evaluation. W-CDMA searcher is ported from a proprietary benchmark from Qualcomm^{TM}. GSM encoder is ported from MiBench [12] and contains a feedback path.

We use ELM [1], a multi-core architecture for energy-efficient embedded computing. ELM has tiled multi-cores connected through an on-chip interconnection network [7]. Each core executes instructions with MIPS-like ISA in a 4-stage in-order dual-issue pipeline. ELM supports DMA-like stream memory instructions that transfer a block of data to other cores' local memory in the background, and these stream memory instructions are used to implement queue operations. ELM has an ensemble organization in which four cores share their local memory. We made each core have its own separate local memory, changed the local memory size to 256KB, and used 16 cores to make the evaluation setup similar to that of the SGMS paper [20] which uses Cell processors.

The Elk compiler generates C++ code from Elk code, and an LLVM-based [23] C++ compiler [30] generates ELM assembly code. The assembly code is executed in a cycle-accurate ELM simulator. We model interconnection as a mesh network with word-wide channels and canonical 4-stage pipeline routers [7]. Therefore, the latency of a message is $4(d + 1)$ cycles if the Manhattan distance to the destination is d. For SGMS, we idealistically assume that every core can access a dedicated memory in 1 Manhattan distance latency (8 cycles), and we implement a sense-reversing barrier [13] using fetch-and-add instructions on the dedicated memory. This results in 75 cycles per barrier while each barrier takes 1600 cycles in the SGMS paper [20].

As in previous work [11, 10, 20], partitioning is done before scheduling. We first fission stateless actors with high computation requirement so that every stateless actor has at most 1/16 of the total computation requirement. Then we assign actors to cores using METIS, a graph partitioning package [17].

In the first experiment, we compare the throughput of team scheduling with that of SGMS as we change the buffer space limit per core from 16KB to 128KB. This experiment measures the efficiency of using limited local memory space, which is critical for multi-core embedded processors. In the second experiment, we intentionally avoid exploiting the amortization flexibility of team scheduling by limiting the maximum repetitions to the ones in the minimum steady state of the entire stream graph, and compare throughput, latency, and buffer usage of the two algorithms. This experiment compares performance of the two scheduling algorithms independent of amortization effects.

3.2 Buffer Space Limited Experiment

Figure 5 compares throughput of both algorithms as we change the buffer space limit per core from 16KB to 128KB. Steady state execution time is measured as the time between the generation of the first and the last output of the furthest downstream actor, and is inversely proportional to throughput. Steady state execution times are normalized to those of SGMS without any amortization. The average speedup from single-core executions is 11×. The results for WCDMA are not shown here because SGMS requires an excessive buffer space even without amortization, which will be shown in Section 3.3. We fuse feedback loops in GSM to single actors for SGMS since SGMS results in a deadlock similar to that shown in Figure 2(b) without fusion. For the particular case of GSM, SGMS does not show its disadvantage with respect to complete serialization of feedback loops since the feedback loops in GSM do not have parallelism that can only be exploited by team scheduling.

SGMS does not satisfy the buffer space constraint for `fft`, `gsm`, `mpeg2`, and `vocoder` when the space constraint is as small as 16KB. In contrast, team scheduling satisfies buffer space constraints across all configurations. The averages are computed only on the applications that satisfy the buffer space constraint with both scheduling algorithms (e.g., the averages for `team16KB` and `sgms16KB` are computed excluding `fft`, `gsm`, `mpeg2`, and `vocoder`). When the buffer space

limit is as large as 128KB, team scheduling achieves an average of 37% higher throughput than that of SGMS, which is especially apparent in `fft`, `fmradio`, and `serpent`. We can see the importance of amortization from its up to 2.1× speedup (`serpent` at `team128KB`).

As mentioned in Section 2.1, there are several ways to schedule actors that belong to a team (in team scheduling) or that are assigned to the same core and the same software pipeline stage (in SGMS). However, push schedule — the most buffer space efficient single-core schedule [16] — saves only an average of 8% of buffer space compared to single appearance schedule. In addition, push schedule does not make SGMS meet buffer space constraints of any application which does not already satisfy the constraint in single appearance schedule. If we apply push schedule to the entire stream graph, we achieve a significant reduction in buffer space requirement compared to single appearance schedule as shown in [16]. However, when we target 16 cores, the application is already partitioned into 16 pieces, and there is little difference between using either scheduling method on each small piece.

In this experiment, we show that team scheduling has better control over buffer space than SGMS has: given buffer space constraint, team scheduling has a better chance of satisfying the constraint and achieves higher throughput by efficiently utilizing the limited buffer space for amortization (i.e., team scheduling achieves balanced trade-off between synchronization overhead and buffer space requirement).

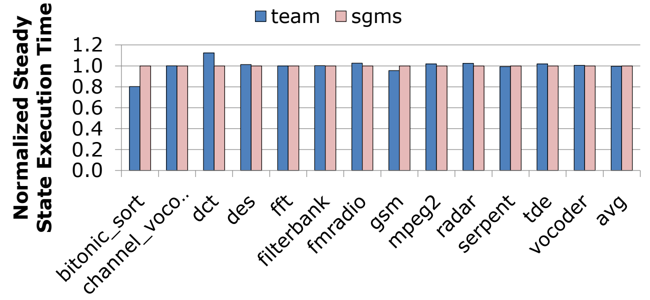
3.3 Amortization Factor Limited Experiment

Figure 6 shows throughput, latency, and buffer requirement of both algorithms while we limit repetition factors to those in the minimum steady state of the entire stream graph. In this experiment, we set the buffer space limit per core to 64KB for team scheduling, which achieves a similar (0.4% higher) throughput to that of SGMS as shown in Figure 6(a) — a larger buffer space limit improves throughput at the expense of longer latency. In Figure 6(b), latency is measured as the time until the first output of the furthest downstream actor is generated. Team scheduling shows 65% lower latency and 46% smaller buffer requirement when its throughput is similar to that of SGMS. SGMS has high latency because of poor load balancing in its software pipeline prologue, resulting in idle cycles while the processor waits for barriers. Team scheduling does not suffer from this problem since actors are pair-wise synchronized and can be fired whenever input tokens are ready.

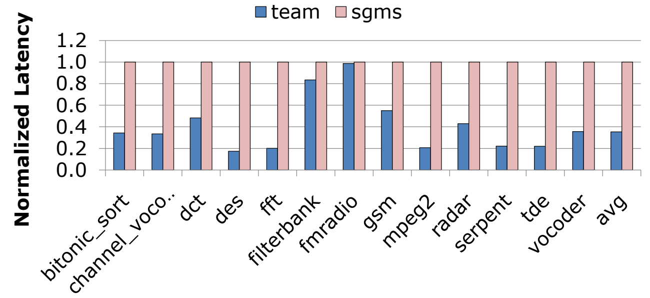
Since we set the buffer space limit to 64KB for team scheduling, team scheduling uses less than 64KB for every application as shown in Figure 6(c). SGMS requires 2MB buffer space for `wcdma`, which is well over the local memory size of each core, 256KB. Hence, we omit `wcdma` in Figure 6(a) and (b). In `wcdma`, there is a series of reduction actors that produce fewer tokens than it consumes, thus actors upstream must be executed hundreds of times, consuming hundreds of KB of data in steady state. Team scheduling avoids excessive buffer requirement from the upstream actors by decoupling the scheduling of upstream and downstream actors.

4. RELATED WORK

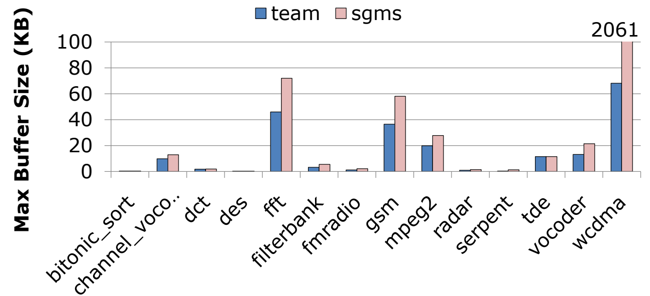
Lee and Messerschmitt [25] lay the foundation of SDF including a necessary and sufficient condition to the existence of a valid static schedule which does not deadlock and requires bounded buffer space. Bhattacharyya et al. [4] present an SDF scheduling algorithm for single-core architectures that reduces the buffer size of single appearance schedules. They use *pairwise grouping of adjacent nodes* (PGAN) heuristic, in which some aspects are similar to our team formation procedure, but in the context of single-core scheduling. In [5], Bhattacharyya et al. present a post-pass optimization scheme that eliminates redundant synchronization in an existing multi-core SDF schedule. Their method eliminates the same set of redundant inter-team synchronizations (e.g., synchronization at (a, b) in Figure 3(b)) as team scheduling, but team scheduling does so without running a sophisticated analysis by exploiting a property of team scheduling, namely constant production-to-consumption ratios of streams between a team pair. They also present a method of reducing synchronization overhead by introducing a few



(a) Steady state execution time normalized to SGMS



(b) Latency normalized to SGMS



(c) Buffer requirement

Figure 6: Results of amortization factor limited experiment with team scheduling and SGMS for 16 cores.

feedback paths that make full checks of other queues unnecessary. However, this method requires additional buffer space, and it will be interesting to evaluate whether using the buffer space for eliminating queue full checks as in [5] is more beneficial than using the same buffer space for amortization as in team scheduling.

The body of SDF work [25, 24, 4, 5, 34, 31] provides a solid theoretical background for optimizing static parts of stream programs. However, their multi-core scheduling algorithms [25, 24, 5, 31] are based on *homogeneous* SDF *graph* (HSDFG), whose construction from an equivalent SDF graph can take an exponential amount of time [31]. In addition, their algorithms do not overlap the execution of different HSDFG iterations, resulting in a smaller degree of parallelism.

[33, 21, 19] present SDF *vectorization* that amortizes the cost associated with actor interactions. For example, Ko et al. [19] develop a vectorization algorithm that reduces context switch overhead and increases memory locality within memory constraints, which is similar to amortization described in this paper in some aspects but is done in the context of single-core scheduling. Amortization, or vectorization, plays a more important role in multi-core scheduling because it not only improves the locality of memory access but also minimizes fixed costs associated with each DMA initiation.

Lin et al. [27] point out exponential buffer space growth in their W-CDMA evaluation and present an algorithm that applies software pipelining in a hierarchical manner. However, in their algorithm, the programmer must define the hierarchy, and the authors failed to keep the scheduling algorithm free from exponential buffer space growth when they designed their later work, SGMS [20].

Gordon et al. [11] point out deadlocks in split-join patterns. However, their deadlock resolving method targets a subset of what is described in this paper and does not handle the case shown in Figure 4(a).

5. CONCLUSION

Previous scheduling algorithms such as SGMS make several assumptions on target applications, e.g., the entire application should follow synchronous data flow (SDF) model. On the contrary, team scheduling starts from an initial schedule that makes minimal assumptions, thus targeting a larger class of applications. Team scheduling successively refines the initial schedule by aggregation and amortization of actors, achieving low synchronization overhead similar to that of SGMS. In addition, team scheduling realizes key performance features such as deadlock-free feedback loops, low latency, and flexible buffer space control since it is less constrained by the minimum steady state of the entire application. Due to its flexibility in buffer space control, team scheduling efficiently utilizes limited local memory space of each core. This is clearly shown by the fact that team scheduling consistently satisfies the buffer space constraint whereas SGMS fails to do so when the space limit per core is small (no larger than 16KB). In the case where the space limit is as large as 128KB, team scheduling achieves on average 37% higher throughput than SGMS. These results demonstrate team scheduling as a critical optimization scheme in stream compilers for a large class of applications targeting embedded multi-core processors, which commonly have limited local memory space.

6. ACKNOWLEDGEMENTS

We thank Aaron Lamb and his colleagues at Qualcomm™ for providing their W-CDMA searcher implementation. We thank James Balfour for the ELM architecture simulator implementation, Ji Young Park for fruitful discussions on our algorithm, and Jooseong Kim for the Elk frontend implementation. We thank the anonymous reviewers who provided valuable feedback. This work is supported in part by the Semiconductor Research Corporation under Grant 2007-HJ-1591 and in part by the National Science Foundation under Grant CNS-0719844. Jongsoo Park is supported by a Samsung Scholarship.

7. REFERENCES

[1] J. Balfour, W. J. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An Energy-Efficient Processor Architecture for Embedded Systems. *Computer Architecture Letters*, 7(1):29–32, 2008.

[2] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[3] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms. *IEEE Transactions on Circuits and Systems*, 42(3):138–150, 1995.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations. *Design Automation for Embedded Systems*, 2(1):33–60, 1997.

[5] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Optimizing Synchronization in Multiprocessor DSP Systems. *IEEE Transactions on Signal Processing*, 45(6):1605–1618, 1997.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 13(4):451–490, 1991.

[7] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[8] ELM Webpage. Concurrent VLSI Architecture Group, Stanford University. http://cva.stanford.edu/projects/elm/software.htm.

[9] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Conference on Supercomputing (SC)*, 2006.

[10] M. I. Gordon, W. Thies, and S. P. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[11] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffman, D. Maze, and S. P. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architecture Support for Programming Language and Operating Systems (ASPLOS)*, pages 291–303, 2002.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE Annual Workshop on Workload Characterization*, pages 83–94, 2001.

[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[14] H. P. Hofstee. Power Efficient Processor Architecture and the CELL Processor. In *International Symposium on High-Performance Computer Architectures (HPCA)*, pages 258–262, 2005.

[15] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[16] M. Karczmarek, W. Thies, and S. P. Amarasinghe. Phased Scheduling of Stream Programs. In *Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 103–112, 2003.

[17] G. Karypis and V. Kumar. METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science, University of Minnesota, 1995.

[18] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, 2001.

[19] M.-Y. Ko, C.-C. Shen, and S. S. Bhattacharyya. Memory-constrained Block Processing for DSP Software Optimization. *Journal of Signal Processing Systems*, 50(2):163–177, 2008.

[20] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, 2008.

[21] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak. Optimizing Computations for Effective Block-Processing. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):604–630, 2000.

[22] M. Lam. Software Pipelining: An Effective Scheduling Technique on VLIW Machines. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[24] E. A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*. PhD thesis, University of California, Berkeley, 1986.

[25] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[26] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in Flowgraph. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 1(1):121–141, 1979.

[27] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge. Hierarchical Coarse-grained Stream Compilation for Software Defined Radio. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 115–124, 2007.

[28] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2002.

[29] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab. *Signals & Systems*. Prentice Hall, 1997.

[30] J. Park, J. Balfour, and W. J. Dally. Maximizing the Filter Rate of L0 Compiler-Managed Instruction Stores by Pinning. Technical Report 126, Concurrent VLSI Architecture Group, Stanford University, 2009.

[31] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A Hierarchical Multiprocessor Scheduling Systems for DSP Applications. Technical Report UCB/ERL M95/36, University of California, Berkeley, 1995.

[32] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *International Symposium on Microarchitecture (MICRO)*, pages 63–74, 1994.

[33] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr. Optimum Vectorization of Scalable Synchronous Dataflow Graphs. In *International Conference on Application-Specific Array Processors (ASAP)*, pages 285–296, 1993.

[34] S. Sriram and S. S. Bhattacharyya. *Embedded Mutiprocesors: Scheduling and Synchronization*. CRC, 2009.

[35] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction (CC)*, pages 179–196, 2002.

[36] Y.-P. E. Wang and T. Ottosson. Cell Search in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1470–1482, 2000.

APPENDIX

A. Constant Production-to-Consumption Ratios between a Team Pair

Let $\overrightarrow{q_G}$ be the minimum repetition vector of stream subgraph G. For teams T and U in stream graph G, according to [4]:

$$\exists \text{ an integer } m \text{ such that } \forall a \in T,\ \overrightarrow{q_G}(a) = m \cdot \overrightarrow{q_T}(a) \quad (1)$$

$$\exists \text{ an integer } n \text{ such that } \forall a \in U,\ \overrightarrow{q_G}(a) = n \cdot \overrightarrow{q_U}(a) \quad (2)$$

For stream s, let $src(s)$ be s's producer and $dst(s)$ be s's consumer. Since the number of tokens produced and consumed at a stream are equal in the minimum steady state of G, $\overrightarrow{q_G}(src(s)) \cdot prod(s) = \overrightarrow{q_G}(dst(s)) \cdot cons(s)$, which is called the *balanced equation* [4]. Substituting Equation (1) and (2) into the balanced equation shows that, for each stream s from T to U, $m \cdot \overrightarrow{q_T}(src(s)) \cdot prod(s) = n \cdot \overrightarrow{q_U}(src(s)) \cdot cons(s)$. This means that the ratio of the number of tokens produced at s by each T firing ($\overrightarrow{q_T}(src(s)) \cdot prod(s)$) to the number of tokens consumed from s by each U firing ($\overrightarrow{q_U}(dst(s)) \cdot cons(s)$) is a constant, $\frac{n}{m}$.

B. Inter-team Synchronization Elimination

Let S_{TU} be the streams from team T to team U. Let s_1 be the stream in S_{TU} that is enqueued last in T's static sched-

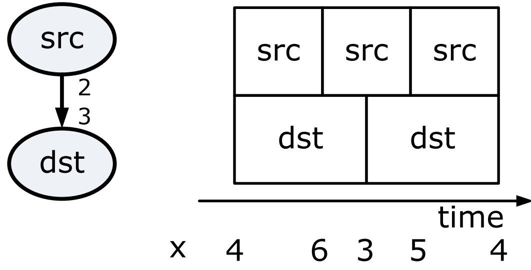


Figure 7: A steady state execution when $p = 2$, $c = 3$, and $x(0) = 4$.

ule. Consider the situation when we check conditions to fire U. Assume that we have checked that s_1 is not empty. This implies that all the other streams in S_{TU} are also not empty, which makes checking if any of those streams is empty unnecessary. This can be shown through contradiction as follows. Suppose that the queue at $s_3 \in S_{TU} - \{s_1\}$ is empty. Then due to the constant production-to-consumption ratios of S_{TU}, the queue at s_1 must be empty which contradicts our assumption.

Let s_2 be the stream in S_{TU} that is dequeued last in U's static schedule. Suppose that the queue lengths of S_{TU} are proportional to their respective number of tokens produced by each T firing. Consider the situation when we check conditions to fire T. Similarly, we can show that, if s_2 is not full, all the other streams in S_{TU} must not be full as well.

C. Serialization-free Queue Length for Producer-consumer Pairs

Consider a stream s. Let p be the number of tokens produced per $src(s)$ firing. Let c be the number of tokens consumed per $dst(s)$ firing. Assume that $src(s)$ is fired every p time steps and $dst(s)$ is fired every c time steps (perfect load balance). Suppose that s has $x(0)$ tokens at time step 0, when a steady state with respect to $src(s)$ and $dst(s)$ begins. At time step t, the number of tokens at s is

$$
\begin{aligned}
x(t) &= x(0) + \lfloor \frac{t}{p} \rfloor \cdot p - \lfloor \frac{t}{c} \rfloor \cdot c \\
&= x(0) + (t \ mod \ c) - (t \ mod \ p) \\
&\quad (\because t = \lfloor \frac{t}{p} \rfloor \cdot p + (t \ mod \ p) = \lfloor \frac{t}{c} \rfloor \cdot c + (t \ mod \ c))
\end{aligned}
$$

Let l be the queue length. When $src(s)$ finishes, $t \ mod \ p = 0$, thus $x(t) = x(0) + (t \ mod \ c) \leq x(0) + c - gcd(p, c)$. Since we need p space to fire $src(s)$ immediately to avoid stalls, $l \geq x(0)+p+c-gcd(p, c)$. When $dst(s)$ finishes, $t \ mod \ c = 0$, thus $x(t) = x(0) - (t \ mod \ p) \geq x(0) - p + gcd(p, c)$. Since we need c remaining tokens to fire $dst(s)$ immediately to avoid stalls, $x(0) - p + gcd(p, c) \geq c$. Therefore, the minimum l that avoids stalls is $2(p + c - gcd(p, c))$, which is achieved by $x(0) = p + c - gcd(p, c)$. Figure 7 shows an example where $p = 2$ and $c = 3$.

D. Minimum Splitter Firings to Fire a Corresponding Joiner

We initialize $x_J(J) = 1$. We traverse G_{SJ} in a reverse topological order (recall that we traverse an acyclic team graph). For each team T we visit, we compute $x_J(T)$ as follows.

$$
\begin{aligned}
&x_J(T) \\
&= \max_{\text{successor } U \text{ of } T} (\text{number of } T \text{ firings to fire } U \ x_J(U) \text{ times}) \\
&= \max_{\text{successor } U \text{ of } T} (\lceil \frac{x_J(U) \cdot cons(s)}{prod(s)} \rceil)
\end{aligned}
$$

E. Team Scheduling Pseudo Code

```
01  construct an initial schedule;
02  initial queue sizing; // Feedback queues are sized here.
03
04  // merge teams
05  q = a priority queue with pairs of teams
        that do not introduce a cycle;
06  while (!q.isEmpty()) {
07   ⟨a, b⟩ = q.remove();
08   if (merging a and b does not exceed buffer limit
        and does not deadlock) {
09     m = merge(a, b);
10     remove all team pairs containing a or b from q;
11     for each (neighbor c of a or b) {
12       if (merging m and c does not introduce a cycle)
13         add ⟨m, c⟩ to q;
14     }
15   }
16  }
17
18  // amortize teams
19  q = construct a priority queue with teams;
20  while (!q.isEmpty()) {
21   a = q.remove();
22   if (amortizing a does not exceed buffer limit
        and does not deadlock) {
23     amortize a;
24     add a to q;
25   }
26  }
```

The time complexity of team scheduling is dominated by the longest path algorithm for buffer requirement computation. Let $|V|$ be the number of actors and $|E|$ be the number of streams. We compute buffer requirement $O(|V|log(b))$ times (lines 8 and 22), when b is the buffer space limit per core and we amortize each team at least by a factor of 2. We use the Bellman-Ford algorithm [2] to compute the longest distance, whose time complexity is $O(|V| \cdot |E|)$. Bellman-Ford is invoked $O(|V|)$ times per each buffer requirement computation; thus the time complexity of team scheduling is $O(|V|^3|E|log(b))$.

The time complexity for cyclicity check is $O(|V|^3)$: cyclicity check is done $O(|V^2|)$ times (line 12) and each check takes $O(|V|)$ by using reachability matrix [4]. The time complexity of finding split-join patterns is $O((|E|log(|E|) + |V|^2)|V|log(b))$: we apply the concept of *dominance frontier* used for constructing *static single assignment* form in the compiler [6], which takes $O(|E|log(|E|) + |V|^2)$ [6, 26].

If the time complexity is unacceptable (e.g., in just-in-time compilation), we can stop during algorithm execution since we maintain a valid schedule throughout the algorithm execution.