Phased Scheduling of Stream Programs

Michal Karczmarek, William Thies and Saman Amarasinghe

MIT LCS

Streaming Application Domain

- Based on audio, video and data streams
- Increasingly prevalent
 - Embedded systems
 - Cell phones, handheld computers, etc.
 - Desktop applications
 - Streaming media
 - Software radio
 - Real-time encryption
 - High-performance servers
 - Software Routers (ex. Click)
 - Cell phone base stations
 - HDTV editing consoles

Properties of Stream Programs

- A large (possibly infinite) amount of data
 - Limited lifespan of each data item
 - Little processing of each data item
- A regular, static computation pattern
 - Stream program structure is relatively constant
 - A lot of opportunities for compiler optimizations

StreamIt Language



Our Contributions

New scheduling technique called Phased Scheduling

- Small buffer sizes for hierarchical programs
- Fine grained control over schedule size vs buffer size tradeoff
- Allows for separate compilation by always avoiding deadlock
- Performs initialization for peeking Filters

Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

Stream Programs

- Consist of Filters and Channels
- Filters perform computation
- Channels act as FIFO queues for data between Filters



Filters

- Execute a work function which:
 - Consumes data from their input
 - Produces data to their output
- Filters consume and produce constant amount of data on every execution of the work function
 - Rates are known at compilation time
- Filter executions are atomic



Stream Program Schedule

- Describes the order in which filters are executed
- Needs to manage grossly mismatched rates between filters
- Manages data buffered up in channels between filters
- Controls latency of data processing

Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

- Performs the computation
- Consumes pop data items
- Produces push data items
- Inspects peek data items

peek, pop
push
$\underbrace{}_{}$

Example:FIR filter



- Example:
 - FIR filter
 - Inspects 3 data items



- Example:
 - FIR filter
 - Inspects 3 data items
 - Consumes 1 data item



- Example:
 - FIR filter
 - Inspects 3 data items
 - Consumes 1 data item
 - Produces 1 data item



Example:

- FIR filter
- Inspects 3 data items
- Consumes 1 data item
- Produces 1 data item



- FIR filter
- Inspects 3 data items
- Consumes 1 data item
- Produces 1 data item
- And again...



- FIR filter
- Inspects 3 data items
- Consumes 1 data item
- Produces 1 data item
- And again...



- FIR filter
- Inspects 3 data items
- Consumes 1 data item
- Produces 1 data item
- And again...



Example:

- FIR filter
- Inspects 3 data items
- Consumes 1 data item
- Produces 1 data item
- And again...



- FIR filter
- Inspects 3 data items
- Consumes 1 data item
- Produces 1 data item
- And again...



StreamIt Pipeline

- Connects multiple components together
- Sequential (data-wise) computation
- Inserts implicit buffers between them



StreamIt SplitJoin

- Also connects several components together
- Parallel computation construct
- Allows for computation of same data (DUPLICATE splitter) or different data (ROUND_ROBIN splitter)



StreamIt FeedbackLoop

- ONLY structure to allow data cycles
- Needs initialization on feedbackPath
- Amount of data on feedbackPath is *delay*



Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

Scheduling – Steady State

- Every valid stream graph has a Steady State
- Steady State does not change amount of data buffered between components
- Steady State can be executed repeatedly forever without growing buffers

- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of 2







- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:

le



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - A



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - A



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - A



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AA



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AA



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AA



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AAB


- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AAB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AAB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - A



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - A



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - A



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - AB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - AB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - AB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABA



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABA



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABA



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABAB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABAB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABAB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABABB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABABB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABABB



- 3:2 Rate Converter
- First filter (A) upsamples by factor of 3
- Second filter (B) downsamples by factor of two
- Schedule:
 - AABBB
 - ABABB



Steady State Example - Buffers

- AABBB requires 6 data items of buffer space between filters A and B
- ABABB requires 4 data items of buffer space between filters A and B



Steady State Example - Latency

- AABBB First data item output after third execution of an filter
 - Also A already consumed 2 data items
- ABABB First data item output after second execution of an filter
 - A consumed only 1 data item



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - A



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - AA



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - AAB



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - AABB
 - Can't execute B again!



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - AABB
 - Can't execute B again!
- Can't execute A one extra time:
 - AABB



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - AABB
 - Can't execute B again!
- Can't execute A one extra time:
 - AABBA



- Filter Peeking provides a new challenge
- Just Steady State doesn't work:
 - AABB
 - Can't execute B again!
- Can't execute A one extra time:
 - AABBAB
 - Left 3 items between A and B!



- Must have data between A and B before starting execution of Steady State Schedule
- Construct two schedules:
 - One for Initialization
 - One for Steady State
- Initialization Schedule leaves data in buffers so Steady State can execute






Initialization Schedule:



Initialization Schedule:

- Leave 3 items between A and B
- Steady State Schedule:



Initialization Schedule:

- Leave 3 items between A and B
- Steady State Schedule:





Initialization Schedule:

- Leave 3 items between A and B
- Steady State Schedule:
 - AA



Initialization Schedule:

- Leave 3 items between A and B
- Steady State Schedule:
 - AAB



Initialization Schedule:

- Leave 3 items between A and B
- Steady State Schedule:
 - AABB



Initialization Schedule:

- Leave 3 items between A and B
- Steady State Schedule:
 - AABBB



Initialization Schedule:

• A

- Leave 3 items between A and B
- Steady State Schedule:
 - AABBB
 - Leave 3 items between A and B



Initialization Schedule:

• A

- Leave 3 items between A and B
- Steady State Schedule:
 - AABBB
 - Leave 3 items between A and B
- See paper for more details



Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

Scheduling

- Steady State tells us how many times each component needs to execute
- Need to decide on an order of execution
- Order of execution affects
 - Buffer size
 - Schedule size
 - Latency

Single Appearance Scheduling (SAS)

- Every Filter is listed in the schedule only once
- Use loop-nests to express the multiplicity of execution of Filters
- Buffer size is not optimal
- Schedule size is minimal

Schedule Size

- Schedules can be stored in two ways
 - Explicitly in a schedule data structure
 - Implicitly as code which executes the schedule's loop-nests
- Schedule size = number of appearances of nodes (filters and splitters/joiners) in the schedule
 - Single appearance schedule size is same as number of nodes in the program
 - Other scheduling techniques can have larger size
 - SAS schedule size is minimal: all nodes must appear in every schedule at least once

- Example: CD-DAT
- CD to Digital Audio Tape rate converter
- Mismatched rates cause large number of executions in Steady State



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule: 7*
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule: 7*
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



- Naïve SAS schedule:
 - 147A 98B 28C 32D
 - Required Buffer Size: 714
 - Unnecessarily large buffer requirements!
- Optimal SAS CD-DAT schedule:
 - 49{3A 2B} 4{7C 8D}
 - Required Buffer size: 258



Pull Schedule Example – Buffer Size

- Pull Scheduling:
 - Always execute the bottom-most element possible
- CD-DAT schedule:
 - 2A B A B 2A B A B C D ... A B C 2D
 - Required Buffer Size: 26
 - 251 entries in the schedule
- Hard to implement efficiently, as schedule is VERY large



SAS vs Pull Schedule

	Buffer Size	Schedule Size
SAS	258	4
Pull Schedule	26	251

Need something in between SAS and Pull Scheduling

Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

Idea:

- What if we take the naïve SAS schedule, and divide it into n roughly equal phases?
- Buffer requirements would reduce roughly by factor of n
- Schedule size would increase by factor of n
- May be OK, because buffer requirements dominate schedule size anyway!

- Try n = 2:
- Two phases are:
 - 74A 49B 14C 16D
 - 73A 49B 14C 16D
- Total Buffer Size: 358
- Small schedule increase
- Greater n for bigger savings



- Try n = 3:
- Three phases are:
 - 48A 32B 9C 10D
 - 53A 35B 10C 11D
 - 46A 31B 9C 11D
- Total Buffer Size: 259
- Basically matched best SAS result
 - Best SAS was 258



- Try n = 28:
- The phases are:
 - 6A 4B 1C 1D
 - 5A 3B 1C 1D
 - •••
 - 4A 3B 1C 2D
- Total Buffer Size: 35
- Drastically beat best SAS result
 - Best SAS was 258
- Close to minimal amount (pull schedule)
 - Pull schedule was 26



CD-DAT Comparison: SAS vs Pull vs Phased

	Buffer Size	Schedule Size
SAS	258	4
Pull Schedule	26	251
Phased Schedule	35	52

- Apply technique hierarchically
- Children have several phases which all have to be executed
- Automatically supports cyclostatic filters
- Children pop/push less data, so can manage parent's buffer sizes more efficiently



- What if a Steady State of a component of a FeedbackLoop required more data than available?
- Single Appearance couldn't do separate compilation!
- Phased Scheduling can provide a fine-grained schedule, which will always allow separate compilation (if possible at all)

Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion
- Every Phase consumes as few items as possible to produce at least one data item
- Every Phase produces as many data items as possible
- Guarantees any schedulable program will be scheduled without deadlock
- Allows for separate compilation
- For details, see our paper

- Simple FeedbackLoop with a tight *delay* constraint
- Not possible to schedule using SAS
- Can schedule using Phased Scheduling
 - Use Minimal Latency Scheduling



 Minimal Latency Phased Schedule:



- Minimal Latency Phased Schedule:
 - join 2B 5split L



- Minimal Latency Phased Schedule:
 - join 2B 5split L
 - join 2B 5split L



- Minimal Latency Phased Schedule:
 - join 2B 5split L
 - join 2B 5split L
 - join 2B 5split L



- Minimal Latency Phased Schedule:
 - join 2B 5split L
 - join 2B 5split L
 - join 2B 5split L
 - join 2B 5split 2L



- Minimal Latency Phased Schedule:
 - join 2B 5split L
 - join 2B 5split L
 - join 2B 5split L
 - join 2B 5split 2L
- Can also be expressed as:
 - 3 {join 2B 5split L}
 - join 2B 5split 2L
- Common to have repeated Phases



Why not SAS?

- Naïve SAS schedule
 - 4join 8B 20split 5L:
 - Not valid because 4join consumes 20 data items
- Would like to form a loop-nest that includes join and L
- But multiplicity of executions of L and join have no common divisors



Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

Results

- SAS vs Minimal Latency
- Used 17 applications
 - 9 from our ASPLOS paper
 - 2 artificial benchmarks
 - 2 from Murthy99
 - Remaining 4 from our internal applications

Results - Buffer Size



Results – Schedule Size



Results - Combined



Overview

- General Stream Concepts
- StreamIt Details
- Program Steady State and Initialization
- Single Appearance and Pull Scheduling
- Phased Scheduling
 - Minimal Latency
- Results
- Related Work and Conclusion

Related Work

- Synchronous Data Flow (SDF)
- Ptolemy [Lee et al.]
- Many results for SAS on SDF
 - Memory Efficient Scheduling [Bhattacharyya97]
 - Buffer Merging [Murthy99]
- Cyclo-Static [Bilsen96]
- Peeking in US Navy Processing Graph Method [Goddard2000]
- Languages: LUSTRE, Esterel, Signal

Conclusion

Presented Phased Scheduling Algorithm

- Provides efficient interface for hierarchical scheduling
- Enables separate compilation with safety from deadlock
- Provides flexible buffer / schedule size trade-off
- Reduces latency of data throughput

Step towards a large scale hierarchical stream programming model

Phased Scheduling of Stream Programs

StreamIt Homepage

http://cag.lcs.mit.edu/streamit