An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design

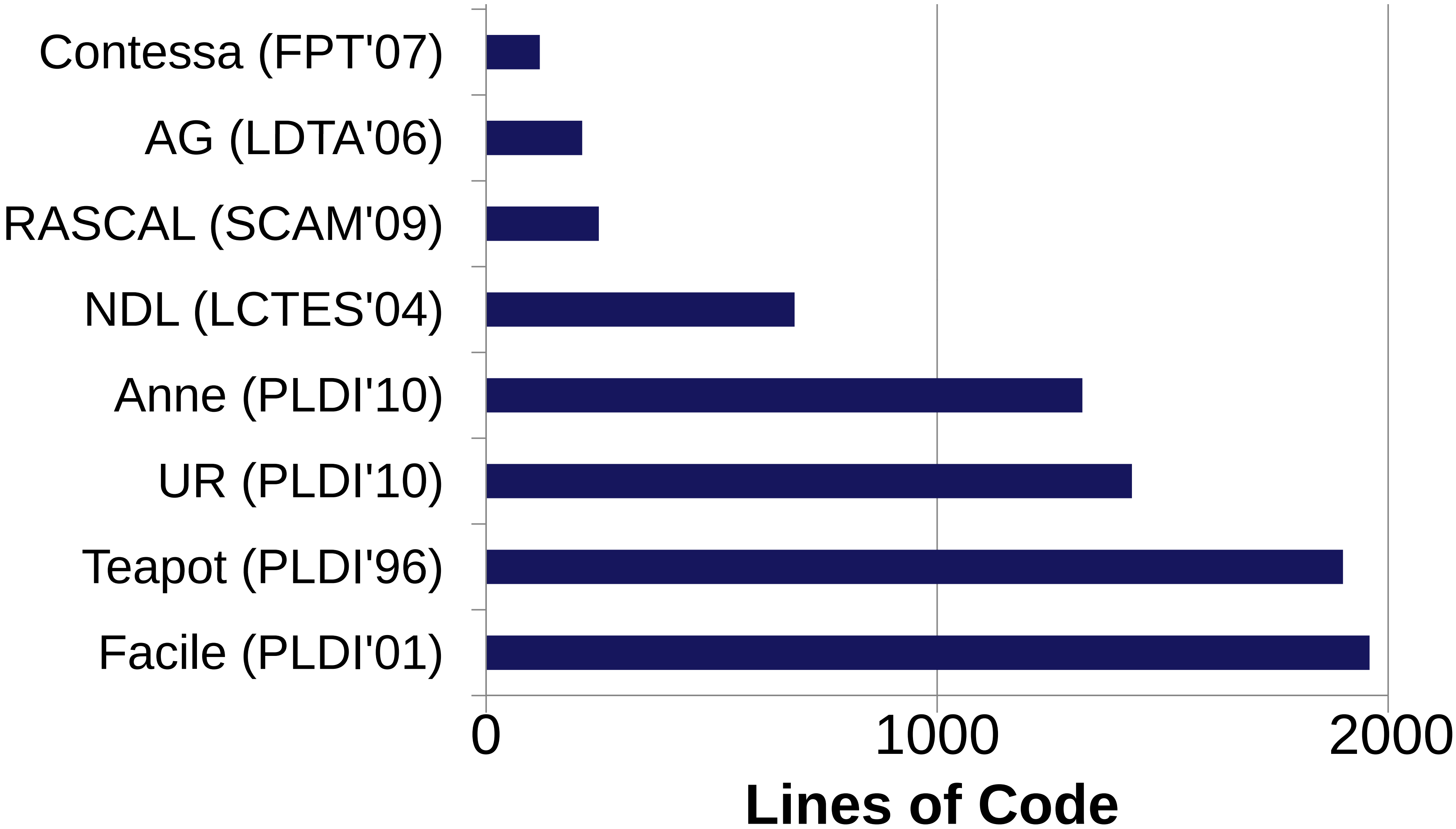
Bill Thies[1] and Saman Amarasinghe[2]

[1] Microsoft Research India

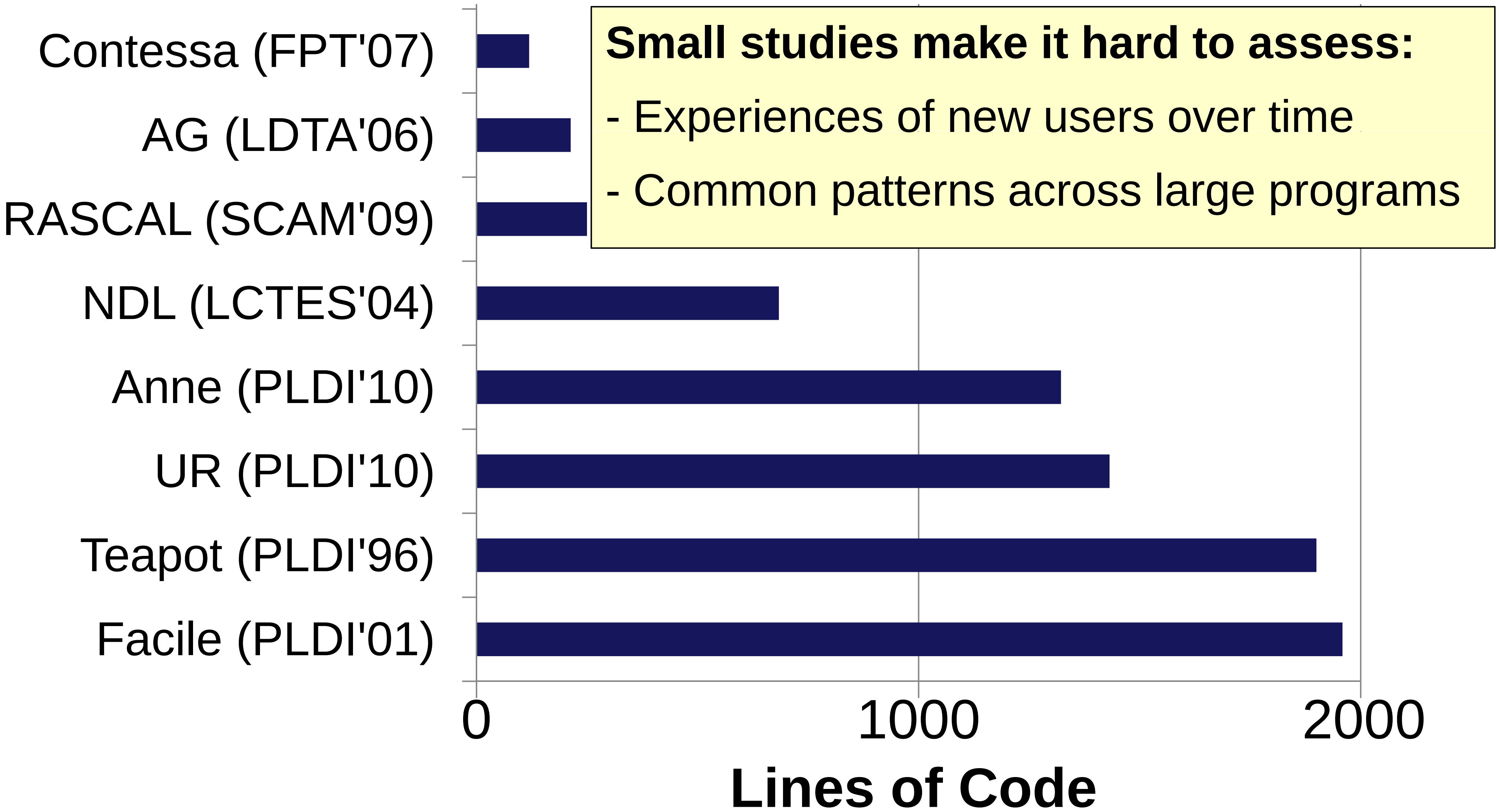
[2] Massachusetts Institute of Technology

PACT 2010

What Does it Take to Evaluate a New Language?



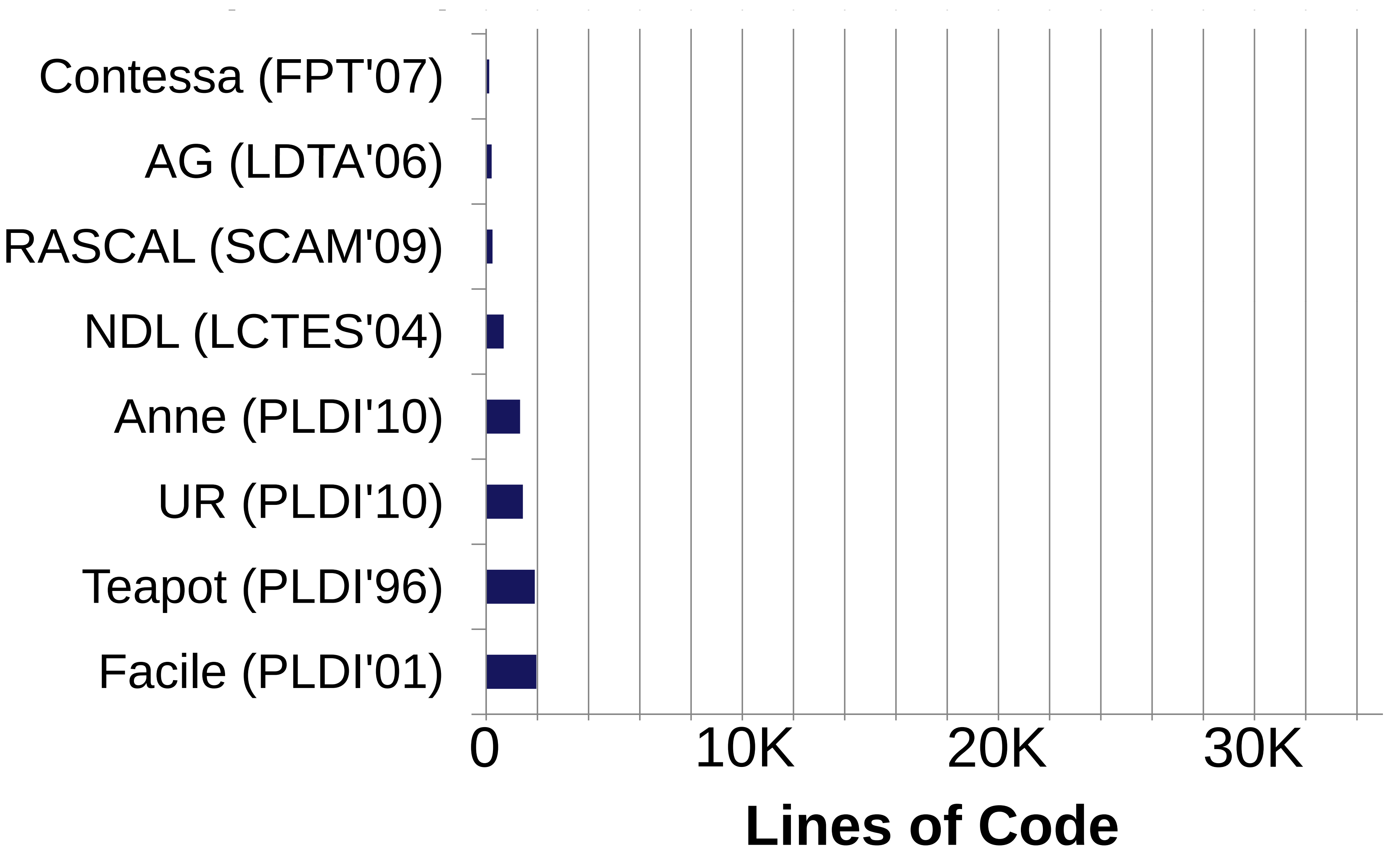
What Does it Take to Evaluate a New Language?



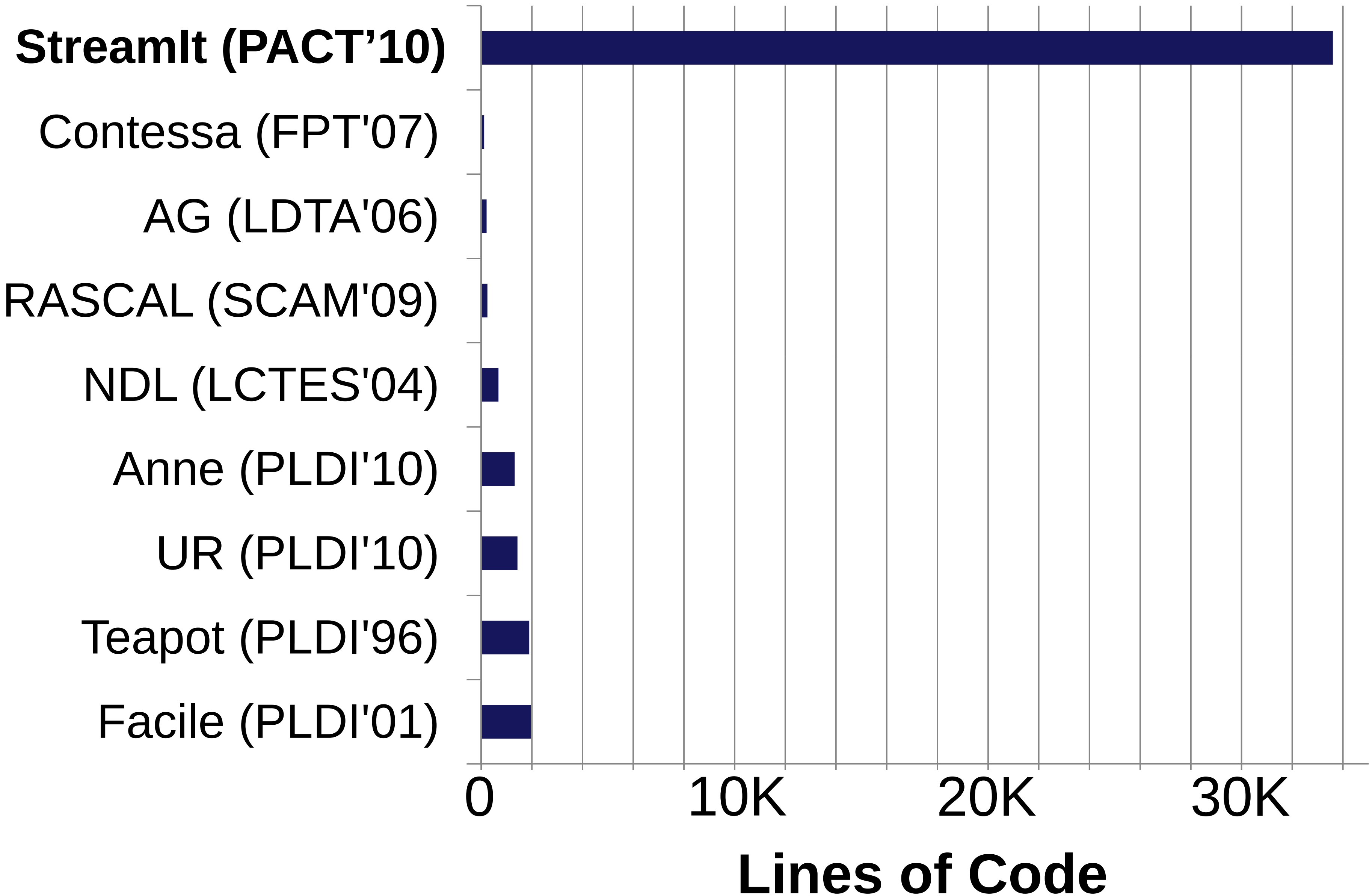
Small studies make it hard to assess:

- Experiences of new users over time
- Common patterns across large programs

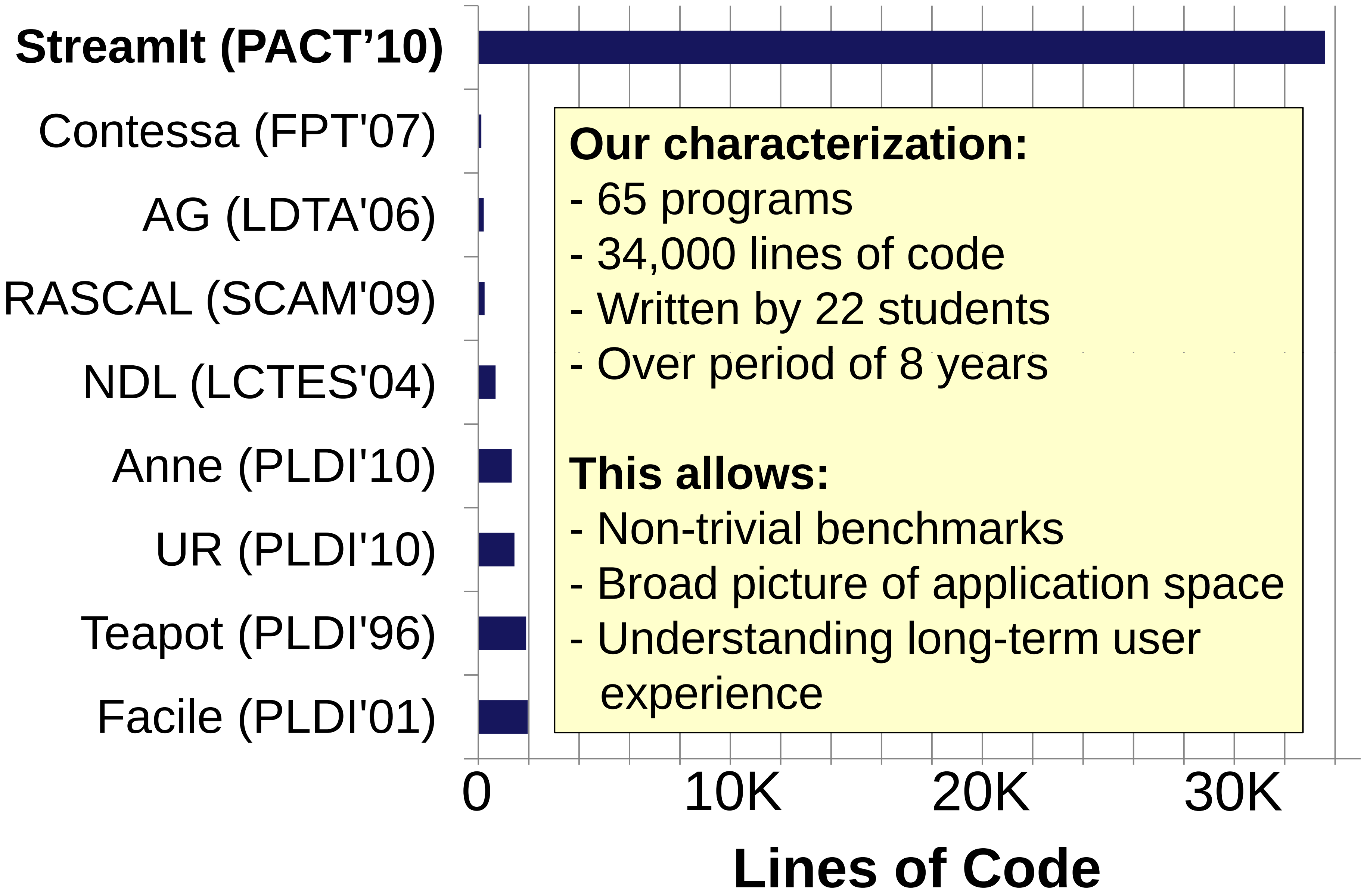
What Does it Take to Evaluate a New Language?



What Does it Take to Evaluate a New Language?



What Does it Take to Evaluate a New Language?

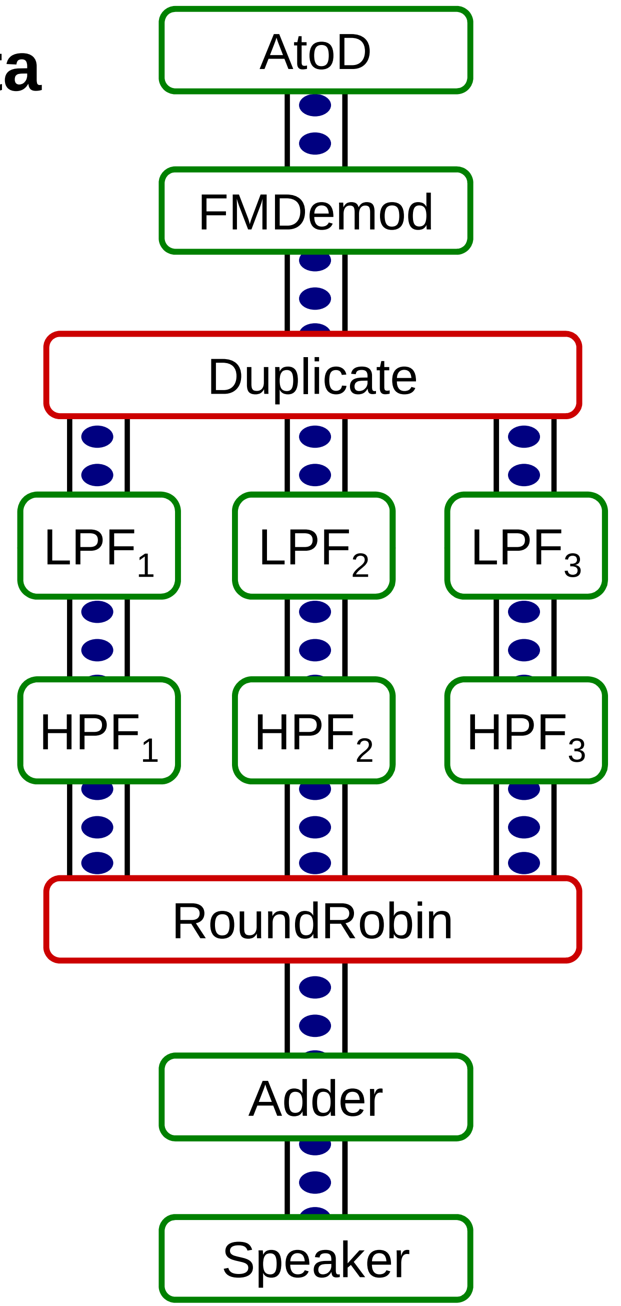


Our characterization:
- 65 programs
- 34,000 lines of code
- Written by 22 students
- Over period of 8 years

This allows:
- Non-trivial benchmarks
- Broad picture of application space
- Understanding long-term user experience

Streaming Application Domain

- **For programs based on streams of data**
  - Audio, video, DSP, networking, and cryptographic processing kernels
  - Examples: HDTV editing, radar tracking, microphone arrays, cell phone base stations, graphics
- **Properties of stream programs**
  - Regular and repeating computation
  - Independent filters with explicit communication

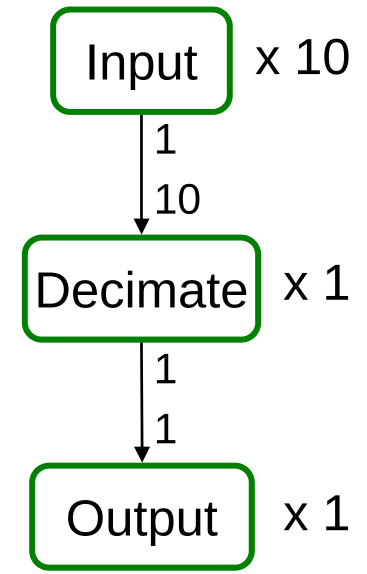


StreamIt: A Language and Compiler for Stream Programs

- **Key idea: design language that enables static analysis**
- **Goals:**
  1. Improve programmer productivity in the streaming domain
  2. Expose and exploit the parallelism in stream programs
- **Project contributions:**
  - Language design for streaming [CC'02, CAN'02, PPoPP'05, IJPP'05]
  - Automatic parallelization [ASPLOS'02, G.Hardware'05, ASPLOS'06, MIT’10]
  - Domain-specific optimizations [PLDI'03, CASES'05, MM'08]
  - Cache-aware scheduling [LCTES'03, LCTES'05]
  - Extracting streams from legacy code [MICRO'07]
  - User + application studies [PLDI'05, P-PHEC'05, IPDPS'06]

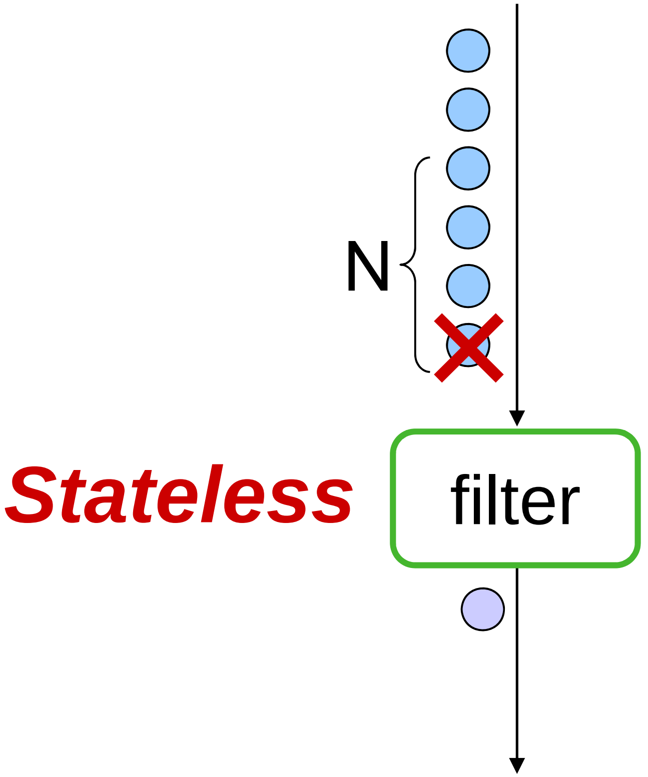
StreamIt Language Basics

- **High-level, architecture-independent language**
  - Backend support for uniprocessors, multicores (Raw, SMP), cluster of workstations
- **Model of computation: synchronous dataflow** [Lee & Messerschmidt, 1987]
  - Program is a graph of independent *filters*
  - Filters have an atomic execution step with known input / output rates
  - Compiler is responsible for scheduling and buffer management
- **Extensions to synchronous dataflow**
  - Dynamic I/O rates
  - Support for sliding window operations
  - Teleport messaging [PPoPP'05]



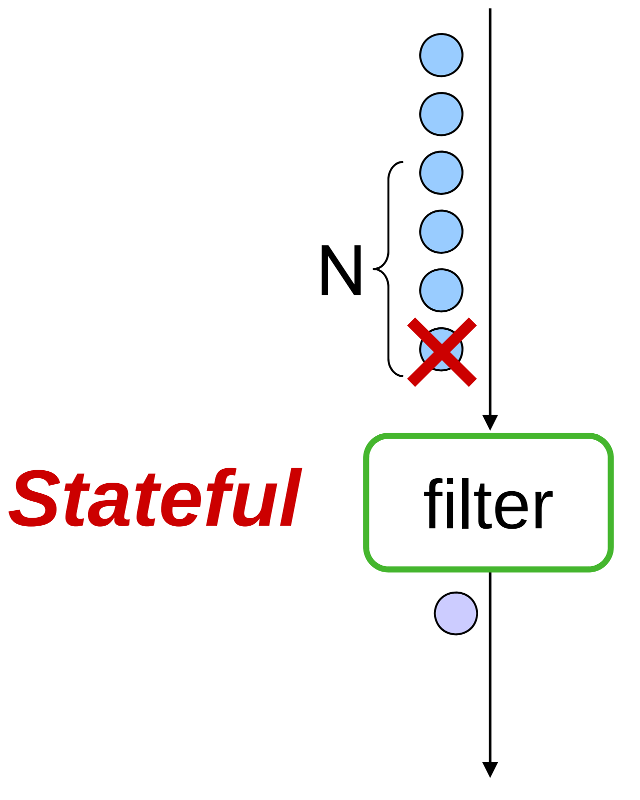
Example Filter: Low Pass Filter

```
float->float filter LowPassFilter (int N, float[N] weights ) {

  work peek N push 1 pop 1 {
    float result = 0;

    for (int i=0; i<weights.length; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```

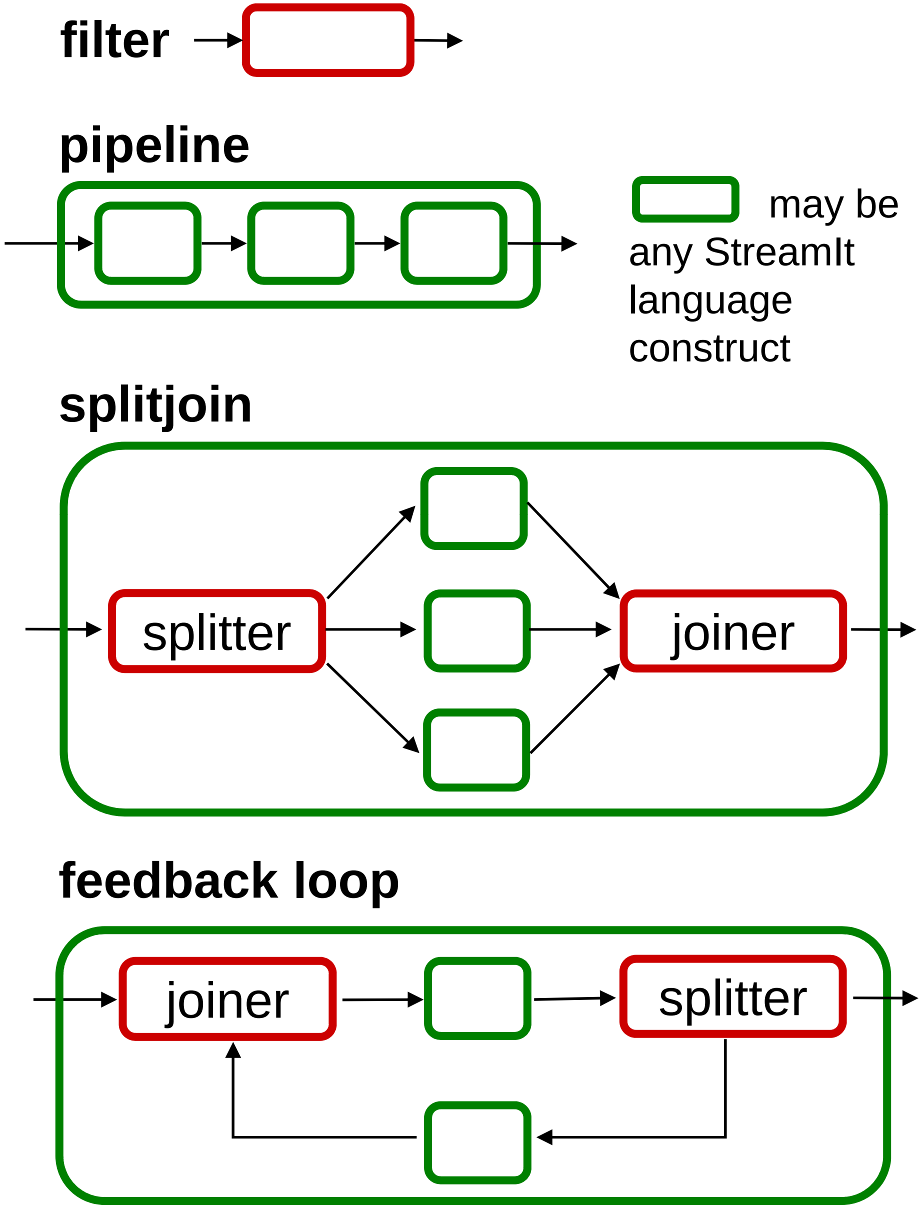


Example Filter: Low Pass Filter

```
float->float filter LowPassFilter (int N ) {
    float[N] weights;
    work peek N push 1 pop 1 {
        float result = 0;
        weights = adaptChannel();
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



Structured Streams



- **Each structure is single-input, single-output**
- **Hierarchical and composable**

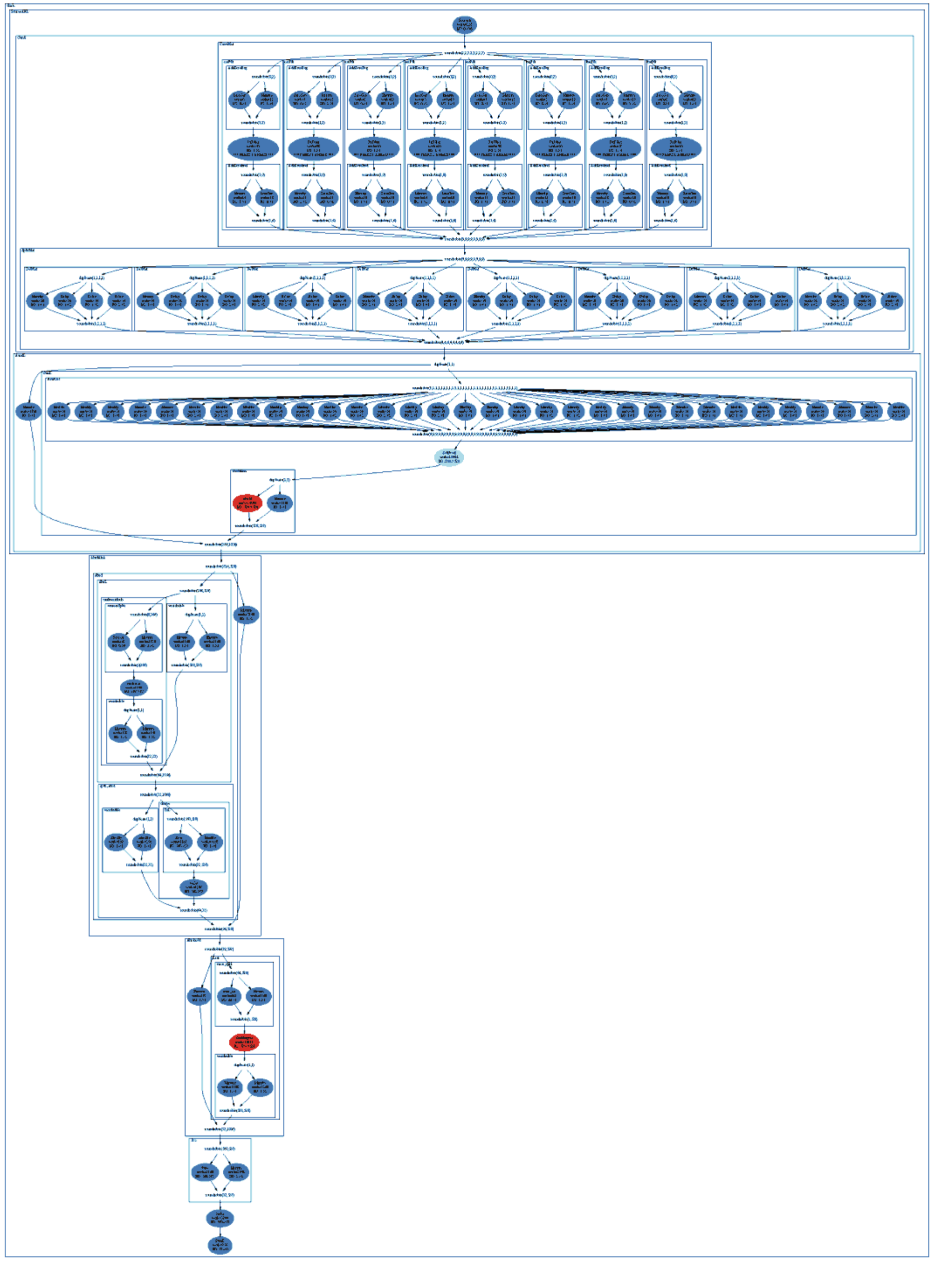
StreamIt Benchmark Suite (1/2)

- **Realistic applications (30):**
  - MPEG2 encoder / decoder
  - Ground Moving Target Indicator
  - Mosaic
  - MP3 subset
  - Medium Pulse Compression Radar
  - JPEG decoder / transcoder
  - Feature Aided Tracking
  - HDTV
  - H264 subset
  - Synthetic Aperture Radar
  - GSM Decoder
  - 802.11a transmitte
  - DES encryption
  - Serpent encryption
  - Vocoder
  - RayTracer
  - 3GPP physical layer
  - Radar Array Front End
  - Freq-hopping radio
  - Orthogonal Frequency Division Multiplexer
  - Channel Vocoder
  - Filterbank
  - Target Detector
  - FM Radio
  - DToA Converter

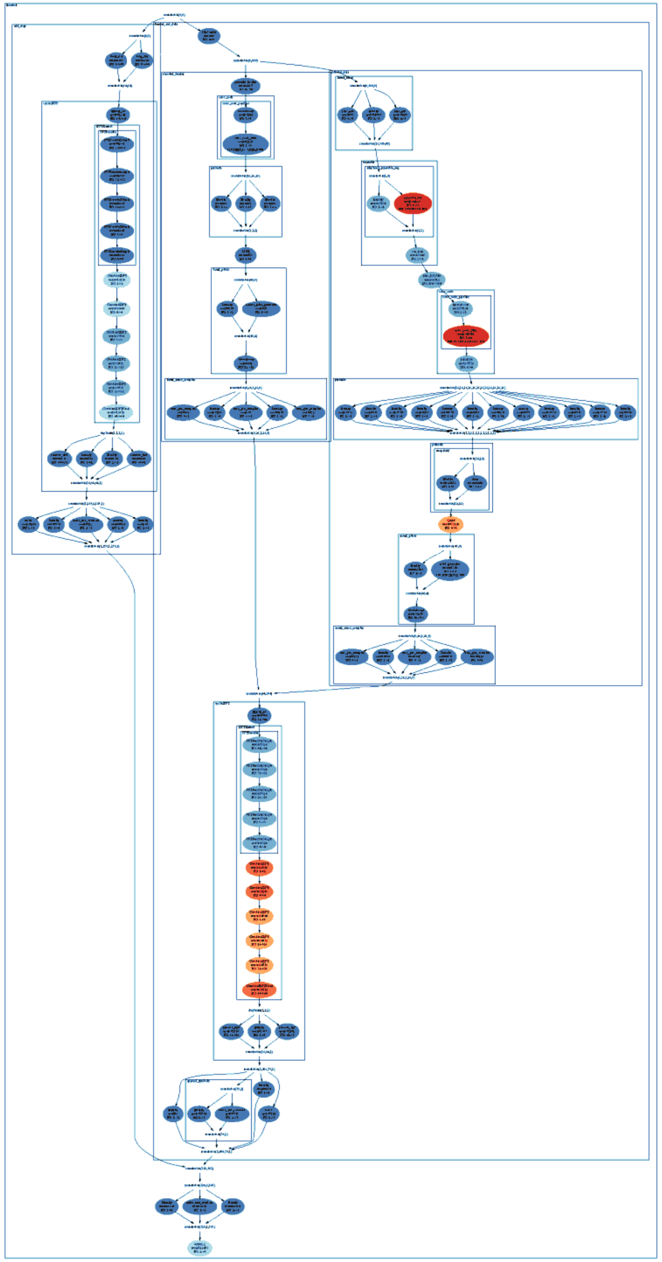
StreamIt Benchmark Suite (2/2)

- **Libraries / kernels (23):**
  - Autocorrelation
  - Cholesky
  - CRC
  - DCT (1D / 2D, float / int)
  - FFT (4 granularities)
  - Lattice
  - Matrix Multiplication
  - Oversampler
  - Rate Convert
  - Time Delay Equalization
  - Trellis
  - VectAdd
- **Graphics pipelines (4):**
  - Reference pipeline
  - Phong shading
  - Shadow volumes
  - Particle system
- **Sorting routines (8)**
  - Bitonic sort (3 versions)
  - Bubble Sort
  - Comparison counting
  - Insertion sort
  - Merge sort
  - Radix sort

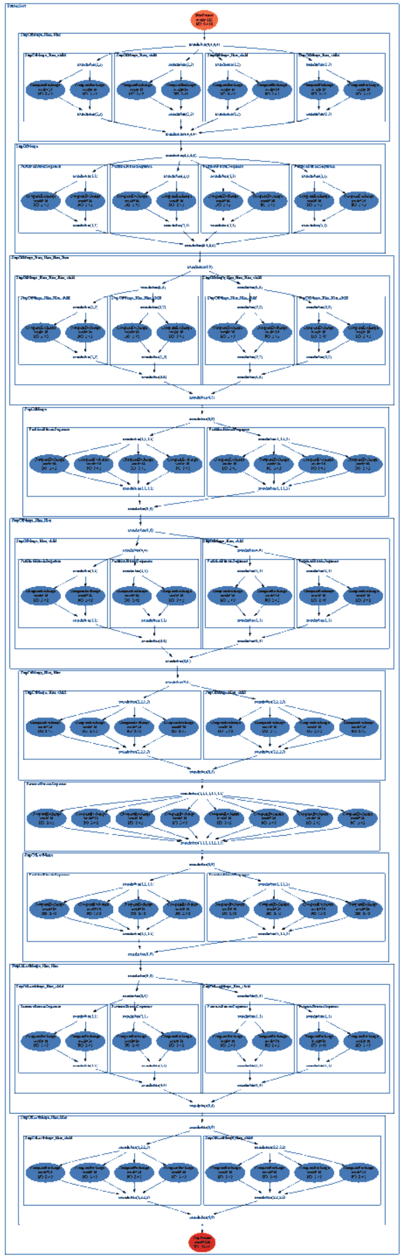
3GPP



802.11a



Bitonic Sort



Note to online viewers:

For high-resolution stream graphs of all benchmarks,

please see pp. 173-240 of this thesis:

http://groups.csail.mit.edu/commit/papers/09/thies-phd-thesis.pdf

Characterization Overview

- **Focus on architecture-independent features**
  - Avoid performance artifacts of the StreamIt compiler
  - Estimate execution time statically (not perfect)
- **Three categories of inquiry:**
  1. Throughput bottlenecks
  2. Scheduling characteristics
  3. Utilization of StreamIt language features

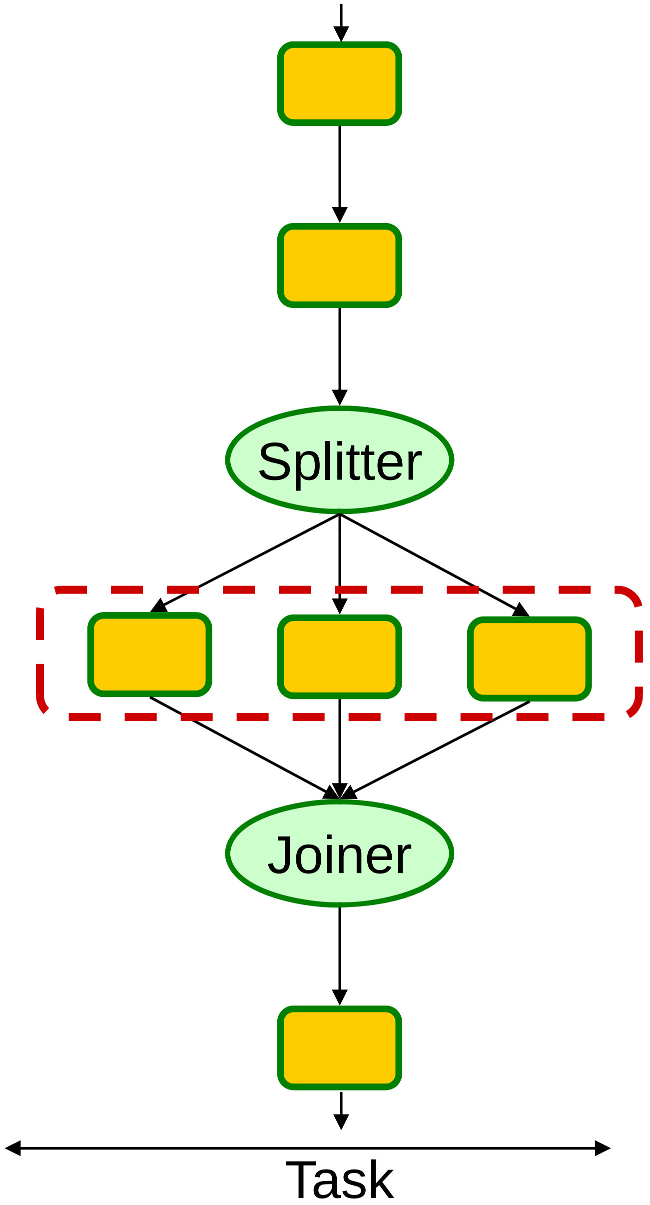
Lessons Learned from the StreamIt Language

What we did right

What we did wrong

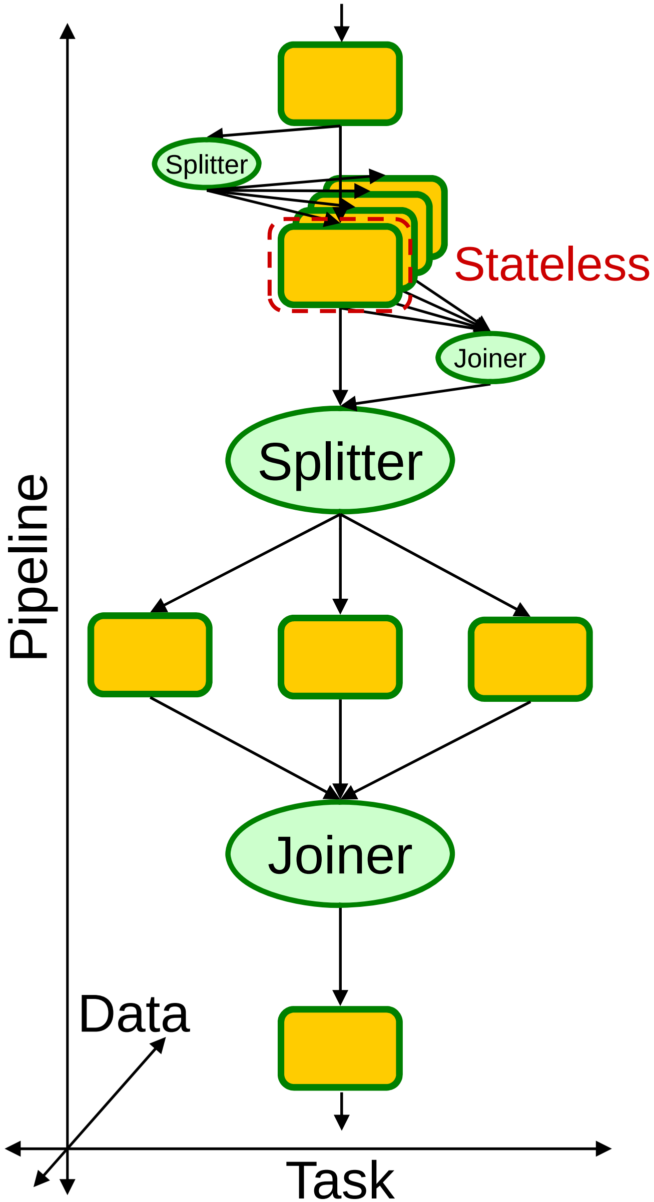
Opportunities for doing better

1. Expose Task, Data, & Pipeline Parallelism



Task parallelism

1. Expose Task, Data, & Pipeline Parallelism

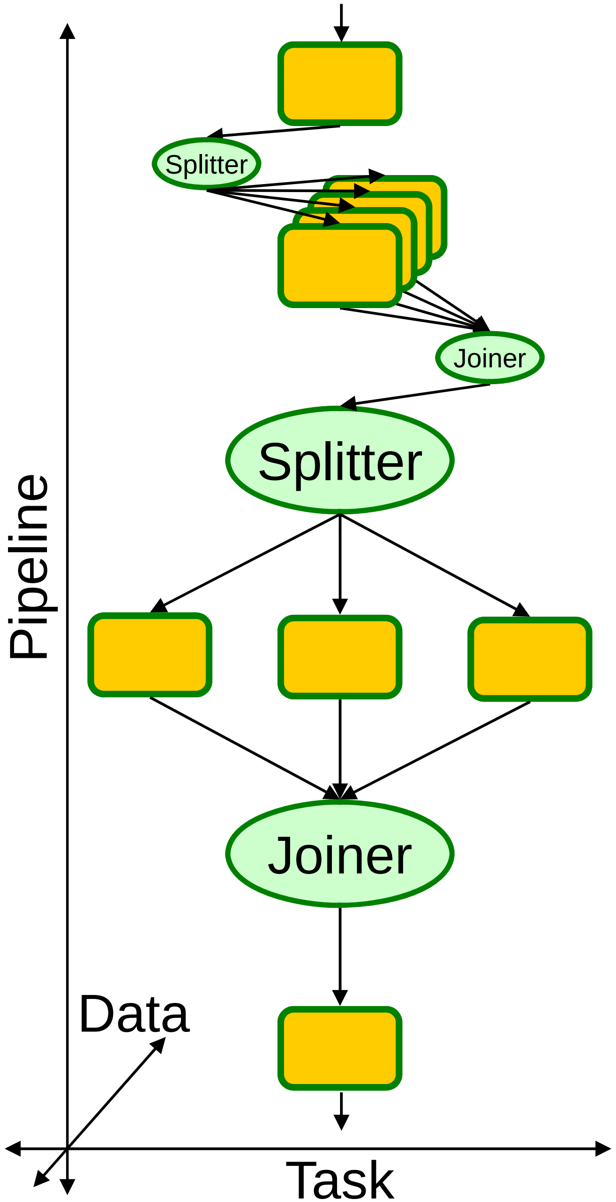


Data parallelism

Task parallelism

Pipeline parallelism

1. Expose Task, Data, & Pipeline Parallelism



Data parallelism

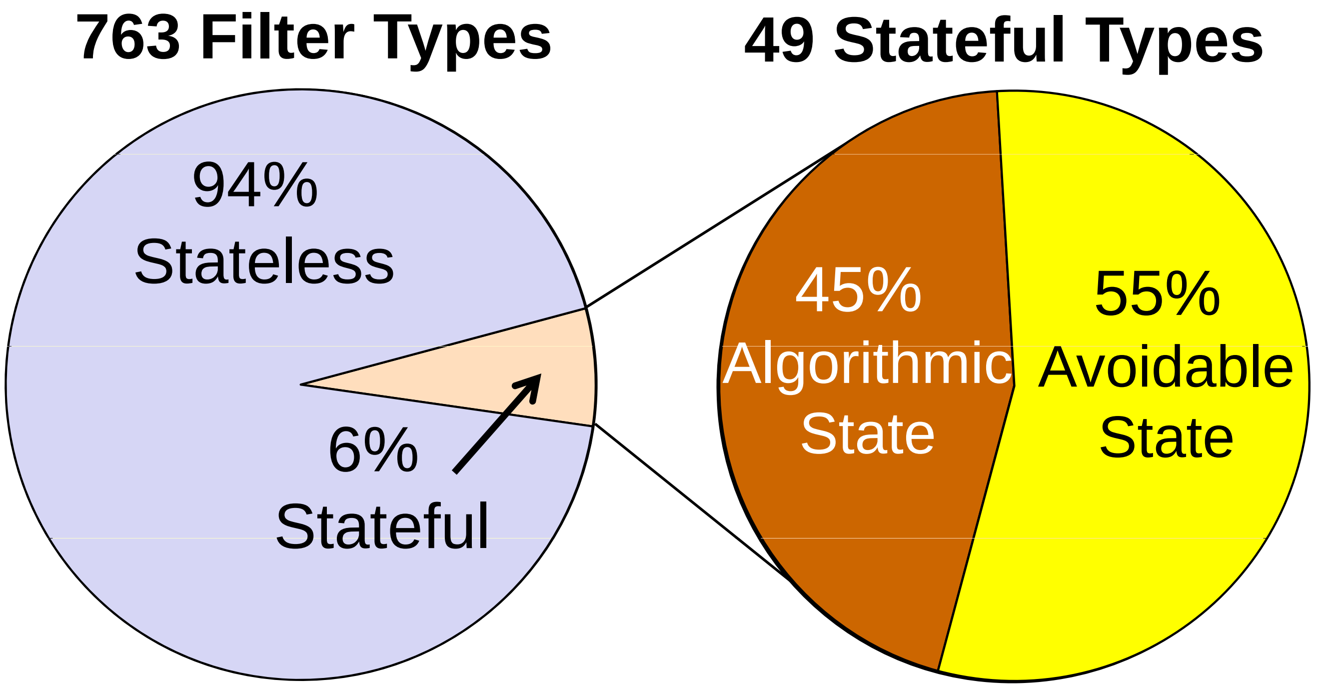
- 74% of benchmarks contain entirely data-parallel filters
- In other benchmarks, 5% to 96% (median 71%) of work is data-parallel

Task parallelism

- 82% of benchmarks contain at least one splitjoin
- Median of 8 splitjoins per benchmark

Pipeline parallelism

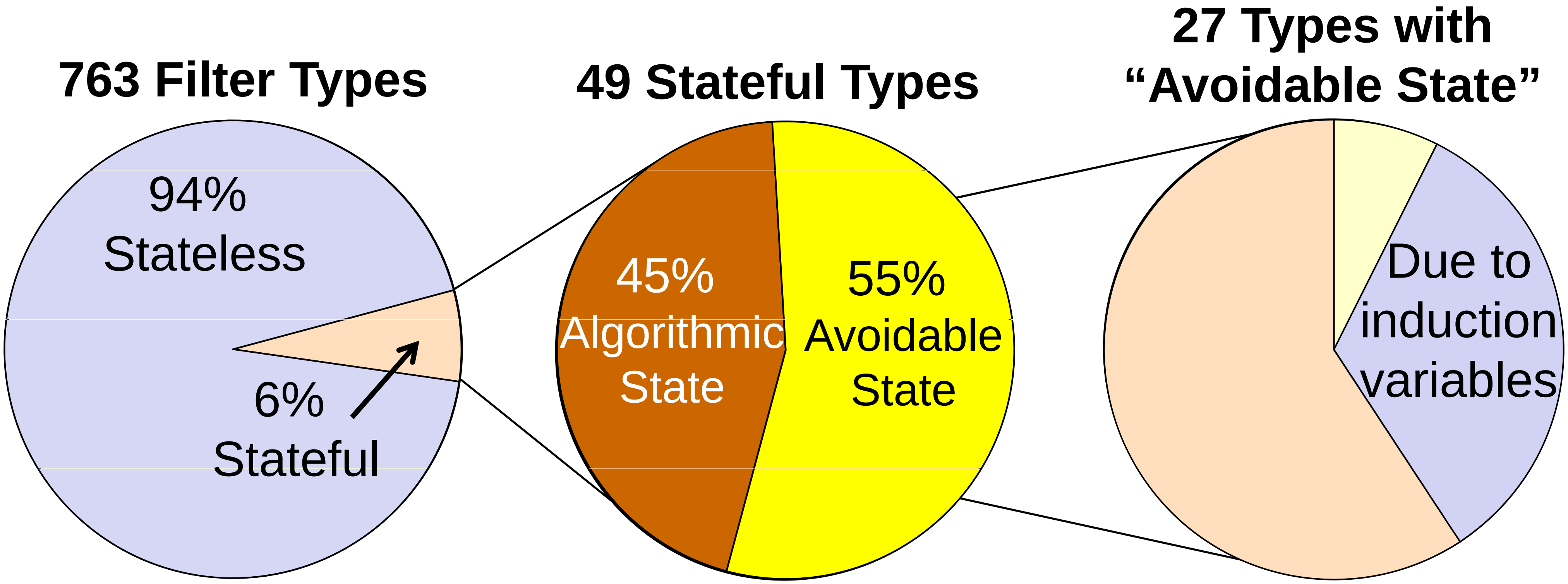
Characterizing Stateful Filters



Sources of Algorithmic State

- **MPEG2:** bit-alignment, reference frame encoding, motion prediction, …
- **HDTV:** Pre-coding and Ungerboeck encoding
- **HDTV + Trellis:** Ungerboeck decoding
- **GSM:** Feedback loops
- **Vocoder:** Accumulator, adaptive filter, feedback loop
- **OFDM:** Incremental phase correction
- **Graphics pipelines:** persistent screen buffers

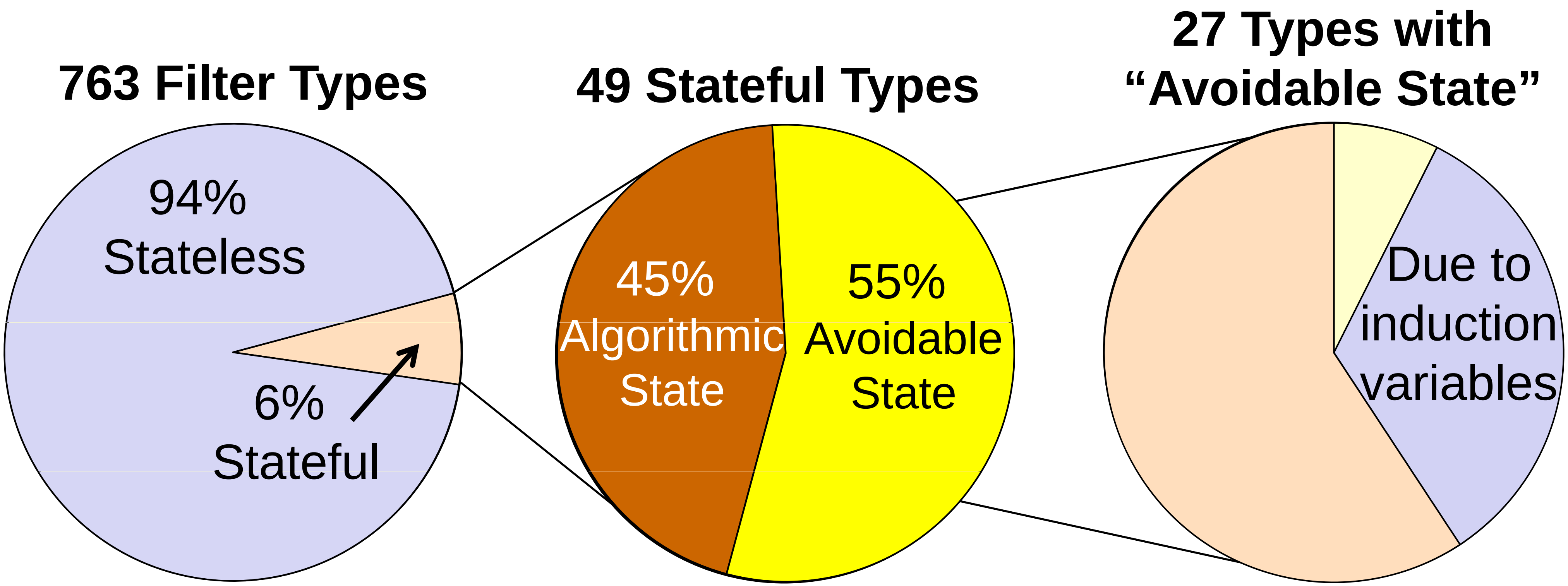
Characterizing Stateful Filters



Sources of Algorithmic State

- **MPEG2:** bit-alignment, reference frame encoding, motion prediction, …
- **HDTV:** Pre-coding and Ungerboeck encoding
- **HDTV + Trellis:** Ungerboeck decoding
- **GSM:** Feedback loops
- **Vocoder:** Accumulator, adaptive filter, feedback loop
- **OFDM:** Incremental phase correction
- **Graphics pipelines:** persistent screen buffers

2. Eliminate Stateful Induction Variables

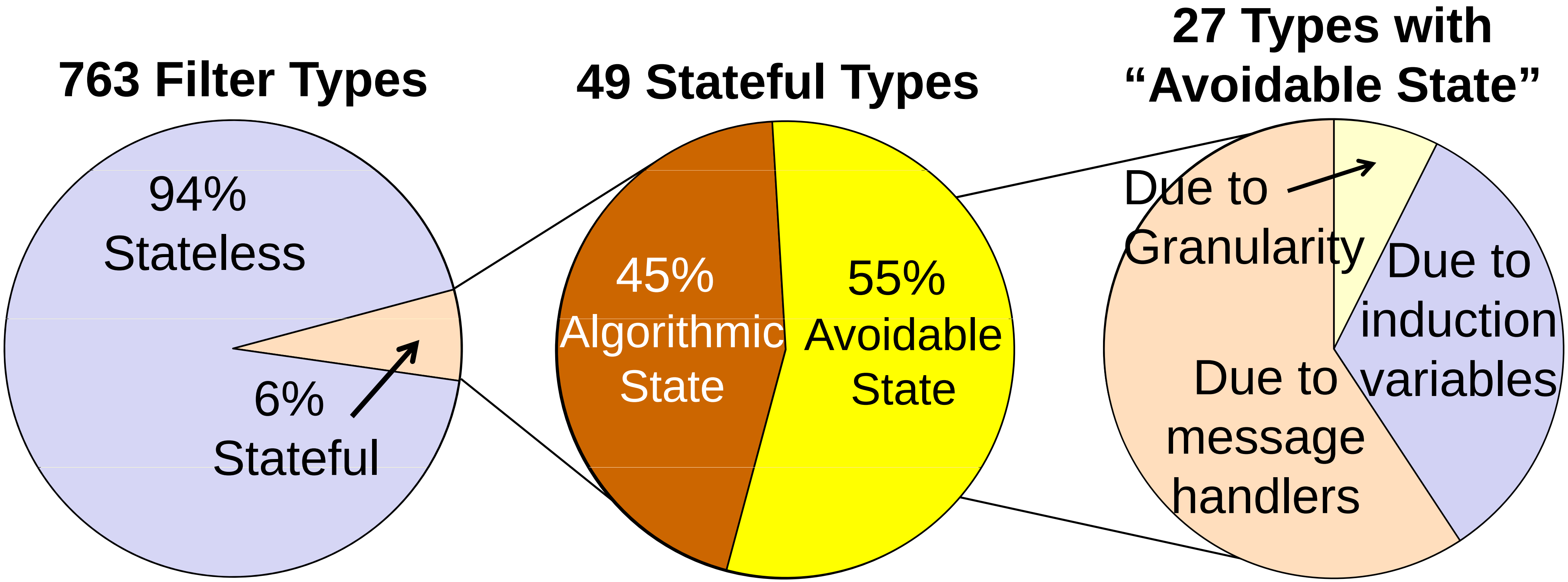


Sources of Induction Variables

- **MPEG encoder:** counts frame # to assign picture type
- **MPD / Radar:** count position in logical vector for FIR
- **Trellis:** noise source flips every N items
- **MPEG encoder / MPD:** maintain logical 2D position (row/column)
- **MPD:** reset accumulator when counter overflows

Opportunity: Language primitive to return current iteration?

2. Eliminate Stateful Induction Variables

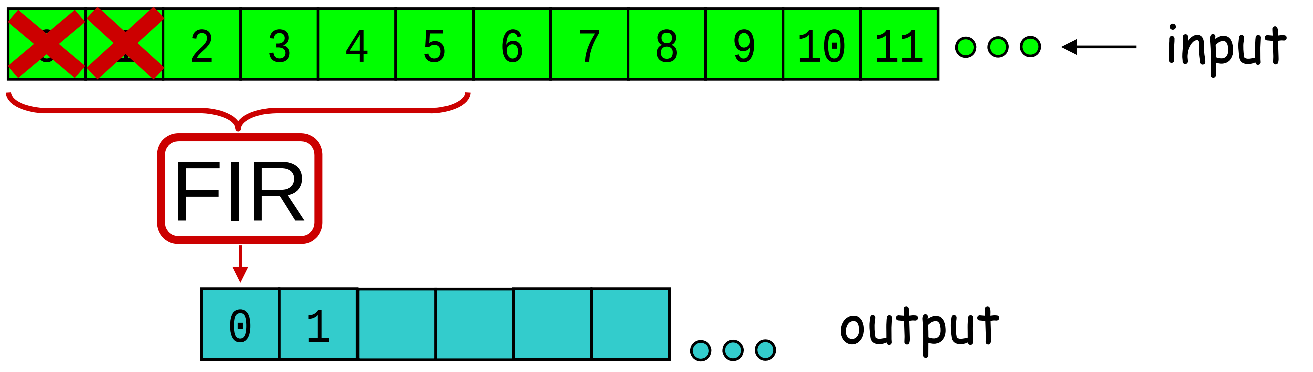


Sources of Induction Variables

- **MPEG encoder:** counts frame # to assign picture type
- **MPD / Radar:** count position in logical vector for FIR
- **Trellis:** noise source flips every N items
- **MPEG encoder / MPD:** maintain logical 2D position (row/column)
- **MPD:** reset accumulator when counter overflows

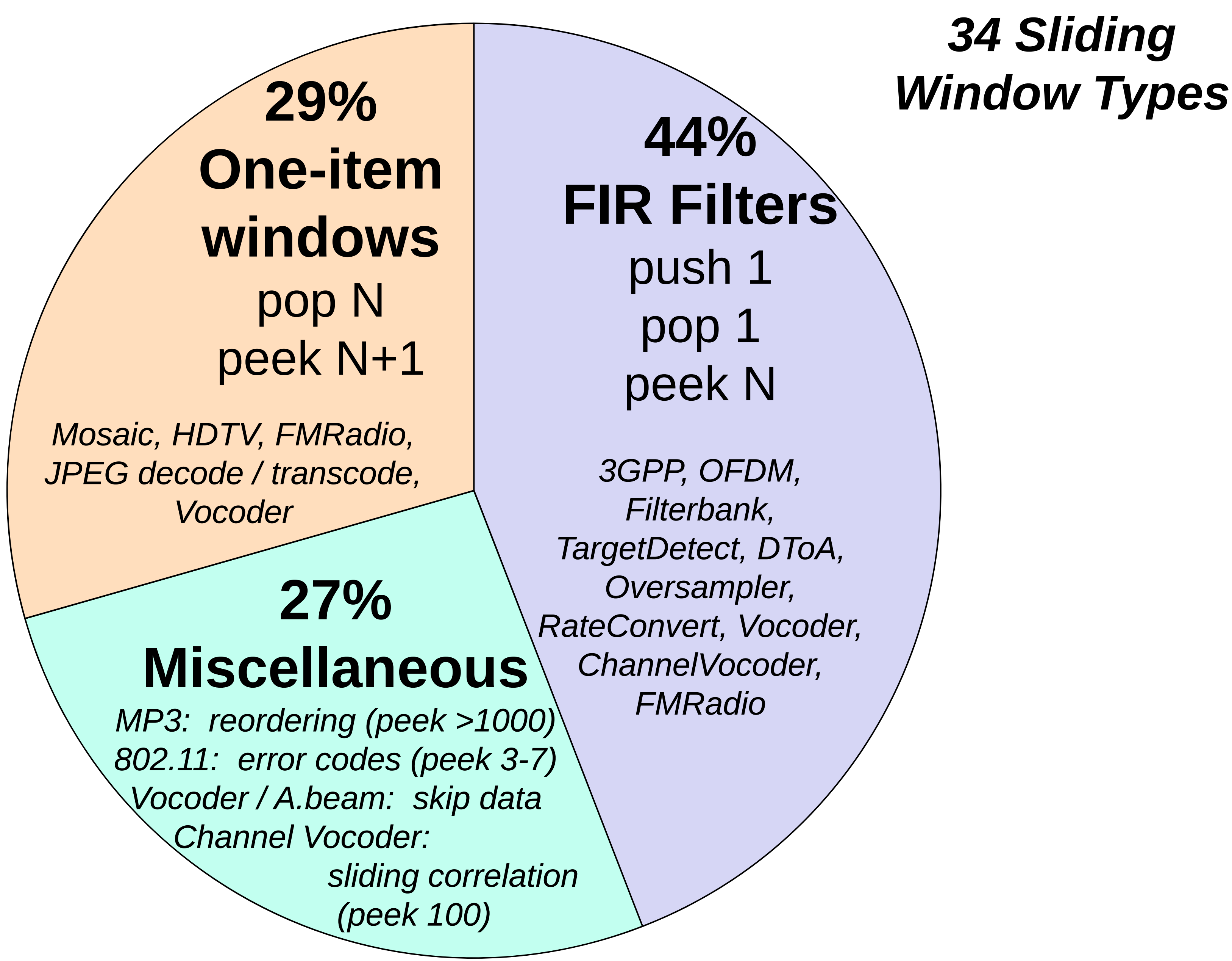
Opportunity: Language primitive to return current iteration?

3. Expose Parallelism in Sliding Windows



- **Legacy codes obscure parallelism in sliding windows**
  - In von-Neumann languages, modulo functions or copy/shift operations prevent detection of parallelism in sliding windows
- **Sliding windows are prevalent in our benchmark suite**
  - 57% of realistic applications contain at least one sliding window
  - Programs with sliding windows have 10 instances on average
  - Without this parallelism, 11 of our benchmarks would have a new throughput bottleneck (work: 3% - 98%, median 8%)

Characterizing Sliding Windows



4. Expose Startup Behaviors

- **Example: difference encoder** (JPEG, Vocoder)

```
int->int filter Diff_Encoder() {
  int state = 0;

  work push 1 pop 1 {
    push(peek(0) – state);
    state = pop();
  }
}
```

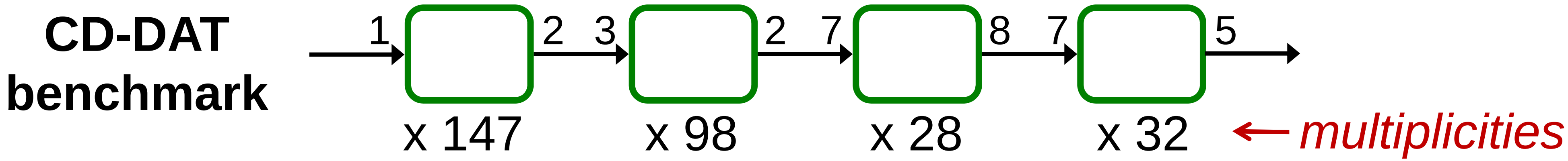
Stateful

```
int->int filter Diff_Encoder() {

  prework push 1 pop 1 {
    push(peek(0));
  }

  work push 1 pop 1 peek 2 {
    push(peek(1) – peek(0));
    pop();
  }
}
```

Stateless

- **Required by 15 programs:**
  - For **delay:** MPD, HDTV, Vocoder, 3GPP, Filterbank, DToA, Lattice, Trellis, GSM, CRC
  - For **picture reordering** (MPEG)
  - For **initialization** (MPD, HDTV, 802.11)
  - For **difference encoder or decoder:** JPEG, Vocoder

5. Surprise:
Mis-Matched Data Rates Uncommon

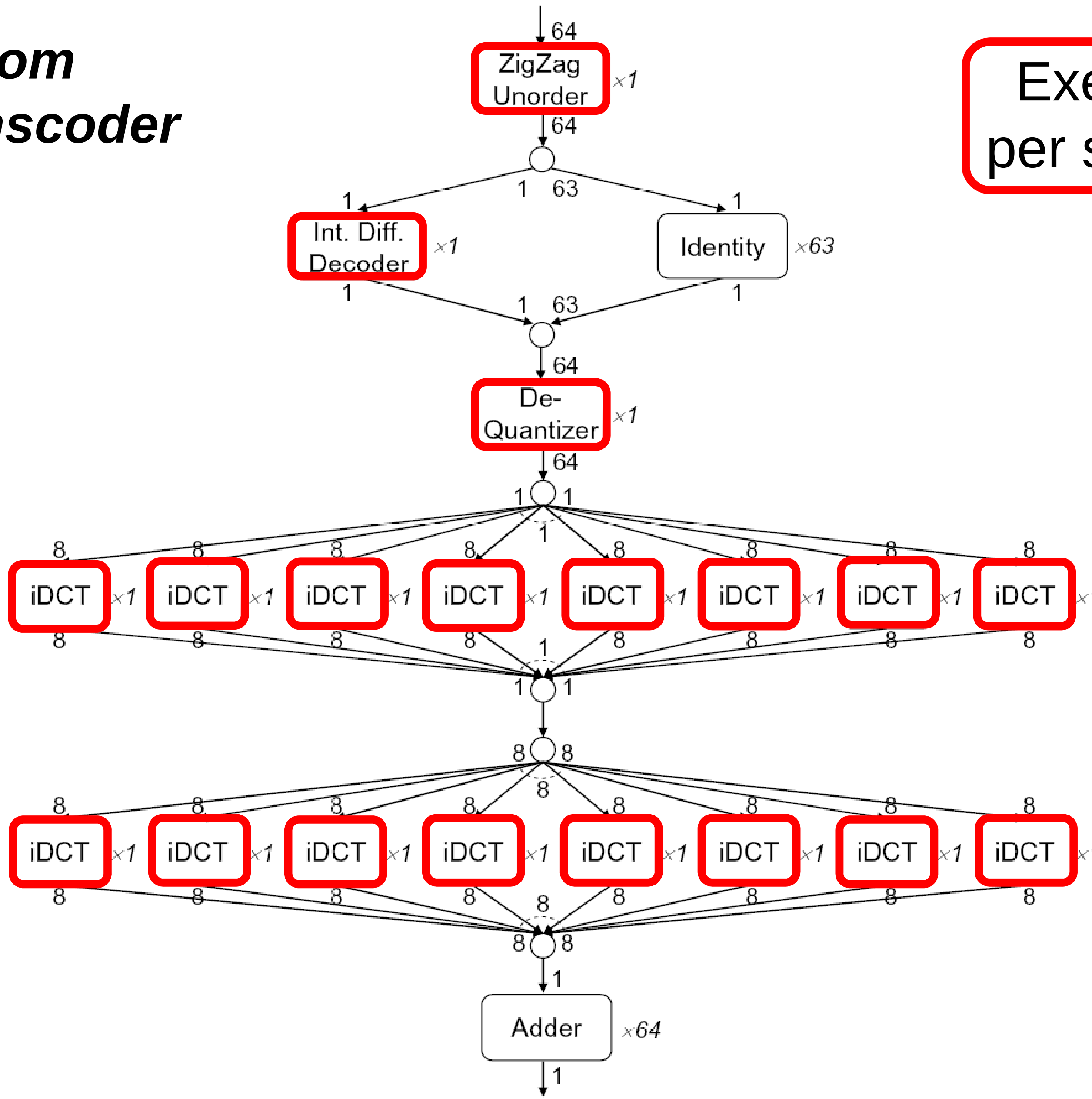


Converts CD audio (44.1 kHz) to digital audio tape (48 kHz)

- **This is a driving application in many papers**
  - Eg: [MBL94] [TZB99] [BB00] [BML95] [CBL01] [MB04] [KSB08]
  - Due to large filter multiplicities, clever scheduling is needed to control code size, buffer size, and latency
- **But are mis-matched rates common in practice? No!**

5. Surprise:
Mis-Matched Data Rates Uncommon

Excerpt from JPEG transcoder



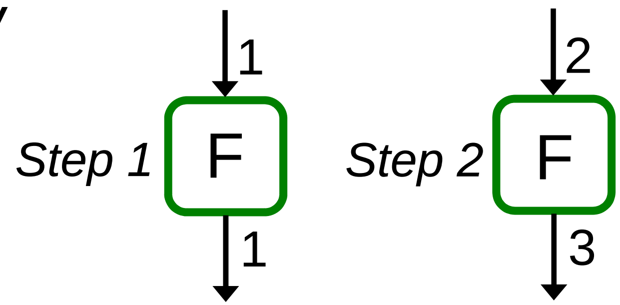
Execute once per steady state

Characterizing Mis-Matched Data Rates

- **In our benchmark suite:**
  - 89% of programs have a filter with a multiplicity of 1
  - On average, 63% of filters share the same multiplicity
  - For 68% of benchmarks, the most common multiplicity is 1
- **Implication for compiler design:**
  Do not expect advanced buffering strategies to have a large impact on average programs
  - Example: Karczmarek, Thies, & Amarasinghe, LCTES'03
  - Space saved on CD-DAT: 14x
  - Space saved on other programs (median): 1.2x

6. Surprise: Multi-Phase Filters Cause More Harm than Good

- **A multi-phase filter divides its execution into many steps**
  - Formally known a *cyclo-static dataflow*
  - Possible benefits:
    - Shorter latencies
    - More natural code



- **We implemented multi-phase filters, and we regretted it**
  - Programmers did not understand the difference between a phase of execution, and a normal function call
  - Compiler was complicated by presences of phases
- **However, phases proved important for splitters / joiners**
  - Routing items needs to be done with minimal latency
  - Otherwise buffers grow large, and deadlock in one case (GSM)

7. Programmers Introduce Unnecessary State in Filters

- **Programmers do not implement things how you expect**

```
void->int filter SquareWave() {
   work push 2 {
      push(0);
      push(1);
   }
}
```

Stateless

```
void->int filter SquareWave() {
   int x = 0;

   work push 1 {
      push(x);
      x = 1 - x;
   }
}
```

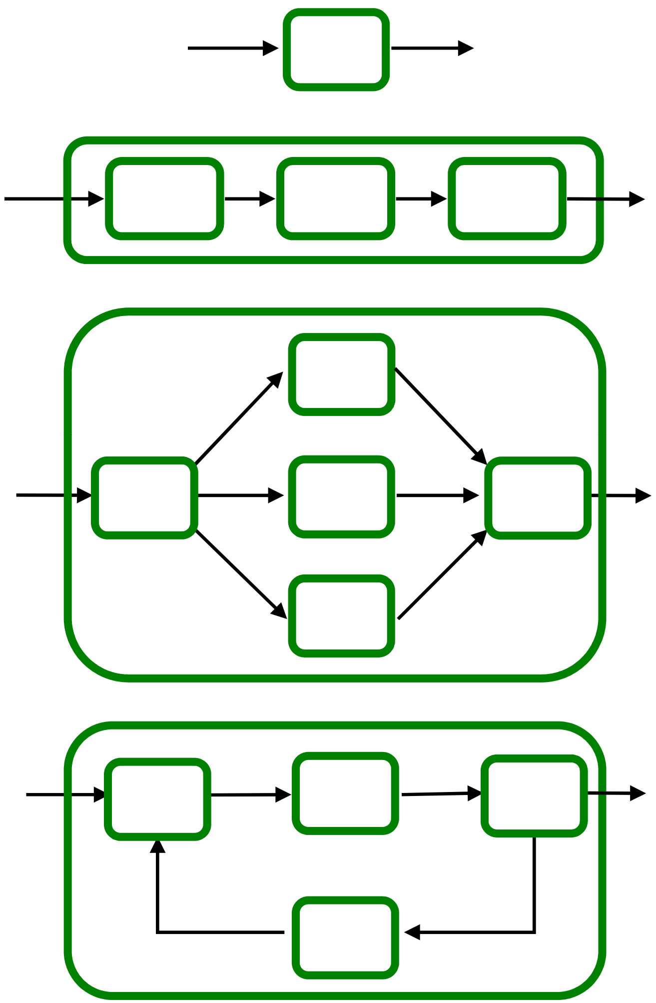
Stateful

- **Opportunity: add a "stateful" modifier to filter decl?**
  – Require programmer to be cognizant of the cost of state

8. Leverage and Improve Upon Structured Streams

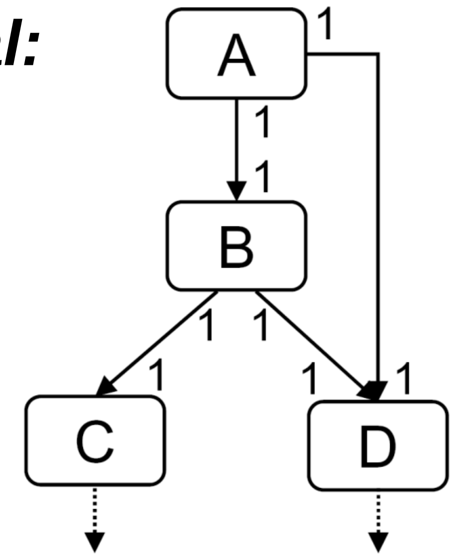
- **Overall, programmers found it useful and tractable to write programs using structured streams**
  - Syntax is simple to write, easy to read

- **However, structured streams are occasionally unnatural**
  - And, in rare cases, insufficient

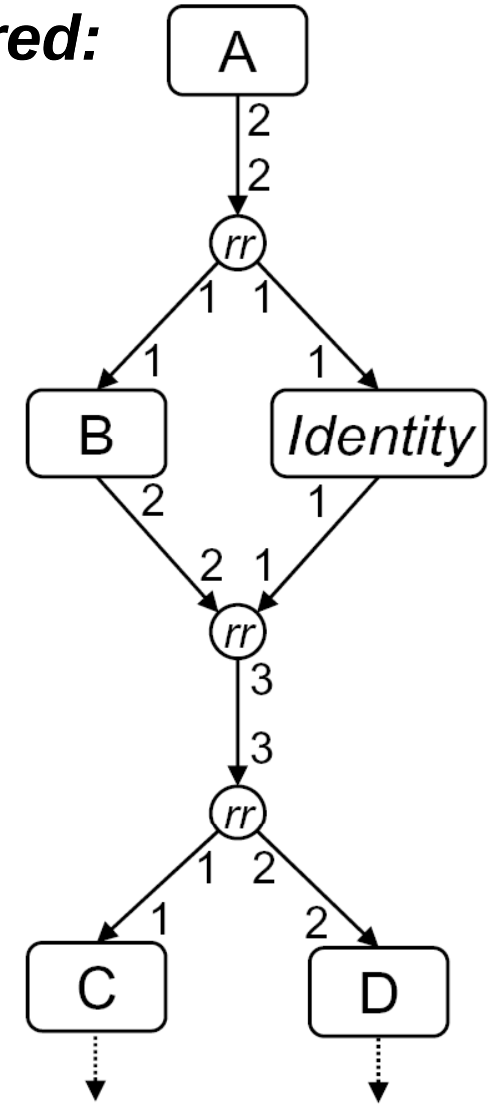


8. Leverage and Improve Upon Structured Streams

Original:



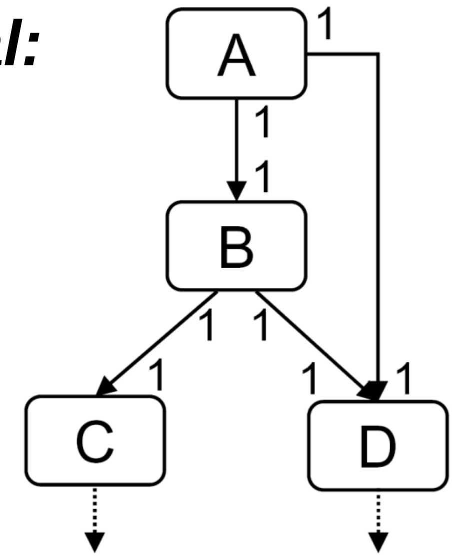
Structured:



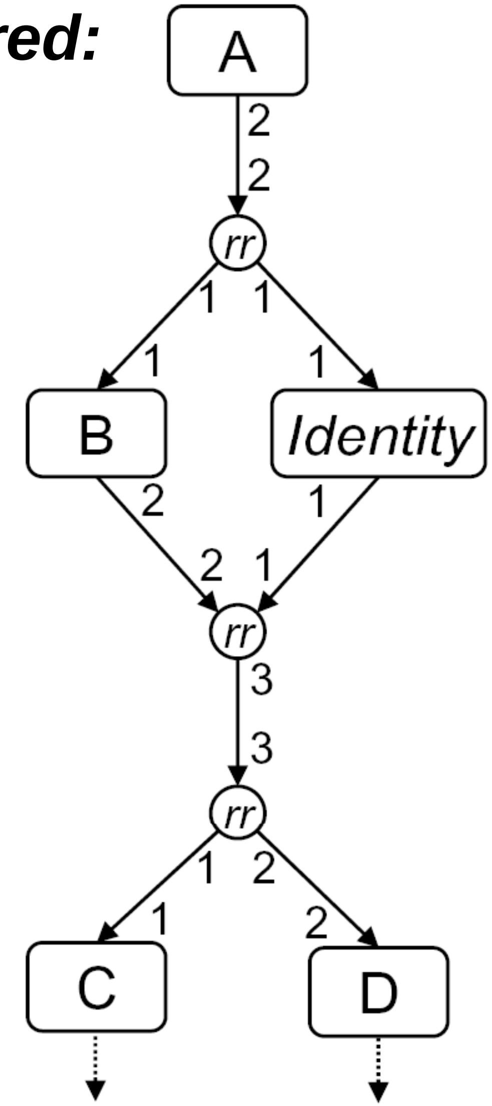
Compiler recovers unstructured graph using *synchronization removal* [Gordon 2010]

8. Leverage and Improve Upon Structured Streams

Original:



Structured:



- **Characterization:**
  - 49% of benchmarks have an Identity node
  - In those benchmarks, Identities account for 3% to 86% (median 20%) of instances
- **Opportunity:**
  - Bypass capability (ala GOTO) for streams

Related Work

- **Benchmark suites in von-Neumann languages often include stream programs, but lose high-level properties**
  - MediaBench
  - ALPBench
  - Berkeley MM Workload
  - HandBench
  - MiBench
  - NetBench
  - SPEC
  - PARSEC
  - Perfect Club
- **Brook language includes 17K LOC benchmark suite**
  - Brook disallows stateful filters; hence, more data parallelism
  - Also more focus on dynamic rates & flexible program behavior
- **Other stream languages lack benchmark characterization**
  - StreamC / KernelC
  - Cg
  - Baker
  - SPUR
  - Spidle
- **In-depth analysis of 12 StreamIt "core" benchmarks published concurrently to this paper** [Gordon 2010]

Conclusions

- **First characterization of a streaming benchmark suite that was written in a stream programming language**
  - 65 programs; 22 programmers; 34 KLOC
- **Implications for streaming languages and compilers:**
  - **DO:** expose task, data, and pipeline parallelism
  - **DO:** expose parallelism in sliding windows
  - **DO:** expose startup behaviors
  - **DO NOT:** optimize for unusual case of mis-matched I/O rates
  - **DO NOT:** bother with multi-phase filters
  - **TRY:** to prevent users from introducing unnecessary state
  - **TRY:** to leverage and improve upon structured streams
  - **TRY:** to prevent induction variables from serializing filters
- **Exercise care in generalizing results beyond StreamIt**

Acknowledgments:
Authors of the StreamIt Benchmarks

- Sitij Agrawal
- Basier Aziz
- Jiawen Chen
- Matthew Drake
- Shirley Fung
- Michael Gordon
- Ola Johnsson
- Andrew Lamb
- Chris Leger
- Michal Karczmarek
- David Maze
- Ali Meli
- Mani Narayanan
- Satish Ramaswamy
- Rodric Rabbah
- Janis Sermulins
- Magnus Stenemo
- Jinwoo Suh
- Zain ul-Abdin
- Amy Williams
- Jeremy Wong