Phased Scheduling of Stream Programs

Michal Karczmarek, William Thies and Saman Amarasinghe

{karczma, thies, saman}@lcs.mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology

ABSTRACT

As embedded DSP applications become more complex, it is increasingly important to provide high-level stream abstractions that can be compiled without sacrificing efficiency. In this paper, we describe scheduler support for StreamIt, a high-level language for signal processing applications. A StreamIt program consists of a set of autonomous filters that communicate with each other via FIFO queues. As in Synchronous Dataflow (SDF), the input and output rates of each filter are known at compile time. However, unlike SDF, the stream graph is represented using hierarchical structures, each of which has a single input and a single output.

We describe a scheduling algorithm that leverages the structure of StreamIt to provide a flexible tradeoff between code size and buffer size. The algorithm describes the execution of each hierarchical unit as a set of phases. A complete cycle through the phases represents a single steady-state execution. By varying the granularity of a phase, our algorithm provides a continuum between single appearance schedules and minimum latency schedules. We demonstrate that a minimal latency schedule is effective in decreasing buffer requirements for some applications, while the phased representation mitigates the associated increase in code size.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.3.2 [**Programming Languages**]: Language Classifications; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Algorithms, Languages, Performance, Experimentation

Keywords

Phased Scheduling, StreamIt, Synchronous Dataflow, Cyclo-Static Dataflow, Buffer Size, Code Size, Stream Programming, DSP

LCTES'03, June 11–13, 2003, San Diego, California, USA.

1. INTRODUCTION

From handheld computers to cell phones to sensor networks, there has been a surge of embedded applications that demand high-performance digital signal processing. These programs constitute a new and important class of applications: those that are centered around *streams* of data. Despite the widespread parallelism and regular communication patterns that are inherit in stream programs, application development in the streaming domain is still very labor-intensive and error-prone. In order to optimize critical loops, DSP programmers are often forced to resort to assembly code, thereby sacrificing portability and robustness for the sake of performance. As the complexity of embedded software grows, this practice will become infeasible. There is a pressing need to provide high-level stream abstractions that can be compiled without sacrificing efficiency.

The goal of the StreamIt project is to provide language and compiler support for high-level stream programming. A StreamIt program consists of a set of autonomous filters that communicate using FIFO queues. Filters can be combined into single-input, single-output modules by using a set of hierarchical primitives, thereby imposing a structure on the stream graph that is akin to structured control flow in a mainstream language. In order to facilitate static scheduling, the input and output rates of each filter are known at compile time.

In this paper, we present techniques for scheduling stream graphs such as those found in StreamIt. The StreamIt representation has much in common with Synchronous Dataflow (SDF) graphs [17], for which there is a large body of literature devoted to scheduling (see [4] for a review). There are two aspects of StreamIt programs that distinguish our scheduling problem from a general SDF graph: 1) StreamIt graphs are hierarchical, with each node having only a single input and single output, and 2) StreamIt allows a "peek" operation whereby nodes can operate on items that they do not consume until a future invocation. In this context, this paper makes the following contributions:

- Fundamental techniques for constructing a hierarchical schedule from a hierarchical stream graph.
- A method for computing an initialization schedule, which is a unique requirement of graphs supporting the peek construct.
- A parameterized phased scheduling algorithm that leverages the structure of a StreamIt graph to give a flexible tradeoff between code size and data size.

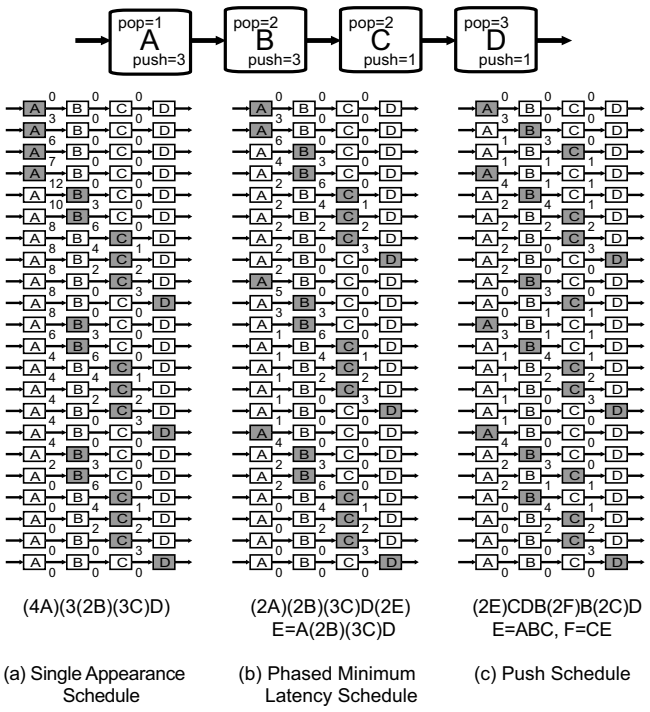


Figure 1: Execution trace of three different scheduling strategies for one steady state execution of a simple pipeline. Channels are annotated with the number of live data items that they contain; shaded nodes represent those that fire on a given time step.

- An instance of the phased scheduling algorithm that computes a minimal latency schedule, guaranteed to avoid deadlock in any valid feedback loop.

This paper is organized as follows. The remainder of this section gives an illustrating example and describes relevant StreamIt constructs; Section 2 explains basic concepts in scheduling StreamIt graphs; Section 3 describes the phased scheduling technique and presents a minimum latency scheduling algorithm; Section 4 presents experimental results; Section 5 describes related work and Section 6 presents conclusions and planned future work.

1.1 Example

A classic problem in the scheduling of synchronous dataflow graphs is the tradeoff between code size and data size [5]. Code size refers to the space needed to represent the schedule, while data size refers to the buffering of items during execution. Generally speaking, smaller schedules contain loops that require coarse-grained execution of nodes, thereby leading to larger buffer requirements.

Consider the example stream graph depicted in Figure 1. Even given a simple pipeline of filters, there is a large space of different schedules, each with different requirements for code and buffer size. Figure 1 illustrates two extreme scheduling policies. First is Single Appearance Scheduling (SAS), which gives the minimal code size: the schedule is a loop nest with each node appearing at a single position. SAS is generally the method of choice in the DSP community, as the contents of each node can be inlined without duplicating code. There are many different SAS schedules for a given

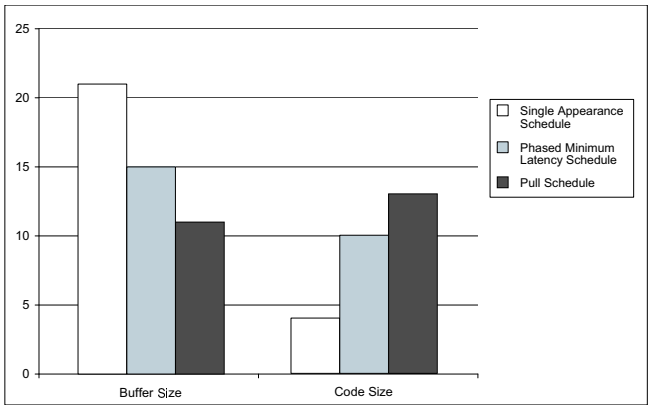


Figure 2: Buffer and code sizes for the execution traces of Figure 1. For brevity, we show these figures on the same graph, even though a unit of storage might have a different cost for code and data.

stream graph; the one shown in Figure 1 is the best SAS schedule for this case.

At the other end of the spectrum is a push schedule, which results in the minimal buffer size at the expense of code size (Figure 1(c)). A push schedule starts by executing the topmost node, and then pushes the items produced through the rest of the graph, always executing the most downstream node possible. When no further node can fire, the top node is executed again. In this case, the push schedule reduces the buffer size by 48% but increases code size by 325% over the SAS schedule.

In this paper, we develop a phased scheduling algorithm that offers a flexible alternative between the extremes of SAS and push scheduling. Shown in Figure 1(b) is the phased minimum latency schedule. It consists of three "phases", each of which is a single-appearance sub-schedule that results in a single output item being produced from the pipeline. The schedule has the same latency as the push schedule, but has reduced code size due the single-appearance property of the phases and the collapse of phases 2 and 3 into a single representation "E".

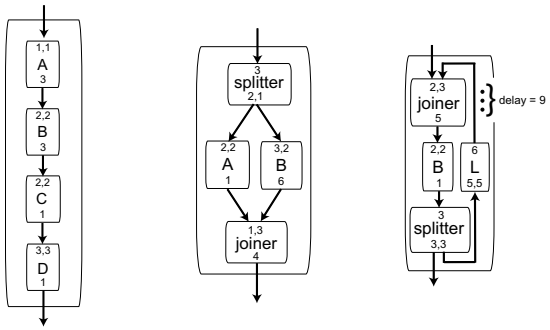
Figure 2 illustrates the tradeoff between code size and data size for the scheduling schemes. It shows that there can be a large tradeoff between code size and buffer size, with phased scheduling striking a compromise between extremes. In Section 3, we give a flexible version of our phased scheduling algorithm, and we also demonstrate that it can handle tight feedback loops for which there does not exist a valid single appearance schedule.

1.2 The StreamIt Language

The source language for our scheduler is StreamIt: an architecture-independent programming language for high-performance streaming applications. This section contains a very brief overview of the semantics of StreamIt. We do not concern ourselves with the syntax of the language, as it is not relevant to scheduling stream graphs. A more detailed description of the design and rationale for StreamIt can be found in [23] or on our website [13].

1.2.1 Language Constructs

The basic unit of computation in StreamIt is the filter.



(a) a pipeline (b) a splitjoin (c) a feedbackloop

Figure 3: Sample StreamIt operators. Each node is labeled with its peek, pop rates (at top) and push rate (at bottom). The L filter has been flipped upside-down for clarity.

A filter is a single-input, single-output block with a user-defined procedure for translating input items to output items. Every filter contains a work function, which is comprised of one or more atomic phases that the filter cycles through during its steady-state execution. A filter can optionally declare a `prework` function that executes instead of `work` on the first invocation of the filter, if special startup behavior is desired. Filters communicate with their neighbors via FIFO queues, called channels, using the intuitive operations of `push(value)`, `pop()`, and `peek(index)`, where `peek` returns the value at position `index` without dequeuing the item. The number of items that are pushed, popped, and peeked[1] on each invocation are declared with each phase of the work function.

StreamIt provides three primitives for composing filters into hierarchical streams (see Figure 3). The pipeline construct cascades a set of filters in sequence, with the output of one connected to the input of the next. The splitjoin construct is used to specify independent parallel streams that diverge from a common splitter and merge into a common joiner—for example, in the Equalizer of Figure 4. StreamIt currently supports two types of splitters: duplicate, which broadcasts its input items to each parallel stream, and roundrobin, which distributes items cyclically to one child after another according to an array of weights. The joiner node must be a roundrobin.

The last control construct provides a means for creating cycles in the stream graph: the feedbackloop. A feedbackloop contains a joiner, a body operator, a splitter, and a loop operator. A feedbackloop has an additional feature to allow it to begin computation: since there are no data items on the feedback path at first, the stream needs to enqueue initial values onto the channel. The number of items pushed onto the feedback path is called the delay, denoted $delay_{fl}$, for a feedbackloop fl.

1.2.2 Design Rationale

StreamIt differs from other stream languages in the single-input, single-output hierarchical structure that it imposes on streams. This structure aims to help the programmer by defining clean, composable modules that admit a linear textual representation. In addition, it helps the compiler by restricting certain analyses to a local level rather than dealing with global properties of the graph. In the context of scheduling, hierarchy is also useful because it allows for the separate compilation of program components. This enables the creation of standardized libraries and their distribution in binary form, rather than source code. This ability may become important as streaming languages become more widely used for larger applications.

[1]We define *peek* as the total number of items read, including the items popped. Thus, we always have that $peek \geq pop$.

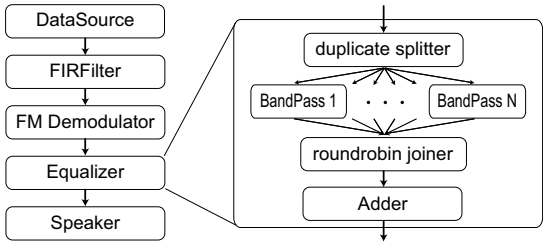


Figure 4: Block diagram of an FM Radio.

Another important feature of StreamIt—and one that requires special support from the scheduler—is the peek construct. By using the peek command, a filter can examine an input item at a given index without removing it from the channel. This exposes to the compiler the reuse of input items between successive invocations of a filter's work function. A primary example is an FIR filter, which pops 1 item but peeks N items. Without the capability to peek, the programmer would have to maintain a persistent circular buffer within the filter to retain previous input items. Apart from being difficult to implement and understand, this would greatly complicate compiler analysis. In particular, the linear analysis and optimization passes within the StreamIt compiler benefit greatly from analyzing peek statements directly instead of reverse-engineering internal filter state [16].

2. STREAMIT SCHEDULING CONCEPTS

This section introduces the general concepts used for scheduling StreamIt programs. Concepts presented here are common with other systems [18]. Section 2.1 presents the StreamIt execution model. Section 2.2 introduces the concept of a steady state and shows how to calculate it. Section 2.3 explains the need for initialization of a StreamIt program.

2.1 StreamIt Execution Model

A StreamIt program is represented by a hierarchical graph, where the leaf nodes are filters, splitters, and joiners, and the composite nodes are pipelines, splitjoins, and feedbackloops. Edges in the graph represent data channels, which operate as FIFO queues.

In order for a filter f to execute, it must have at least $peek_f$ items on its input channel. Execution will decrease the amount of data on its input channel by pop_f and increase the amount of data on its output channel by $push_f$. Similarly, a splitter s will consume pop_s data from its input channel and push $push_{s,i}$ data onto its ith output channel, while a joiner j will consume $pop_{j,i}$ items from its ith input channel and push $push_j$ onto its output channel.

Each filter, splitter, and joiner in the graph has two epochs of execution: one for initialization, and one for the steady state. Within each epoch, a given filter can have any number of phases, each of which is an atomic execution step with its

own input and output rates. At the start of the program, each node starts in phase 0 of the initial epoch. It then advances through its initialization phases, executing each a single time before transitioning to phase 0 of the steady state epoch. Within the steady state, a filter executes its steady state phases cyclically.

2.2 Steady State Schedule

One of the most important concepts in scheduling streaming applications is the steady state schedule. A steady state schedule is a schedule that the program can repeatedly execute forever. It has the property that the amount of data buffered up between any two nodes does not change from before to after its execution.

A "steady state" of a program is a collection of number of times that every node in the program needs to execute in a steady state schedule. It does not impose an order of execution on the nodes in the program.

2.2.1 Minimal Steady State

We now summarize some of the key properties of steady states, which are presented in [17]. Detailed proofs of these properties in the context of StreamIt can be found in [15].

The first property concerns the size of a steady state. The size is defined to be the sum of the repetitions of all nodes in the schedule.

THEOREM 1 (MINIMAL STEADY STATE UNIQUENESS). *A StreamIt program that has a valid steady state, has a unique minimal steady state.*

This means that for every valid StreamIt program, there is a unique set of steady state multiplicities that fires as few nodes as possible. Our scheduler will produce schedules that execute exactly the minimal steady state of a program.

THEOREM 2 (MULTIPLICITY OF STEADY STATES). *If a StreamIt program has a valid steady state, then all its steady states are strict multiples of its minimal steady state.*

This property means that in order to find a minimal steady state schedule of a stream operator, we can find any of its steady states and divide it by the *gcd* of executions of all its children to find the minimal steady state schedule.

2.2.2 Calculating Minimal Steady States

For a general stream graph, the minimal steady state can be calculated in a linear algebra framework by formulating a set of balance equations [17]. However, with StreamIt we leverage the structure of the stream graph to calculate steady states in a hierarchical manner. That is, a minimal steady state is calculated for all child operators of a pipeline, splitjoin and feedbackloop, and then the schedule is computed for the actual parent operator using these minimal states as atomic executions. This approach is useful in the context of separate compilation, where the entire graph might not be available at compile time; additionally, the steady state multiplicity of a given node in relation to its parent is useful for our scheduling algorithms.

For brevity, we omit the equations for finding the minimal steady states. The steady states are calculated hierarchically; filters with multiple phases are represented by a single, coarser phase for the sake of the steady-state schedule. Details can be found in [15]. For example, the minimal steady states of the stream graphs in Figure 3 are as follows:

Sample Pipeline:	$steady(\text{A}) = 4$
	$steady(\text{B}) = 6$
	$steady(\text{C}) = 9$
	$steady(\text{D}) = 3$
Sample SplitJoin:	$steady(\text{splitter}) = 2$
	$steady(\text{A}) = 2$
	$steady(\text{B}) = 1$
	$steady(\text{joiner}) = 2$
Sample FeedbackLoop:	$steady(\text{joiner}) = 6$
	$steady(\text{B}) = 15$
	$steady(\text{splitter}) = 5$
	$steady(\text{L}) = 3$

Note that these numbers represent the multiplicity of each node in one steady state execution of its parent. In Section 3, we consider how to order these executions to form a valid schedule.

2.3 Initialization Schedule

Unlike traditional SDF graphs, StreamIt programs may require a separate schedule for initialization. This is for two reasons. First, each filter might contain an initialization stage, where the input and output rates are different than in the steady state. But even without the initialization epoch, an initialization schedule is necessary if any filter makes use of StreamIt's *peek* construct, in which input items can be examined without being consumed.

To understand the impact of peeking on scheduling, consider a filter f, with $peek_f = 2$ and a $pop_f = 1$. When a StreamIt program is first run, there is no data present on any of the channels (ignoring the case of a feedbackloop delay). This means that for the first execution, filter f requires that two data items be pushed onto its input channel. After the first execution of f, it will have consumed one data item, and left at least one data item on its input channel. Thus in order to execute f for the second time, no more than one extra data item needs to be pushed onto f's input channel. The same situation persists for all subsequent executions of f – no more than one additional data item is required on f's input channel in order to execute f.

This example illustrates that the first execution of a filter may require special treatment. Namely, some nodes will need to push extra items at the start of execution so that downstream filters can fire for the first time. Due to this condition, a StreamIt node may need to be initialized before it can enter steady state execution.

3. PHASED SCHEDULING

A schedule for a given hierarchical node of a StreamIt program is a list of the node's immediate children, specifying the order in which they should be executed. More precisely, since filters (and, as we will see, hierarchical nodes as well) can have multiple phases, a schedule is a list of phases of child nodes. In order for a schedule to be legal, it must satisfy two conditions: first, for every execution of a node, a sufficient amount of data must be present on its input channel(s); second, in the case of the steady state, an infinite repetition of the schedule must require a finite amount of memory. The second condition is ensured by using the steady state multiplicities calculated in the previous section, while the first condition is one that we must respect when choosing an ordering for the nodes.

Our phased scheduling algorithm, shown in Figure 5, operates in a hierarchical fashion. That is, it constructs a schedule for a given pipeline, splitjoin, or feedbackloop as a sequence of the schedules of its children. A schedule is represented as a sequence of phases. In the base case of a filter, these phases are specified by the StreamIt program (with one small modification, described below), while at hierarchical nodes they are computed by our algorithm. To schedule an entire StreamIt program, our algorithm should be applied as a post-order traversal of the stream graph.

Intuitively, our algorithm is based on the observation that a hierarchical stream displays cyclic behavior as it executes its components. At the coarsest level of granularity, these cycles are evident in the steady state schedule: each iteration of the steady state is exactly the same. The aim of our algorithm is to exploit a finer level of granularity in execution behavior—the basic unit being a phase of the push schedule for the stream. Generally speaking, a phase of the push schedule holds the smallest sequence of filter executions that will both consume input and produce output for the stream. Our algorithm allows a parameterized level of granularity by collapsing some of these fine-grained phases together and shuffling the resulting schedule so that the phases of a given child stream are all adjacent. As we demonstrate below, a single appearance schedule and minimum latency schedule are both special cases of a parameterized phased schedule. For a more mathematical description of the phased scheduling technique, see [15].

```
// a Phase represents a component of a schedule
struct Phase {
  Stream str              : stream that this Phase corresponds to
  Phase[] children        : component Phases, each corresponding to a child of <str>
}

// returns a "push phase" for <s>: the minimal sequence of child executions that
// executes the entrance node, the exit node, and flushes all data downstream
Phase pushSchedule (Stream s) {
  result ← Phase(s, {})
  child ← entrance(s)
  do {
    result.children ← result.children ∘ currentPhase(child)
    simulate(child)
    child ← most downstream node in s that can fire
  } loop until (child = entrance(s) ∧ exit(s) has fired at least once)
  return result
}

// returns a phased schedule for <s> that contains no more than <maxPhases> phases
Phase[] phasedSchedule (Stream s, int maxPhases, int mode) {
  // get maximally fine-grained phases
  phases ← {}
  do {
    phases ← phases ∘ pushSchedule(s)
  } loop until { each node of s has fired all its init phases      (if mode = INIT)
               { each node of s has completed its steady state  (if mode = STEADY)
  // combine into <maxPhases> groups, with contents sorted by child stream
  numPhases ← min(maxPhases, phases.length)
  ∀ i ∈ [0, numPhases - 1] {
    phaseStart ← floor(i × phases.length/numPhases)
    phaseEnd ← floor((i+1) × phases.length/numPhases) - 1
    newChildren ← phases[phaseStart].children ∘ … ∘ phases[phaseEnd].children
    result[i] ← Phase(s, sortByStream(newChildren))
  }
  return result
}
```

Figure 5: Phased Scheduling Algorithm.

3.1 Algorithm Details

We now consider in more detail the pseudocode in Figure 5; refer to Figures 6 and 7 for an example. The algorithm inputs a Stream `s` and returns a sequences of phases that represent the schedule for that stream. It also inputs two additional parameters: `maxPhases`, which specifies the maximum number of phases in the resulting schedule, and `mode`, which indicates whether we are scheduling for the initial or steady-state epoch. The algorithm starts by assembling a series of fine-grained phases, each of which corresponds to a push schedule as built by the `pushSchedule` routine.

The `pushSchedule` routine simulates a push schedule until the bottom node is fired at least once (in most cases, this will correspond to an output being produced). A push schedule is one in which downstream nodes are fired as much as possible before upstream nodes are considered. The routine starts with the entrance node of the stream, *i.e.,* the first child of a pipeline, the splitter of a splitjoin, or the joiner of a feedbackloop. It then pushes live items as far forward as possible, only executing the entrance node again if the exit node could not fire.

There are two subtleties in the `pushSchedule` procedure. First, note that it always flushes extra items from the stream: the exit node might fire multiple times, even though all firings were caused by a single execution of the entrance node. Second, in the case of a feedbackloop, it is careful to push items around the feedback path even after the splitter (the exit node) has fired. That is, the ranking of nodes in a feedback loop is (*joiner*, *body*, *splitter*, *loop*), and pushing of items through the loop node is necessary to ensure a correct steady-state schedule.

The `phasedSchedule` routine builds up a maximal list of phases from the push schedule. In the steady state, this list is complete when each node has completed its steady state repetitions, while in the initialization mode, simulation is finished when each node has executed its initial phases. To ensure that the initialization schedule provides enough data items for the peeking requirements of the steady state, we add an extra initialization phase to each filter before running the algorithm. For filter f, this phase has rates $peek' = peek_f - pop_f, pop' = push' = 0$. Since this phase must execute in the initial schedule, it ensures that there will be $peek - pop$ items present at the start of the steady state. A steady state schedule is then possible to construct, since the filter can return the buffer to this state by firing once with *peek* items on the channel.

Once it has gathered the list of maximally fine-grained phases, the `phasedSchedule` algorithm makes two modifications. First, it combines some adjacent phases so that only `maxPhases` are returned. Combination works simply by concatenating the sequence of child executions from the given phases. Second, even if no phases are combined, the algorithm re-arranges the order of child phases so that all phases corresponding to a given stream are adjacent. This is an attempt to provide a canonical form for a given series of executions, so that phases with the same form can be compressed in the resulting schedule (see Section 3.4).

Note that the pseudocode given in Figure 5 specifies only the behavior of the algorithm, rather than the implementation. In our implementation, we avoid symbolic execution of the entire steady state by calculating, from the bottom-up, the number of node firings that will be required in each phase. If upstream nodes produce more data than is necessary, then we drain this data through the stream by firing downstream nodes again. In this technique, each child is visited no more than twice per calculation of a parent phase.

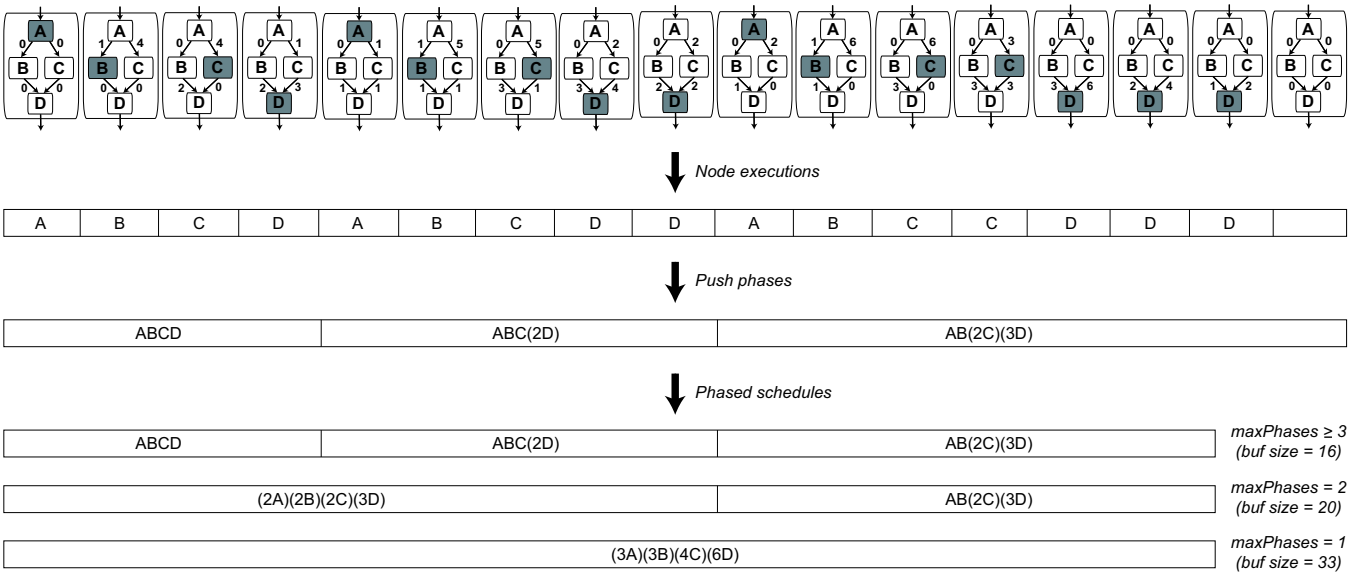


Figure 6: Example construction of phased schedules for the splitjoin of Figure 7. First, execution is simulated for one steady state according to a push schedule; the stream graph is labeled with the number of items on each channel following the firing of a shaded node. Then, fine-grained phases are formed that include executions of both the entry (A) and exit (D) nodes. Finally, the fine-grained phases are combined into *maxPhases* phases, each of which is factored into a single appearance schedule. Note that buffer size increases with the granularity of the phases, as shown at right.

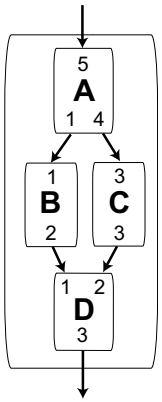


Figure 7: Example splitjoin to illustrate phased scheduling. Each node is annotated with its input and output rates.

3.2 Generalizing Other Techniques

As alluded to above, single appearance scheduling and minimum latency scheduling are special cases of our parameterized phased scheduling algorithm. A single appearance schedule is defined as a schedule where each node appears in one position of the loop nest denoting the schedule. Because the nodes within a phase are sorted by child stream, this is equivalent to a phased schedule with a single phase:

$$singleAppSchedule[s] = \langle phasedSchedule(s, 1, \text{INIT}), phasedSchedule(s, 1, \text{STEADY})\rangle$$

A minimum latency schedule exhibits the following property: if i input items have been consumed by a hierarchical node when it produces its jth output item, then there does not exist a schedule which produces j output items while consuming less than i input items. This condition is necessary and sufficient for a schedule to be minimum latency. We can construct a minimum latency schedule as a phased schedule with an unlimited number of phases:

$$minLatencySchedule[s] = \langle phasedSchedule(s, \infty, \text{INIT}), phasedSchedule(s, \infty, \text{STEADY})\rangle$$

This schedule is guaranteed to be minimum latency, since it is comprised of push phases that do not fire the entrance node once the exit node has been fired (see `pushSchedule` in Figure 5).

Thus, single appearance and minimum latency schedules represent extreme values of the `maxPhases` parameter. Other values of `maxPhases` indicate compromises between these two extremes. Also, note that different levels of granularity could be applied to different streams in the same graph, depending on the constraints; the algorithm does not depend on the granularity of the children when it is scheduling a parent node.

3.3 Scheduling Feedback Loops

Some feedback loops require a minimum number of phases in order to construct a valid schedule. This is because if the latency of child streams is too high, then a node could deadlock waiting for its own (upcoming) output to propagate through the loop. For example, in our GSM benchmark, there is a tightly constrained feedback loop (see Figure 9). While it is impossible to schedule this loop with a single appearance schedule, a minimum latency schedule results in a legal ordering (see Figure 8).

Figure 10 provides an algorithm for calculating the minimum number of phases that are required to schedule a feedback loop. The routine's functionality is similar to the phased scheduler, except for one key difference: the joiner is executed as much as possible before the items that it pushes are propagated around the loop. This ensures that the reshuffling step of the phased scheduling algorithm will be legal, since no element in the schedule will depend on items that it produced earlier in the same phase. Note that the `phasedSchedule` algorithm gives an undefined result if a given loop is impossible to schedule with the requested number of phases; thus, `phasesForFeedback` should always be called first to see how many phases are needed.

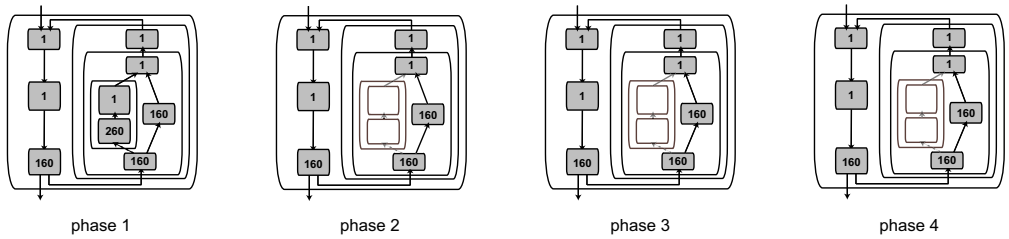


Figure 8: Phased minimum latency schedule for one steady state execution of the feedback loop of Figure 9. Nodes are labeled with the number of times they fire in a given phase. No single appearance schedule exists for this loop.

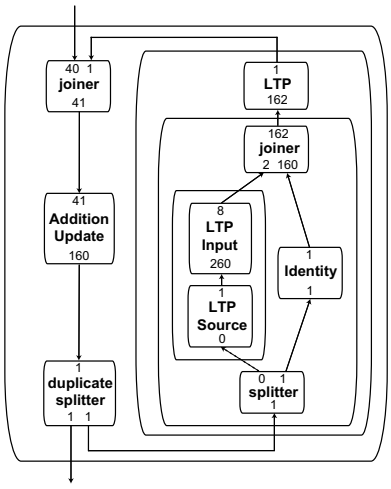


Figure 9: A tightly coupled feedback loop that appears as one component of our GSM benchmark. Nodes are labeled with their pop and push rates.

```
// returns the minimum number of phases required to execute feedbackloop <s>
int phasesForFeedback(FeedbackLoop s, int mode) {
  phaseCount ← 0
  do {
    phaseCount++
    while (canFire(joiner(s)))
      simulate(joiner(s))
    while ( ∃ c ∈ s, c ≠ joiner(s) and canFire(c))
      simulate(c)
  } loop until { each node of s has fired all its init phases      (if mode = INIT)
               { each node of s has completed its steady state  (if mode = STEADY)
  return phaseCount
}
```

Figure 10: Algorithm to detect the minimum number of phases required by a given feedback loop.

3.4 Schedule Representation

In the discussion above, a schedule is represented simply as a sequence of phases for child nodes. However, since this representation can become large for some programs, our implementation employs compression to keep code size to a minimum.

We compress the schedule in three simple ways. First, we collect repetitions of identical phases into a loop. For example, if `A` is a phase:

$$\texttt{A=BBB} \rightarrow \texttt{A=\{3B\}}$$

Second, if a hierarchical phase contains only one execution of a child, we substitute all occurrences of the parent with a direct call to the child:

$$\texttt{A=BCD, C=E} \rightarrow \texttt{A=BED}$$

Finally, if a phase is used only once, then it can be replaced by its child phases, even if there are multiple children:

$$\texttt{A=BCD, C=EFG, C} \textit{ used only once} \rightarrow \texttt{A=BEFGD}$$

To improve the compression of the schedule, we repeatedly apply the above three transformations until no further changes can be made.

In the future, an additional optimization could be explored regarding schedule compression. Instead of representing different phases for a given stream by distinct entries in the schedule, we could record only the name of the stream in the schedule and postpone the resolution of the current phase number until runtime. This would allow more opportunities for schedule compression, as two different phases would be considered equal if they call the same child streams, rather than requiring them to call the same phases on those children. However, proper evaluation of this technique would need to take into account the overhead of this indirection at runtime, and we do not evaluate it here.

4. RESULTS

We have implemented the phased single appearance and minimum latency scheduling algorithms as part of the StreamIt compiler, and we evaluate them in this section. Section 4.1 presents the applications used for evaluation, while Section 4.2 presents the results and analysis.

4.1 Benchmarks

Our benchmark suite contains 17 applications. Out of these applications, 15 represent meaningful computations taken from real-life applications, while two were chosen to highlight the effectiveness of phased scheduling.

SJPeek1024 and SJPeek31 are synthetic benchmarks, designed to highlight the strengths of phased schedules [15]. SJPeek1024 requires an initialization schedule which benefits from the finer granularity of minimum latency scheduling. SJPeek31 contains a push/pop mismatch which causes a combinatorial blow-up using single appearance scheduling.

Nine test applications (BitonicSort, FFT, FilterBank, FIR, Radio, GSM, 3GPP, Radar and Vocoder) are also used in [10]. BitonicSort performs a 32 element bitonic sort; FFT performs a 64-element FFT; FilterBank is an 8 channel filter bank; FIR is a 64-tap fine-grained FIR filter; Radio is an FM radio decoder with an equalizer; 3GPP is a 3GPP Radio Access Protocol application; Radar is a radar array front-end with beamforming; Vocoder is a 28 channel Vocoder.

Two test applications (CD-DAT and QMF) are borrowed from [20]. We model only the communication properties of

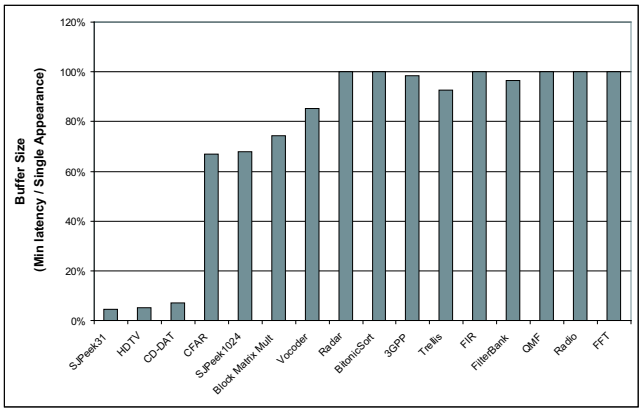


Figure 11: Buffer size required by a phased minimum latency schedule, normalized to buffer size of a hierarchical single appearance schedule.

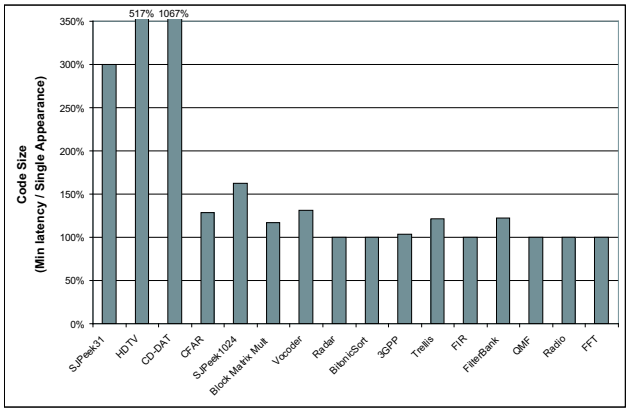


Figure 12: Code size required by a phased minimum latency schedule, normalized to code size of a hierarchical single appearance schedule.

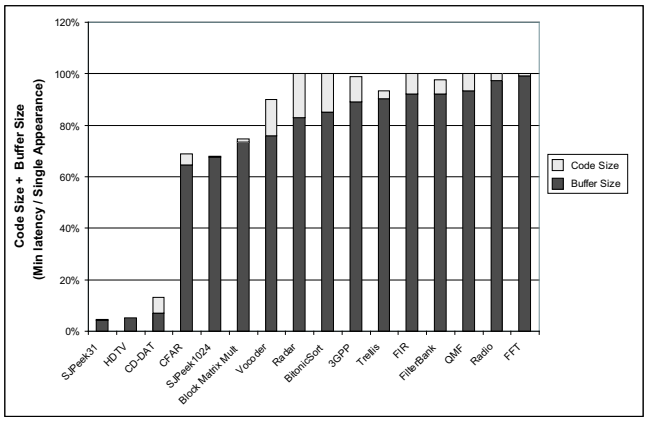


Figure 13: Sum of buffer and code size required by a phased minimum latency schedule, normalized to that of a hierarchical single appearance schedule. We give equal weight to the code and buffer size only for illustration; in an actual system, the relative weights are complex and depend upon resource constraints.

the graphs; the code inside of the filters has not been implemented. CD-DAT performs sample rate conversion and is exactly the same as that described in [20]. QMF is a filter bank application which uses a 1/2-1/2 split for the spectrum up to a depth of 3 (qmf12_3d in [20]). It was slightly modified to use StreamIt's pre-defined splitter and joiner constructs. The high-pass and low-pass filtering in multiple-output blocks has been converted to a splitter followed by filters on each of the output channels. The low and high pass filters have also been given a peek amount of 16 so they can perform their function in the way intended by StreamIt.

The remaining 4 applications were chosen from our sample applications used for testing the StreamIt compiler. HDTV performs a HDTV signal decoding/encoding. CFAR implements PCA Constant False Alarm Rate detection. Block Matrix Mult performs a blocked matrix multiplication - it multiplies a 12x12 matrix by a 9x12 matrix in blocks of 3x3 sub-matrices. Trellis performs trellis encoding/decoding.

4.2 Results

Table 1 presents the code and buffer sizes required by our hierarchical single appearance and minimum latency scheduling algorithms for our benchmark suite. Note that the GSM application cannot be scheduled using a single appearance schedule, because it has a tightly constrained feedback loop (see Figure 9). Thus, we omit GSM in Figures 11 to 13.

Several applications show a very large improvement in buffer size necessary for execution (see Figure 11). These improvements are usually coupled with an increase in code size (Figure 12). However, as shown in Figure 13, minimum latency scheduling never increases the sum of code size and data size for any application. In computing this sum, note that we give equal weight to the code size and data size only for the sake of illustration; in an actual system, the relative cost of each kind of storage would greatly depend on resource constraints and the implementation strategy.

The CD-DAT benchmark exhibits a decrease in buffer size from 1021 to 72, a 93% improvement. [20] reports a buffer size of 226 after applying buffer merging techniques. Our improvement is due to reducing the combinatorial growth of the buffers using phased scheduling.

For our synthetic benchmarks SJPeek31 and SJPeek1024, buffer sizes decrease by 95% and 32%, respectively. In the case of SJPeek1024, the improvement is due to creating fine grained phases which allow the initialization schedule to transfer smaller amounts of data and allow the children of a splitjoin to drain their data before the splitter provides them with more. This improvement is only evident in the presence of peeking. In the case of SJPeek31, the improvement reflects reduced combinatorial growth in addition to the fine-grained benefit with peeking.

It is important to note that the schedules we consider in our evaluation have the elements of a hierarchical phase sorted as described in Section 3: all of the phases of a given child stream are executed before advancing to the next child. For both single-appearance and minimum latency scheduling, this represents only one possible execution order for child phases; in particular, a more fine-grained interleaving of children could reduce buffer requirements. While we do not explore the range of possible interleavings within

Benchmark	Number of Nodes	Number of Node Executions	Single Appearance		Minimal Latency	
			Code Size	Buffer Size	Code Size	Buffer Size
SJPeek31	6	12063	8	19964	24	874
HDTV	170	390038	230	550692	1190	28300
CD-DAT	6	612	6	1021	64	72
CFAR	4	193	7	193	9	129
SJPeek1024	6	3081	8	7168	13	4864
Block Matrix Mult	43	1956	48	4212	56	3132
Vocoder	117	415	156	1285	205	1094
Radar	68	161	68	332	68	332
BitonicSort	370	468	370	2112	370	2112
3GPP	94	356	104	986	108	970
Trellis	14	301	14	538	17	499
FIRfine	132	152	132	1560	132	1560
FilterBank	53	312	95	2063	116	1991
QMF	65	184	85	1225	85	1225
Radio	30	43	35	1351	35	1351
FFT	26	448	26	3584	26	3584
GSM	47	3356	-	-	64	3900

Table 1: Results of running single appearance and minimal latency scheduling algorithms on various applications.

a hierarchical node, note that the hierarchy of the stream graph provides a set granularity at which the leaf nodes of the graph can be interleaved—for example, in a hierarchical single-appearance schedule, two consecutive executions of a pipeline construct would execute all of its nodes once before executing all of the nodes again. We are exploring other interleaving strategies for the nodes within a given phase.

5. RELATED WORK

There has been a wealth of research on scheduling dataflow graphs. This section introduces some of the other projects.

Ptolemy [18] is a simulation environment for heterogeneous embedded systems, including Synchronous Data Flow, the domain that is most similar to StreamIt. SDF programs, however, do not include the peeking constructs of StreamIt. The SDF computation model does not impose structure on the program. All actors (the SDF equivalent of filters) are allowed to have multiple input and output channels. [1] provides an overview of dataflow synchronous languages.

There are many results on the scheduling of SDF programs [4, 6]. While the tradeoff between data size and code size is well recognized, most projects focus on minimizing memory requirements while maintaining minimal code size in the form of a single appearance schedule. A single appearance schedule is attractive because filters can be inlined into the schedule without effecting the size of the code. In this paper, we assume that the schedule and the filter code are stored separately, and that non-single appearance schedules are supported with function calls. [5] considers a hybrid model between these two extremes, in which actor invocations are inlined unless the resulting code grows too large.

There are a number of approaches to minimizing the buffer requirements for single-appearance schedules (see [4] for a review). APGAN (Pairwise Grouping of Adjacent Nodes) and RPMC (Recursive Partitioning by Minimum Cuts) are two complementary heuristics that have shown to be effective when applied together, taking the best result [3]. Another technique for reducing buffering requirements is buffer merging [19, 20], which could be explored for use in StreamIt in the future. Yet another approach is the GASAS system, which uses genetic algorithms to minimize buffer size [25].

Buffer minimization has also been done in the context of a multiprocessor implementation [11]. Using a linear programming framework, they minimize the buffer size across schedules that have optimal throughput.

There are some streaming computation models which are less constrained than SDF. The most relevant is Cyclo-Static Data Flow (CSDF) [7, 21]. CSDF actors have multiple work functions, each of which can produce/consume a different number of data items. Phased scheduling could be viewed as a generalization of CSDF to the case where hierarchical stream containers – not just leaf nodes – have cyclic phases. In addition, incorporating child phases into parent schedules allows the phase information in a CSDF graph to be fully utilized, *e.g.,* for decreasing latency and for scheduling tightly constrained feedback loops.

[24] proposes a model where the flow of data is not static, but may depend on data being processed. The model is called Cyclo-Dynamic Data Flow (CDDF). This greatly improves the flexibility of programming, but prevents fully static scheduling of programs. The U.S. Navy Processing Graph Method (PGM) uses a version of SDF with an equivalent of peeking [9]. The paper is focused on real-time execution and provides analysis and verification of latency of data flow through their system.

A large number of programming languages have included a concept of a stream; see [22] for a survey. Synchronous languages such as LUSTRE [12], Esterel [2], and Signal [8] also target the embedded domain, but they are more control-oriented than StreamIt and are less amenable to static scheduling. Sisal (Stream and Iteration in a Single Assignment Language) is a high-performance, implicitly parallel functional language [14]. The Distributed Optimizing Sisal Compiler [14] considers compiling Sisal to distributed memory machines, although it is implemented as a coarse-grained master/slave runtime system instead of a fine-grained static schedule.

6. CONCLUSION AND FUTURE WORK

This paper presents a general phased scheduling algorithm for structured Synchronous Dataflow Graphs as used by the StreamIt language. Unlike other languages, StreamIt enforces a hierarchical, single-input single-output structure on the stream graph, thus allowing a variety of new approaches to stream scheduling.

A hierarchical approach to scheduling of streaming applications allows for very simple algorithms. Program graphs do not have to be considered globally, thus less data needs to be kept track of. In the hierarchical approaches presented here, we only need to consider immediate children of a given stream operator.

The phased approach to scheduling allows the scheduling of arbitrarily tight feedbackloops and allows for more fine-grained control of buffer requirements. The fine-grained control of buffer requirements can provide dramatic reduction of buffer sizes when scheduling streaming applications, as has been presented here. Furthermore, phased schedules lend themselves to some easy forms of compression, thus further reducing the schedule size.

Future work will concentrate on expanding phased scheduling to implement schedules that have some real-life constraints put upon them. For example, a program may need to keep all its data in processor caches to provide high performance. Adapting buffer sharing to phased scheduling will also be explored, as it promises further reduction in buffer requirements.

7. ACKNOWLEDGEMENTS

Many people have contributed to the StreamIt infrastructure that was utilized in this paper, including Michael Gordon, David Maze, Jasper Lin, and Andrew Lamb. We also thank Ali Meli, Chris Leger, and Jeremy Wong for implementing some of the applications. The StreamIt project is supported by grants from DARPA, NSF, and the MIT Oxygen Alliance.

8. REFERENCES

[1] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-Flow Synchronous Languages. In *REX School/Symposium*, pages 1–45, 1993.

[2] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] Chuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Impelementations. *Journal of Design Automation for Embedded Systems*, January 1997.

[4] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2), June 1999.

[5] S. S. Bhattacharyya. Optimization Trade-offs in the Synthesis of Software for Embedded DSP. In *CASES*, October 1999. Washington, D. C.

[6] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[7] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-Static Dataflow. *IEEE Trans. on Signal Processing*, pages 397–408, February 1996.

[8] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag Lecture Notes in Computer Science*, 274:257–277, 1987.

[9] S. Goddard and K. Jeffay. Analyzing the Real-Time Properties of a U.S. Navy Singer Processing System. *International Journal of Reliability. Quality and Safety Engineering*, 7(4), 2000.

[10] Michael Gordon, William Thies, Michal Karczmarek, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, pages 75–86, October 2002.

[11] R. Govindarajan, Guang R. Gao, and Palash Desai. Minimizing memory requirements in rate optimal schedules. In *Proc. of the 1994 International Conference on Application Specific Array Processors*, pages 75–86. IEEE Computer Society, August 1994.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[13] StreamIt Homepage. `http://compiler.lcs.mit.edu/streamit`.

[14] J. Gaudiot and W. Bohm and T. DeBoni and J. Feo and P. Mille. The Sisal Model of Functional Programming and its Implementation. In *Proc. of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis*, 1997.

[15] Michal Karczmarek. Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language. S.M. Thesis, Massachusetts Instititue of Technology, Laboratory for Computer Science, 2002.

[16] Andrew Lamb, William Thies, and Saman Amarasinghe. Linear Analaysis and Optimization of Stream Programs. In *PLDI*, 2003.

[17] E. Lee and D. Messershmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, C-36(1):24–35, January 1987.

[18] Edward A. Lee. Overview of the Ptolemy Project. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, UC Berkeley, CA, March 2001.

[19] P. K. Murthy and S. S. Bhattacharyya. A Buffer Merging Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications. In *International Symposium on System Synthesis*, 1999.

[20] P. K. Murthy and S. S. Bhattacharyya. Buffer Merging — A Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications. Technical report, Inst. for Adv. Computer Studies, UMD College Park, 2000.

[21] T. Parks, J. Pino, and E. Lee. A Comparison of Synchronous and CycloStatic Dataflow. In *IEEE Asilomar Conference on Signals, Systems, and Computers*, 1995.

[22] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[23] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the International Conference on Compiler Construction*, 2002.

[24] P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-dynamic dataflow. In *4th EUROMICRO Workshop on Parallel and Distributed Processing*, January 1996.

[25] Eckart Zitzler, Jurgen Teich, and Shuvra S. Bhattacharyya. Evolutionary Algorithms for the Synthesis of Embedded Software. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 8(4), August 2000.