

A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs

William Thies, Vikram Chandrasekhar, Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Legacy Code

- **310 billion lines of legacy code in industry today**
 - 60-80% of typical IT budget spent re-engineering legacy code
 - (Source: Gartner Group)
- **Now code must be migrated to multicore machines**
 - Current best practice: manual translation

Parallelization: Man vs. Compiler

	Man	Compiler
		
Speed	1 op / sec	1,000,000,000 op / sec
Working Set	100 lines	1,000,000 lines
Accuracy	Makes mistakes	Fail-safe
Effectiveness	GOOD	BAD
Preserve the	Functionality	Implementation
Approach	do { attempt parallelism } until pass regtest	Be conservative!

Can we improve compilers by making them more human?

Humanizing Compilers

- First step: change our expectations of correctness

Current: An Omnipotent Being



Zeus

New: An Expert Programmer



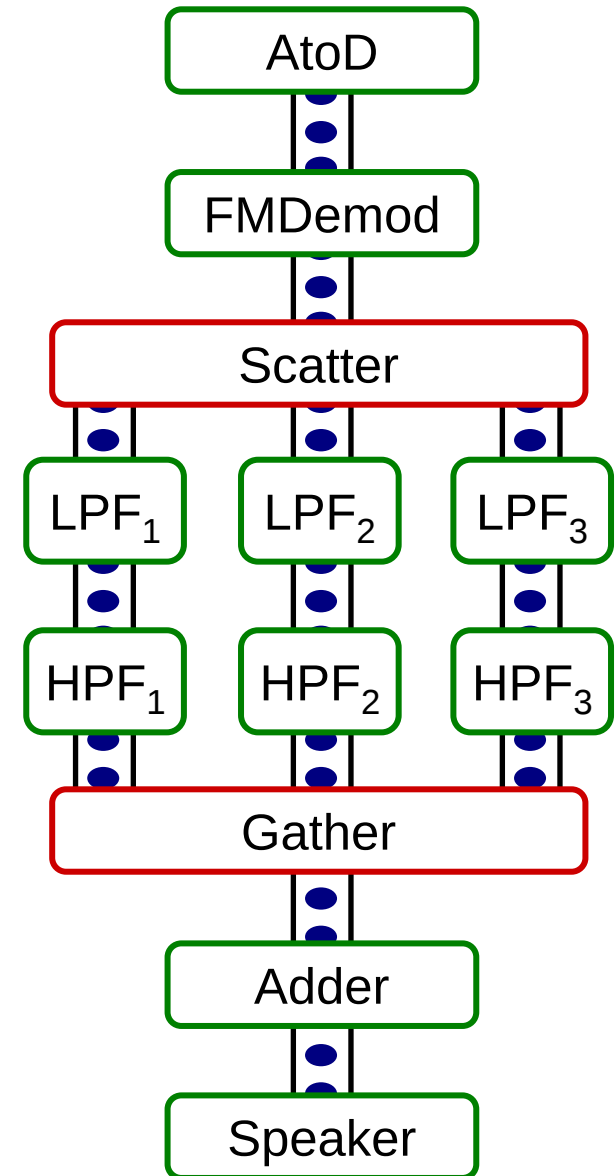
Richard Stallman

Humanizing Compilers

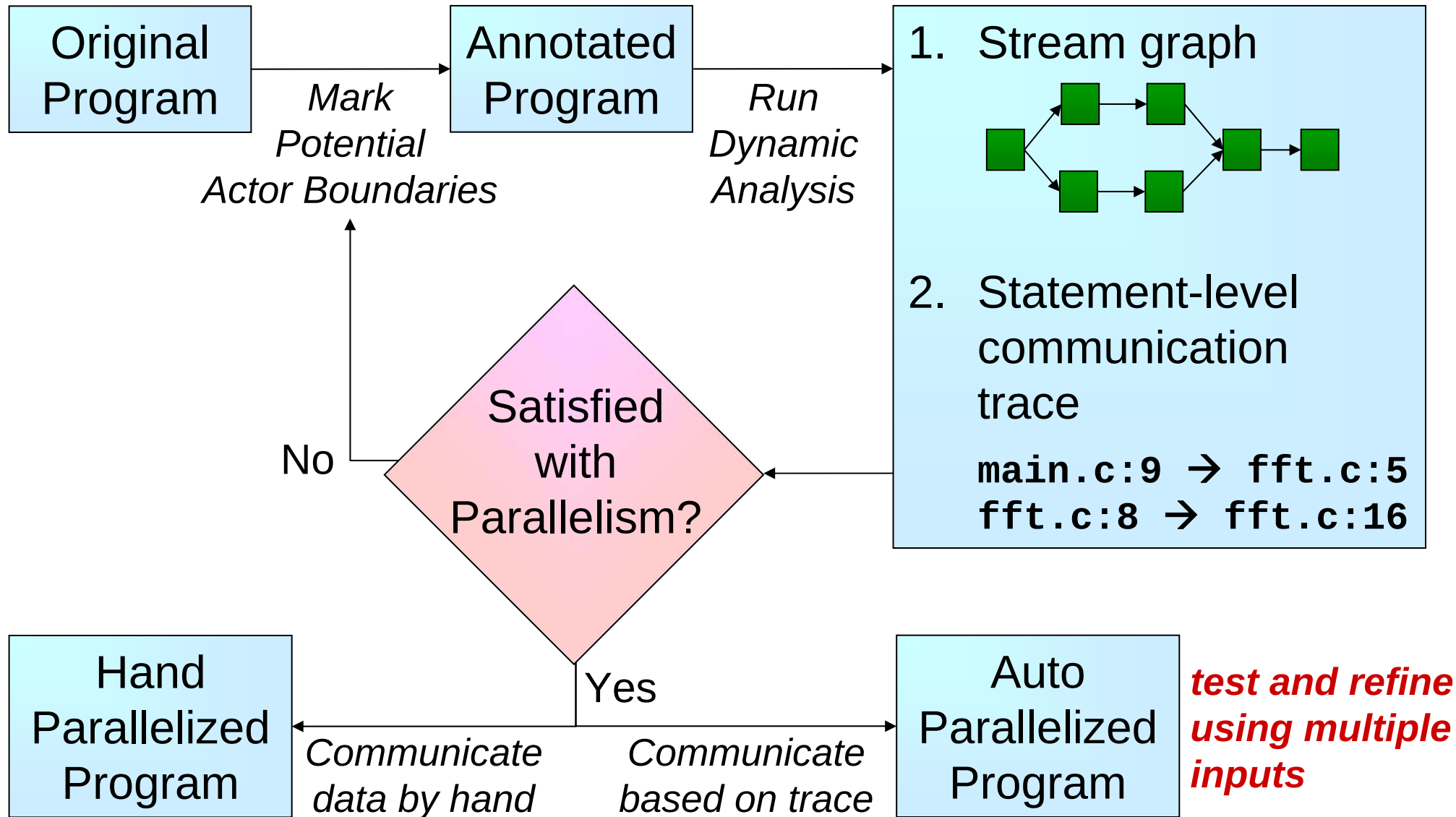
- **First step: change our expectations of correctness**
- **Second step: use compilers differently**
 - Option A: Treat them like a programmer
 - Transformations distrusted, subject to test
 - Compiler must examine failures and fix them
 - Option B: Treat them like a tool
 - Make suggestions to programmer
 - Assist programmers in understanding high-level structure
- **How does this change the problem?**
 - Can utilize unsound but useful information
 - In this talk: utilize **dynamic analysis**

Dynamic Analysis for Extracting Coarse-Grained Parallelism from C

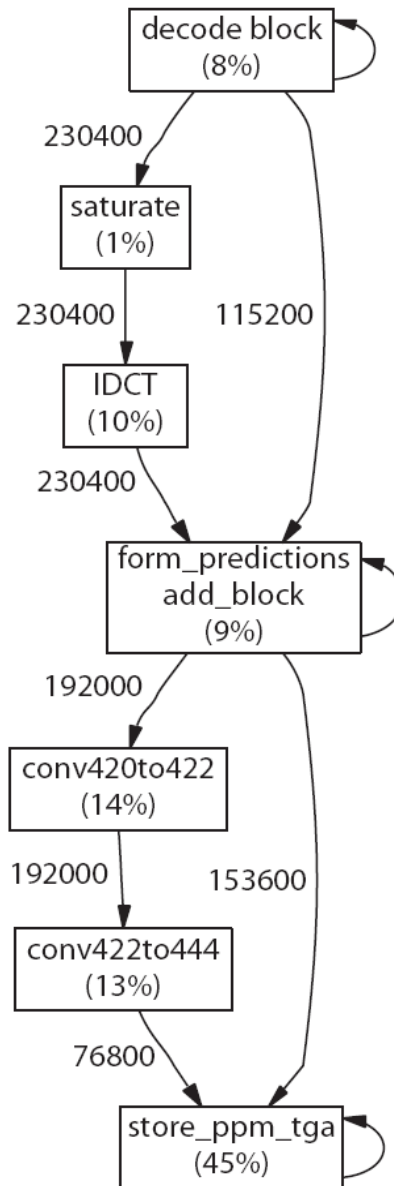
- **Focus on stream programs**
 - Audio, video, DSP, networking, and cryptographic processing kernels
 - Regular communication patterns
- **Static analysis complex or intractable**
 - Potential aliasing (pointer arithmetic, function pointers, etc.)
 - Heap manipulation (e.g., Huffman tree)
 - Circular buffers (modulo ops)
 - Correlated input parameters
- **Opportunity for dynamic analysis**
 - **If flow of data is very stable, can infer it with a small sample**



Overview of Our Approach

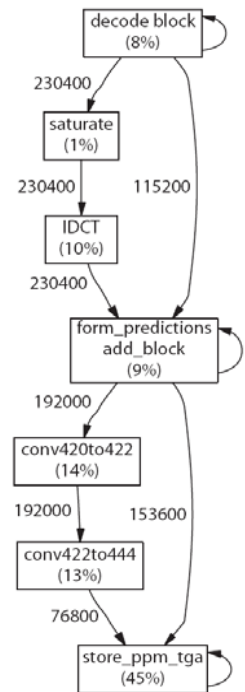
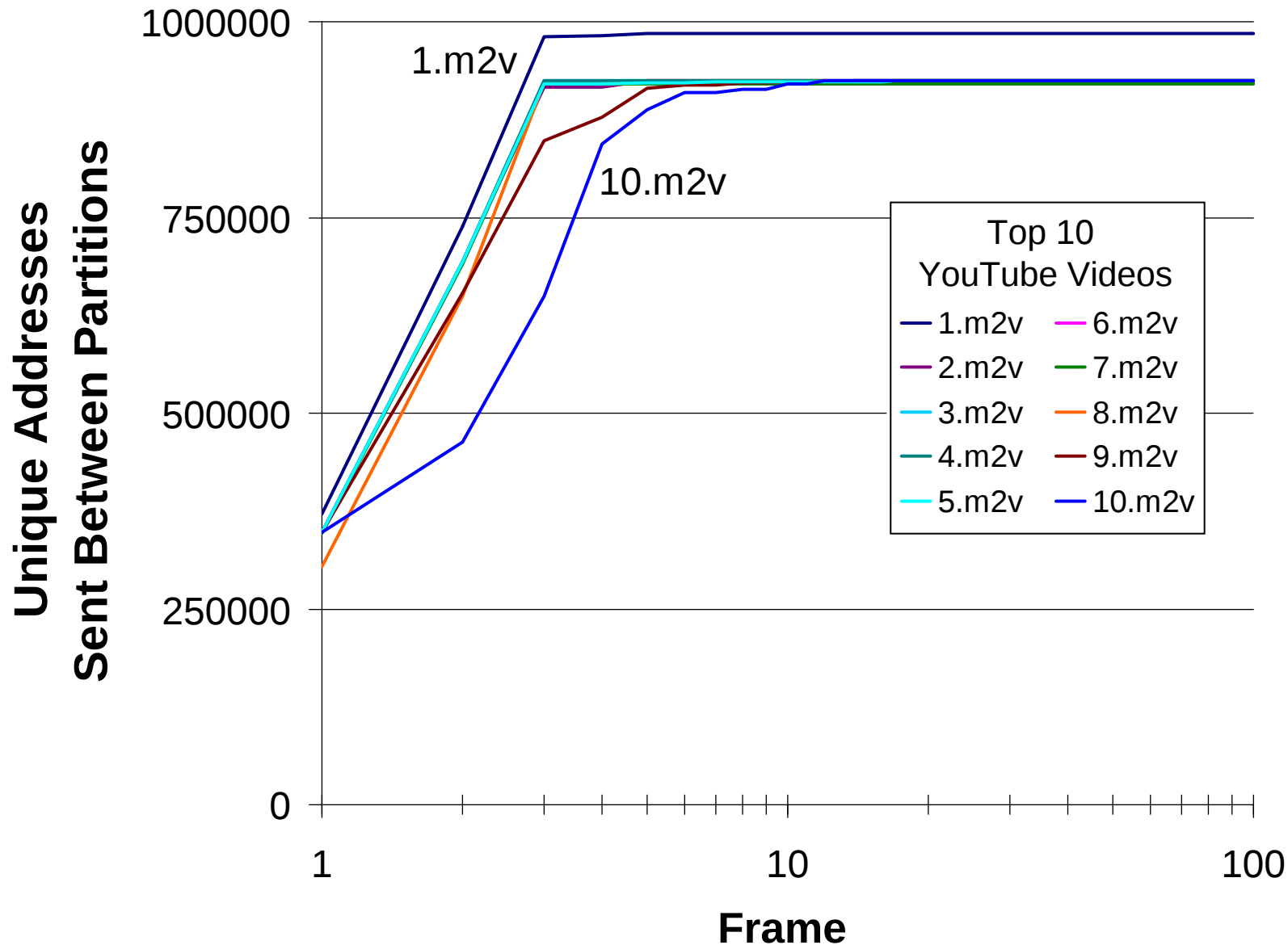


Stability of MPEG-2



MPEG-2 Decoder

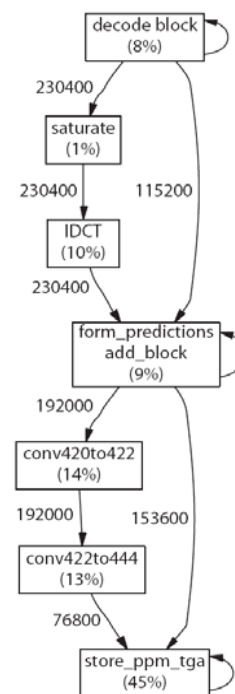
Stability of MPEG-2 (Within an Execution)



Stability of MPEG-2 (Across Executions)

MPEG-2		Testing File									
		1.m2v	2.m2v	3.m2v	4.m2v	5.m2v	6.m2v	7.m2v	8.m2v	9.m2v	10.m2v
Training File	1.m2v	3	3	3	3	3	3	3	3	3	3
	2.m2v	3	3	3	3	3	3	3	3	3	3
	3.m2v	5	5	5	5	5	5	5	5	5	5
	4.m2v	3	3	3	3	3	3	3	3	3	3
	5.m2v	3	3	3	3	3	3	3	3	3	3
	6.m2v	3	3	3	3	3	3	3	3	3	3
	7.m2v	3	3	3	3	3	3	3	3	3	3
	8.m2v	3	3	3	3	3	3	3	3	3	3
	9.m2v	3	3	3	3	3	3	3	3	3	3
	10.m2v	4	4	4	4	4	4	4	4	4	4

Minimum number of training iterations (frames) needed on each video in order to correctly decode the other videos.

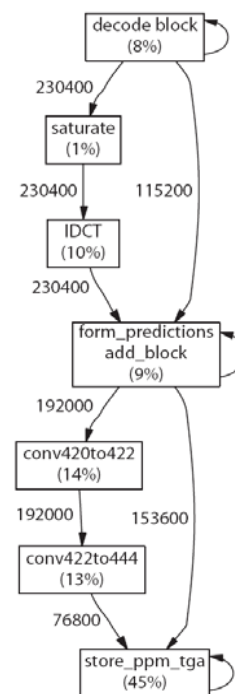


Stability of MPEG-2 (Across Executions)

MPEG-2		Testing File									
		1.m2v	2.m2v	3.m2v	4.m2v	5.m2v	6.m2v	7.m2v	8.m2v	9.m2v	10.m2v
Training File	1.m2v										3
	2.m2v										
	3.m2v										
	4.m2v										
	5.m2v										
Training File	6.m2v										3
	7.m2v										3
	8.m2v										3
	9.m2v										3
	10.m2v	4	4	4	4	4	4	4	4	4	4

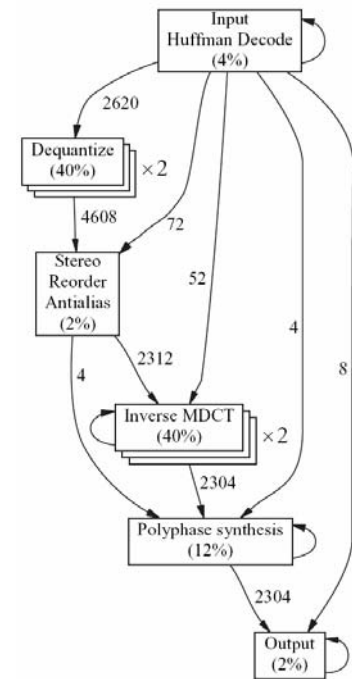
5 frames of training on one video is sufficient to correctly parallelize any other video

Minimum number of training iterations (frames) needed on each video in order to correctly decode the other videos.



Stability of MP3 (Across Executions)

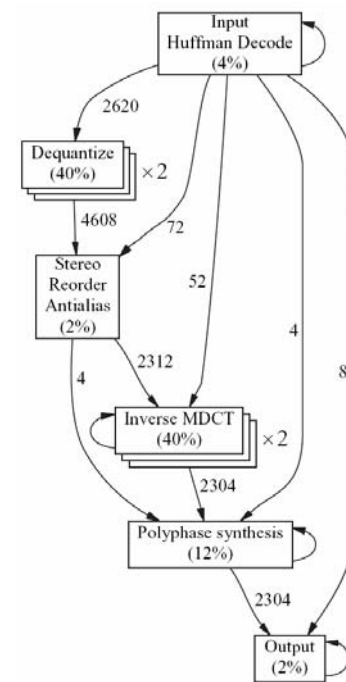
MP3		Testing File									
		1.mp3	2.mp3	3.mp3	4.mp3	5.mp3	6.mp3	7.mp3	8.mp3	9.mp3	10.mp3
Training File	1.mp3	1	1	1	1	1	1	1	1	—	—
	2.mp3	1	1	1	1	1	1	1	1	—	—
	3.mp3	1	1	1	1	1	1	1	1	—	—
	4.mp3	1	1	1	1	1	1	1	1	—	—
	5.mp3	1	1	1	1	1	1	1	1	—	—
	6.mp3	1	1	1	1	1	1	1	1	—	—
	7.mp3	1	1	1	1	1	1	1	1	—	—
	8.mp3	1	1	1	1	1	1	1	1	—	—
	9.mp3	1	1	1	1	1	1	1	1	17900	—
	10.mp3	5	5	5	5	5	5	5	5	5	5



Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.

Stability of MP3 (Across Executions)

MP3		Testing File									
		1.mp3	2.mp3	3.mp3	4.mp3	5.mp3	6.mp3	7.mp3	8.mp3	9.mp3	10.mp3
Training File	1.mp3	1	1	1	1	1	1	1	1	—	—
	2.mp3	1	1	1	1	1	1	1	1	—	—
	3.mp3	1	1	1	1	1	1	1	1	—	—
	4.mp3	1	1	1	1	1	1	1	1	—	—
	5.mp3	1	1	1	1	1	1	1	1	—	—
	6.mp3	1	1	1	1	1	1	1	1	—	—
	7.mp3	1	1	1	1	1	1	1	1	—	—
	8.mp3	1	1	1	1	1	1	1	1	—	—
	9.mp3	1	1	1	1	1	1	1	1	17900	—
	10.mp3	5	5	5	5	5	5	5	5	5	5

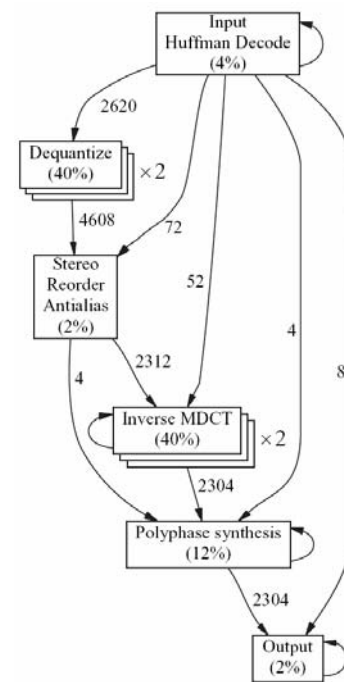


Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.

Stability of MP3 (Across Executions)

MP3		Testing File									
		1.mp3	2.mp3	3.mp3	4.mp3	5.mp3	6.mp3	7.mp3	8.mp3	9.mp3	10.mp3
Training File	1.mp3	1	1	1	1	1	1	1	1	—	—
	2.mp3	1	1	1	1	1	1	1	1	—	—
	3.mp3	1	1	1	1	1	1	1	1	—	—
	4.mp3	1	1	1	1	1	1	1	1	—	—
	5.mp3	1	1	1	1	1	1	1	1	—	—
	6.mp3	1	1	1	1	1	1	1	1	—	—
	7.mp3	1	1	1	1	1	1	1	1	—	—
	8.mp3	1	1	1	1	1	1	1	1	—	—
	9.mp3	1	1	1	1	1	1	1	1	17900	—
	10.mp3	5	5	5	5	5	5	5	5	5	5

Layer 1 frames

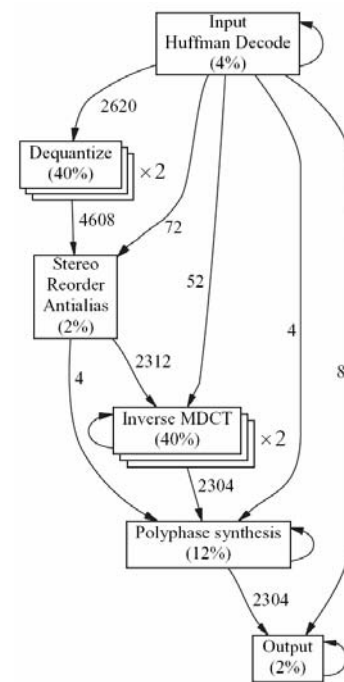


Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.

Stability of MP3 (Across Executions)

MP3		Testing File									
		1.mp3	2.mp3	3.mp3	4.mp3	5.mp3	6.mp3	7.mp3	8.mp3	9.mp3	10.mp3
Training File	1.mp3	1	1	1	1	1	1	1	1	—	—
	2.mp3	1	1	1	1	1	1	1	1	—	—
	3.mp3	1	1	1	1	1	1	1	1	—	—
	4.mp3	1	1	1	1	1	1	1	1	—	—
	5.mp3	1	1	1	1	1	1	1	1	—	—
	6.mp3	1	1	1	1	1	1	1	1	—	—
	7.mp3	1	1	1	1	1	1	1	1	—	—
	8.mp3	1	1	1	1	1	1	1	1	—	—
	9.mp3	1	1	1	1	1	1	1	1	17900	—
	10.mp3	5	5	5	5	5	5	5	5	5	5

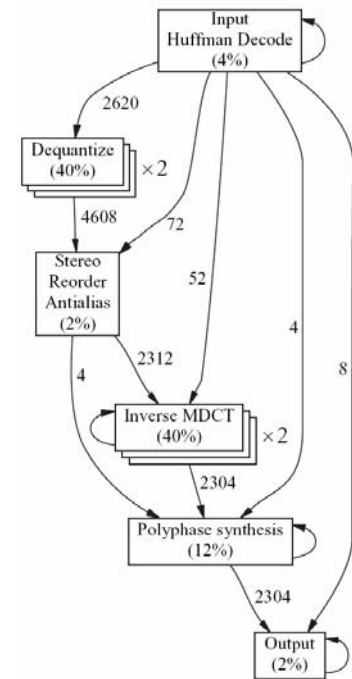
CRC Error



Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.

Stability of MP3 (Across Executions)

MP3		Testing File									
		1.mp3	2.mp3	3.mp3	4.mp3	5.mp3	6.mp3	7.mp3	8.mp3	9.mp3	10.mp3
Training File	1.mp3	1	1	1	1	1	1	1	1	—	—
	2.mp3	1	1	1	1	1	1	1	1	—	—
	3.mp3	1	1	1	1	1	1	1	1	—	—
	4.mp3	1	1	1	1	1	1	1	1	—	—
	5.mp3	1	1	1	1	1	1	1	1	—	—
	6.mp3	1	1	1	1	1	1	1	1	—	—
	7.mp3	1	1	1	1	1	1	1	1	—	—
	8.mp3	1	1	1	1	1	1	1	1	—	—
	9.mp3	1	1	1	1	1	1	1	1	17900	—
	10.mp3	5	5	5	5	5	5	5	5	5	5



Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.

Outline

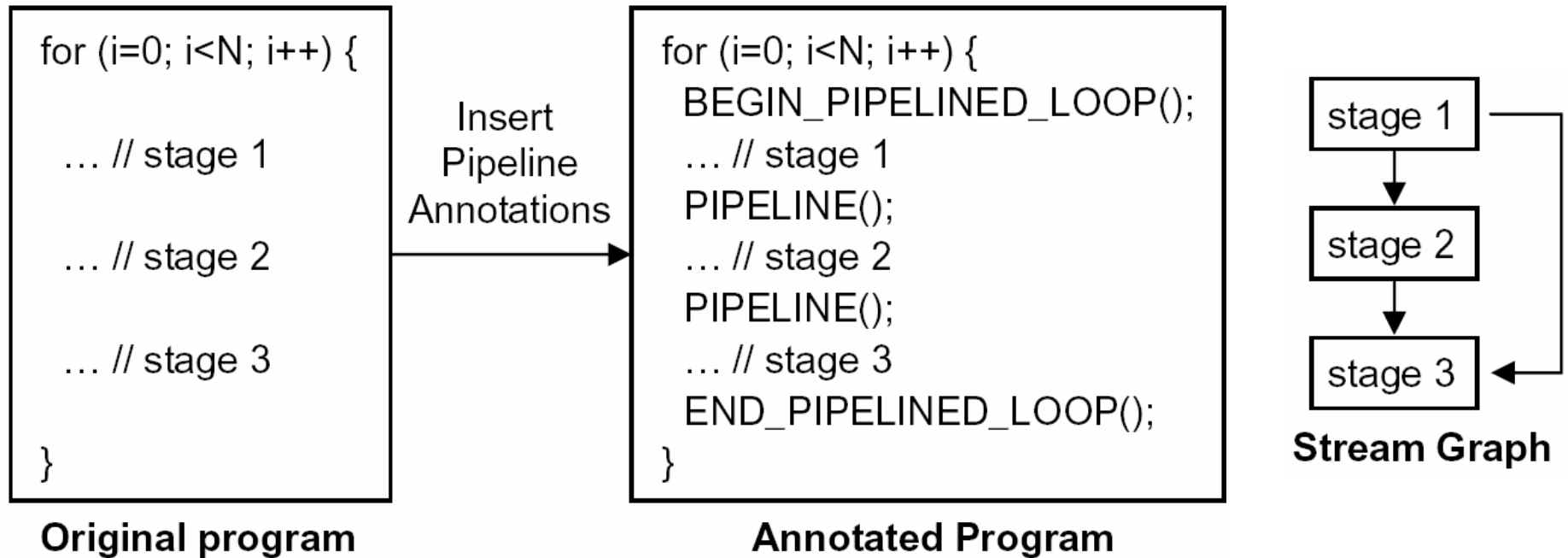
- **Analysis Tool**
- **Case Studies**

Outline

- **Analysis Tool**
- **Case Studies**

Annotating Pipeline Parallelism

- **Programmer indicates potential actor boundaries in a long-running loop**



- **Serves as a fundamental API for pipeline parallelism**
 - Comparable to OpenMP for data parallelism
 - Comparable to Threads for task parallelism

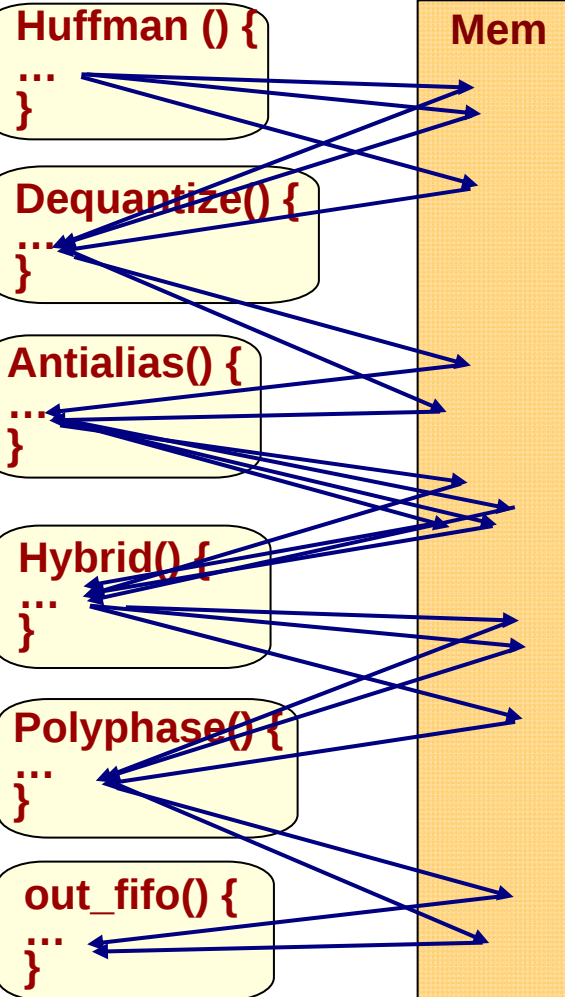
Dynamic Analysis

Legacy C Code

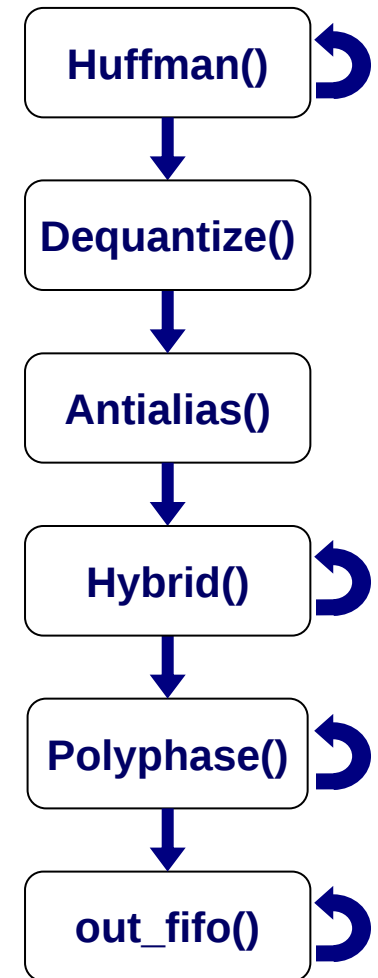
```
while (!end_bs(&bs)) {  
    BEGIN_PIPELINED_LOOP();  
    for (ch=0; ch<stereo; ch++) {  
        III_huffman_decode(is[ch], &III_side_info, ch, gr,  
                           part2_start, &fr_ps);  
        PIPELINE();  
        III_dequantize_sample(is[ch], ro[ch], III_scalefac,  
                             &(III_side_info.ch[ch].gr[gr]), ch, &fr_ps);  
    }  
    ...  
    PIPELINE();  
    for (ch=0; ch<stereo; ch++) {  
        ...  
        III_antialias(re, hybridIn, /* Antialias butterflies */  
                     &(III_side_info.ch[ch].gr[gr]), &fr_ps);  
        for (sb=0; sb<SBLIMIT; sb++) { /* Hybrid synthesis */  
            PIPELINE();  
            III_hybrid(hybridIn[sb], hybridOut[sb], sb, ch,  
                      &(III_side_info.ch[ch].gr[gr]), &fr_ps);  
            PIPELINE();  
        }  
        /* Frequency inversion for polyphase */  
        for (ss=0; ss<18; ss++)  
            for (sb=0; sb<SBLIMIT; sb++)  
                if ((ss%2) && (sb%2))  
                    hybridOut[sb][ss] = -hybridOut[sb][ss];  
        for (ss=0; ss<18; ss++) { /* Polyphase synthesis */  
            for (sb=0; sb<SBLIMIT; sb++)  
                polyPhaseIn[sb] = hybridOut[sb][ss];  
            clip += SubBandSynthesis (polyPhaseIn, ch,  
                                     &((*pcm_sample)[ch][ss][0]));  
        }  
        PIPELINE();  
        /* Output PCM sample points for one granule */  
        out_fifo(*pcm_sample, 18, &fr_ps, done, musicout,  
                &sample_frames);  
        END_PIPELINED_LOOP();  
    }  
    ...  
}
```

MP3 Decoding

Record Who Produces / Consumes each Location

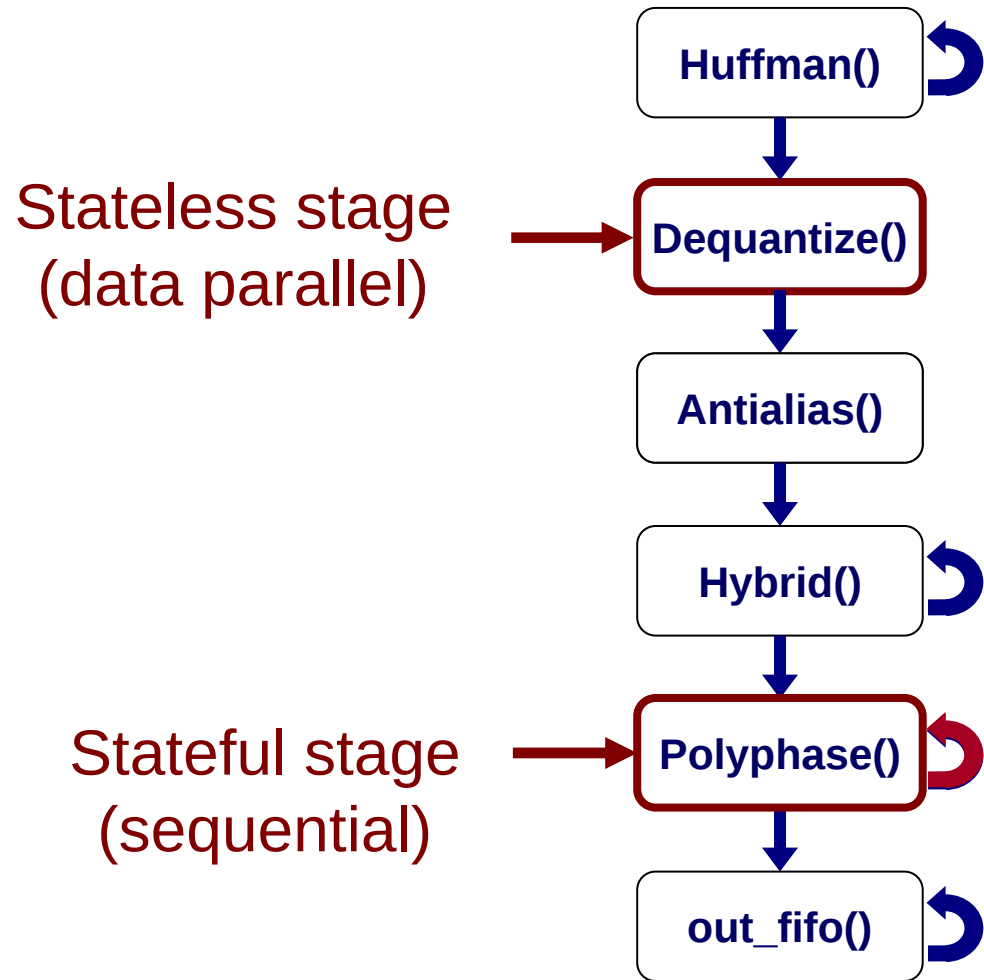


Build Block Diagram



Implemented Using Valgrind

Exploiting the Parallelism

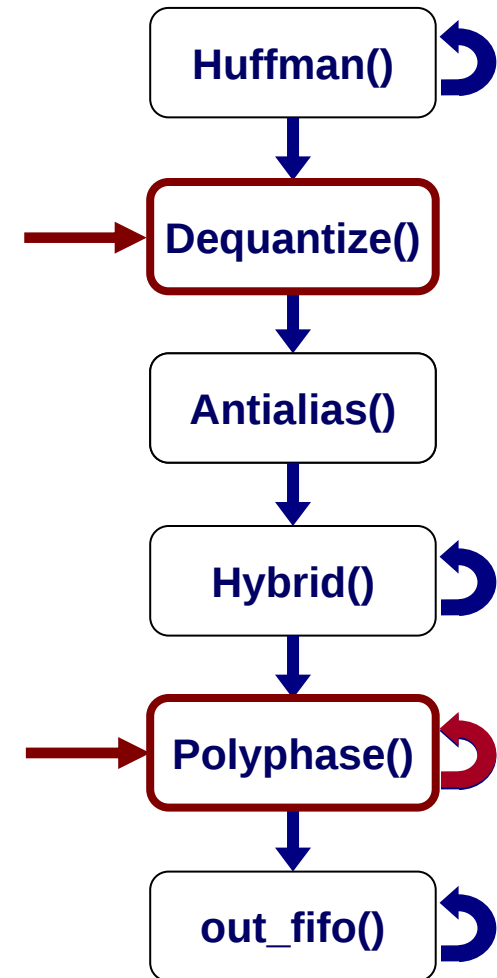


Exploiting the Parallelism

```
for (i=0; i<N; i++) {  
    ...  
    PIPELINE();  
    Dequantize();  
    PIPELINE();  
    ....  
}
```

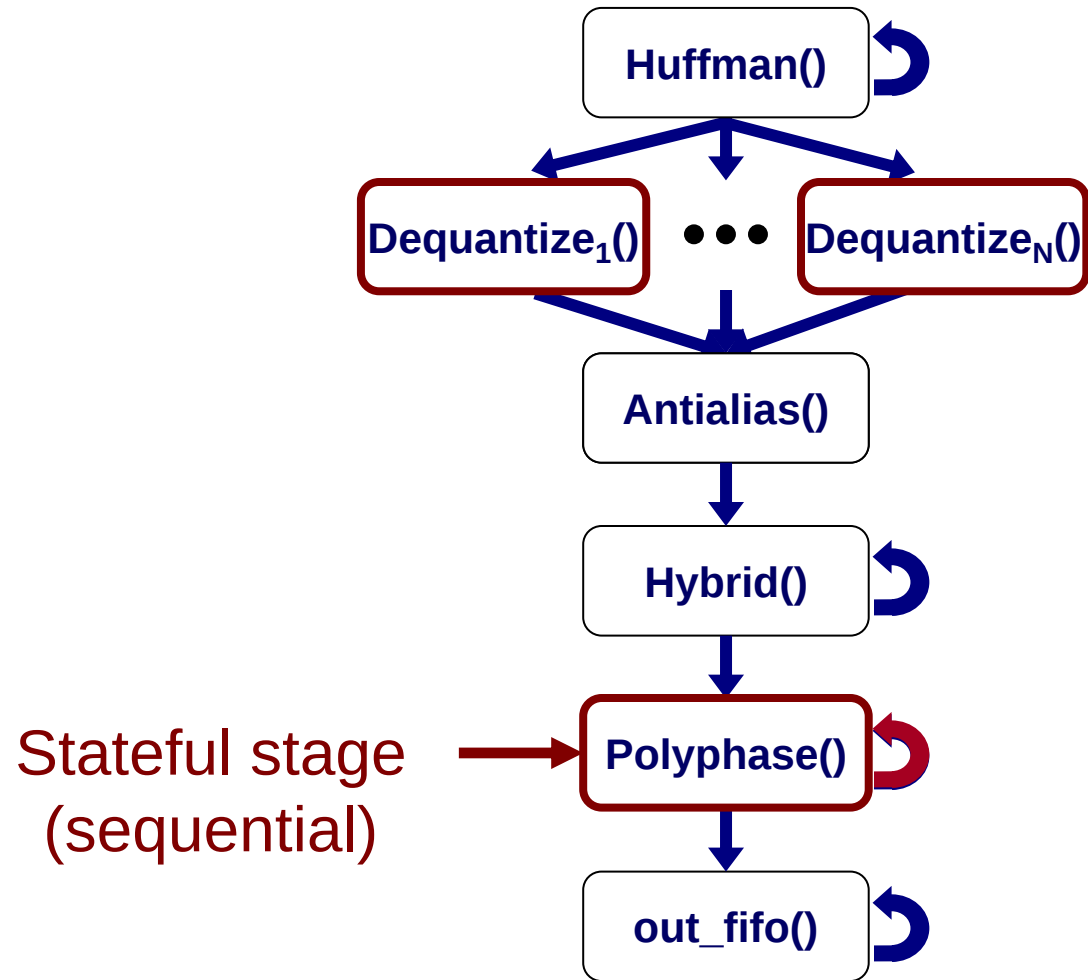
Stateless stage
(data parallel)

Stateful stage
(sequential)



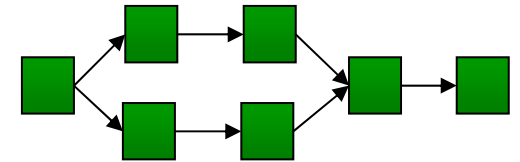
Exploiting the Parallelism

```
for (i=0; i<N; i++) {  
    ...  
    PIPELINE(N);  
    Dequantize();  
    PIPELINE();  
    ....  
}
```



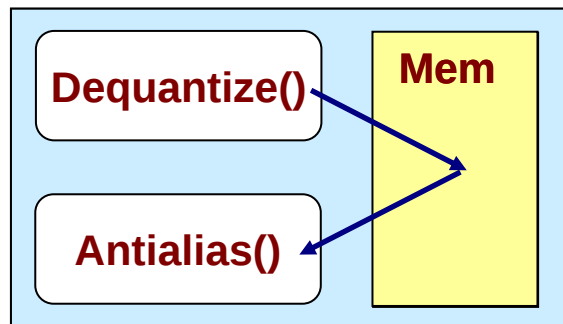
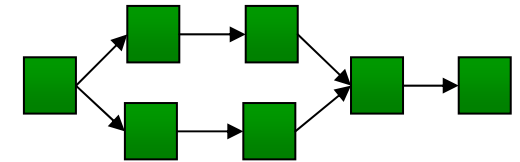
Parallel Runtime Environment

- Pipeline parallelism requires buffering between stages
- Two ways to implement buffering:
 1. Modify original program to add buffers
 - ✓ 2. Wrap original code in virtual execution environment
- We fork each actor into an independent process, and communicate the recorded variables via pipes



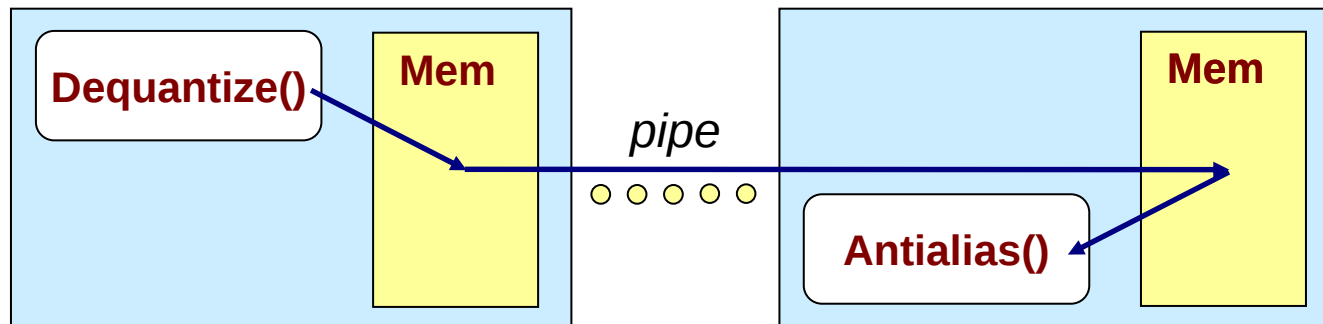
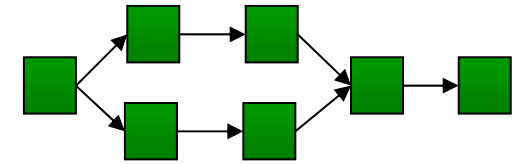
Parallel Runtime Environment

- Pipeline parallelism requires buffering between stages
- Two ways to implement buffering:
 1. Modify original program to add buffers
 - ✓ 2. Wrap original code in virtual execution environment
- We fork each actor into an independent process, and communicate the recorded variables via pipes



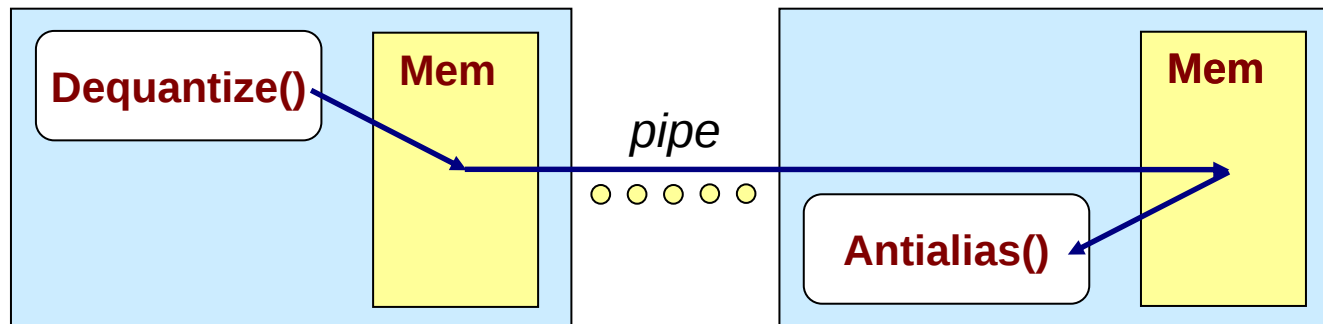
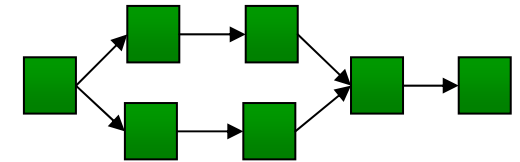
Parallel Runtime Environment

- Pipeline parallelism requires buffering between stages
- Two ways to implement buffering:
 1. Modify original program to add buffers
 - ✓ 2. Wrap original code in virtual execution environment
- We fork each actor into an independent process, and communicate the recorded variables via pipes



Parallel Runtime Environment

- Pipeline parallelism requires buffering between stages
- Two ways to implement buffering:
 1. Modify original program to add buffers
 - ✓ 2. Wrap original code in virtual execution environment
- We fork each actor into an independent process, and communicate the recorded variables via pipes



Programmer assistance needed for:

- malloc'd data
- nested loops
- reduction vars

- Robust in the presence of aliasing
- Suitable to shared or distributed memory
- Efficient (7% communication overhead on MP3)

Outline

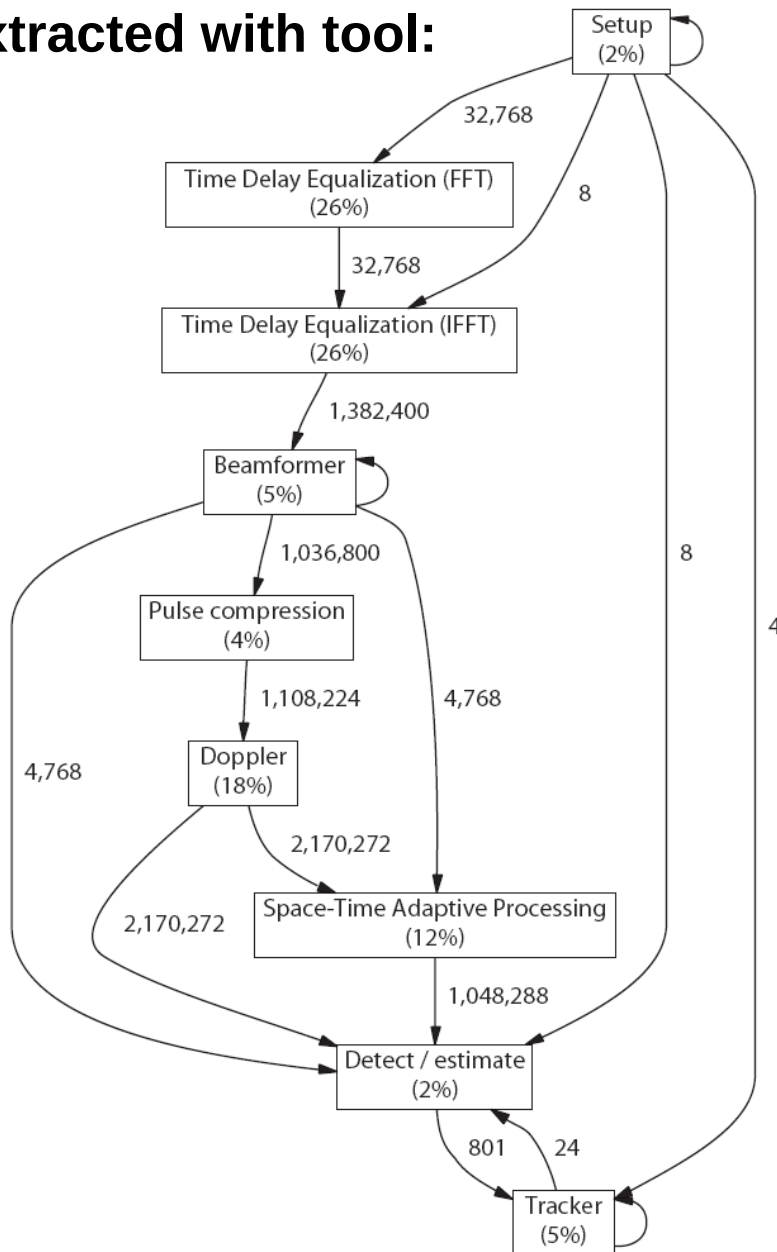
- **Analysis Tool**
- **Case Studies**

Extracted Stream Graphs

Benchmark	Description	Source	Lines of Code
GMTI	Ground Moving Target Indicator	MIT Lincoln Laboratory	37,000
MP3	MP3 audio decoder	Fraunhofer IIS	5,000
MPEG-2	MPEG-2 video decoder	MediaBench	10,000
197.parser	Grammatical parser of English language	SPECINT 2000	11,000
256.bzip2	bzip2 compression and decompression	SPECINT 2000	5,000
456.hmmer	Calibrating HMMs for biosequence analysis	SPECCPU 2006	36,000

Ground Moving Target Indicator (GMTI)

Extracted with tool:



From GMTI specification:

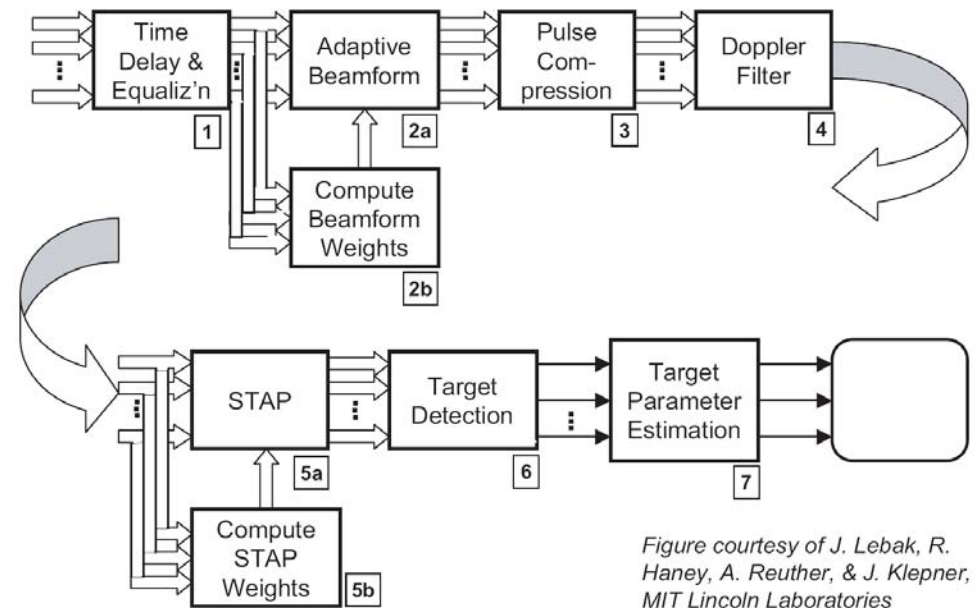
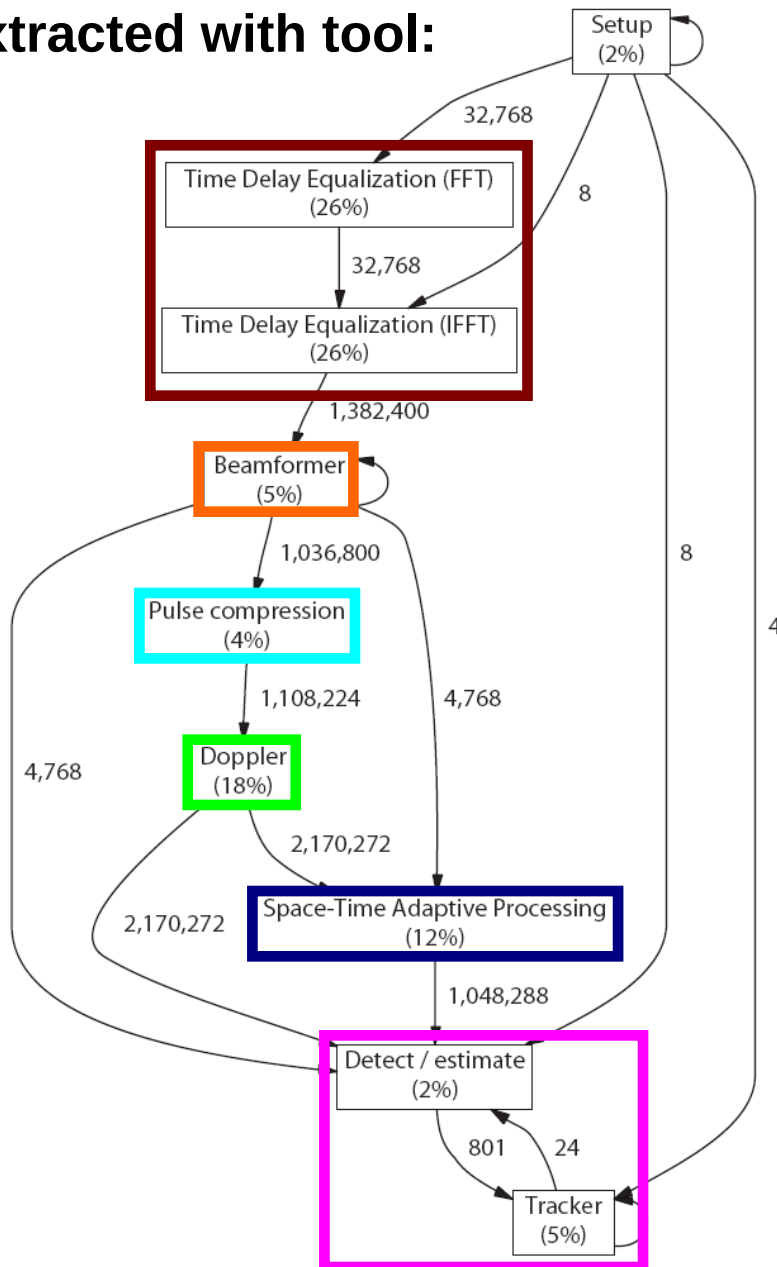


Figure courtesy of J. Lebak, R. Haney, A. Reuther, & J. Klepner, MIT Lincoln Laboratories

Ground Moving Target Indicator (GMTI)

Extracted with tool:



From GMTI specification:

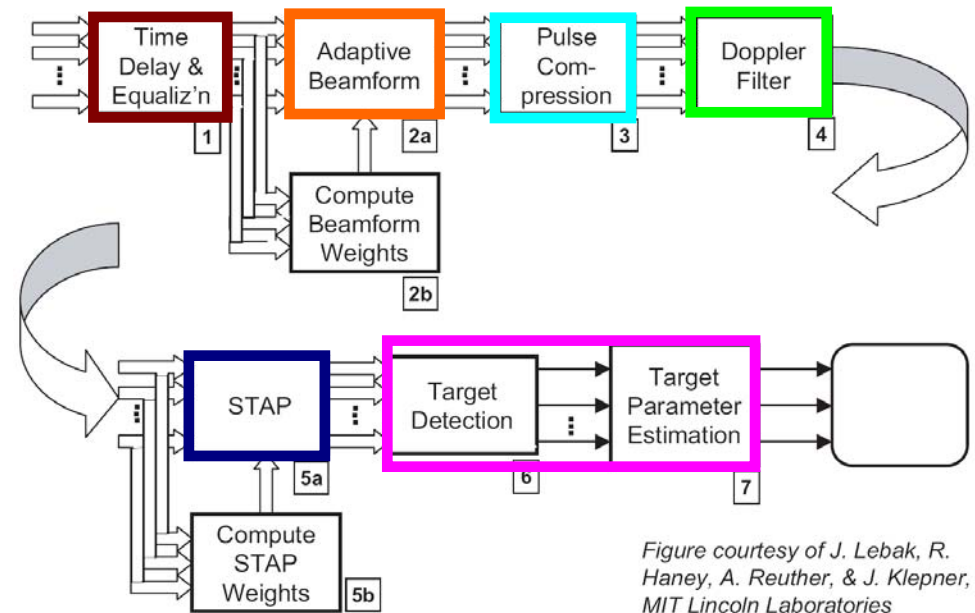
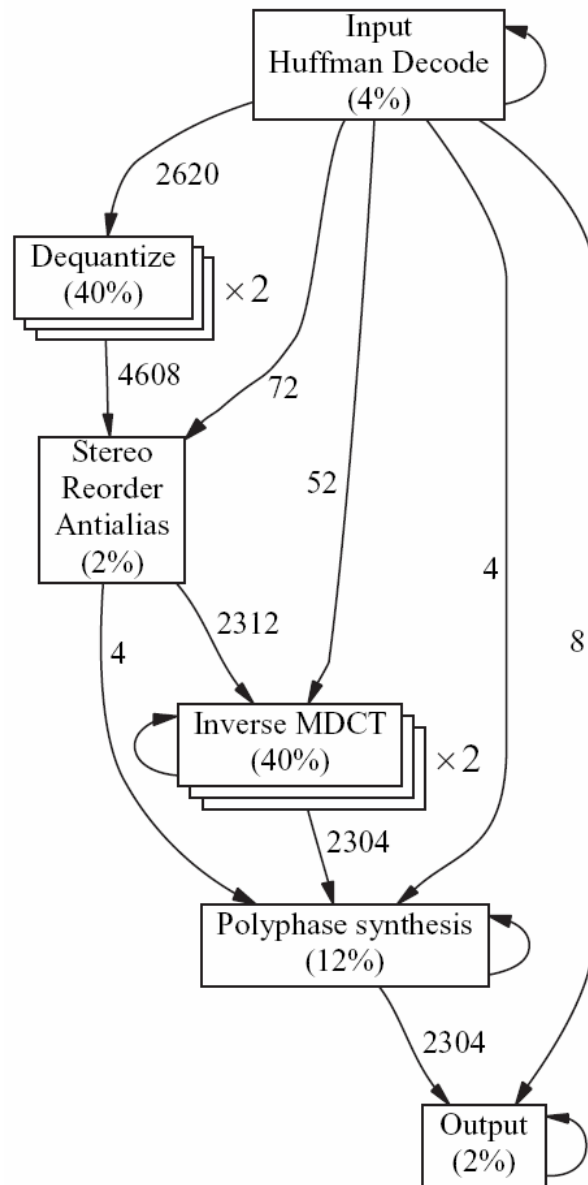
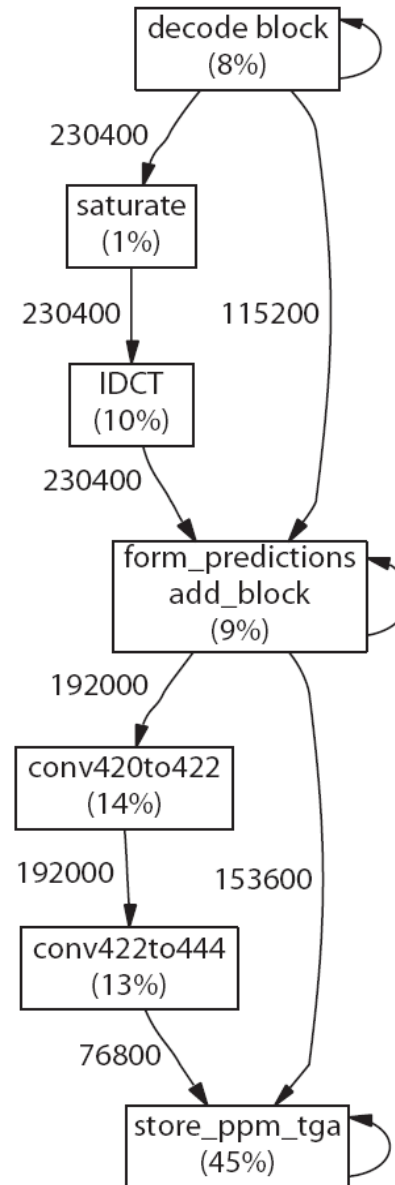


Figure courtesy of J. Lebak, R. Haney, A. Reuther, & J. Klepner, MIT Lincoln Laboratories

Audio and Video Codecs

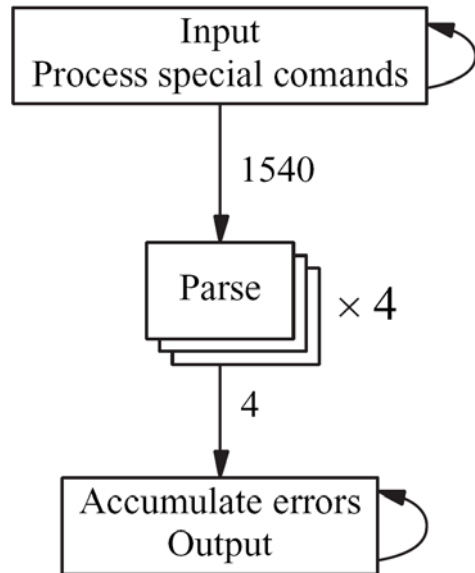


MP3 Decoder

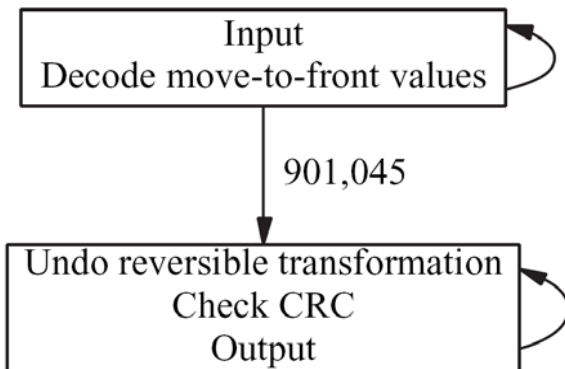


MPEG-2 Decoder

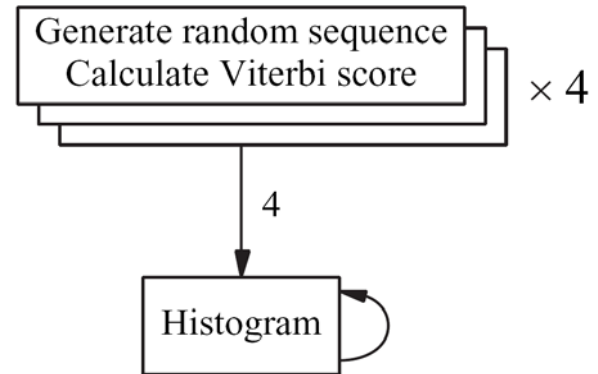
SPEC Benchmarks



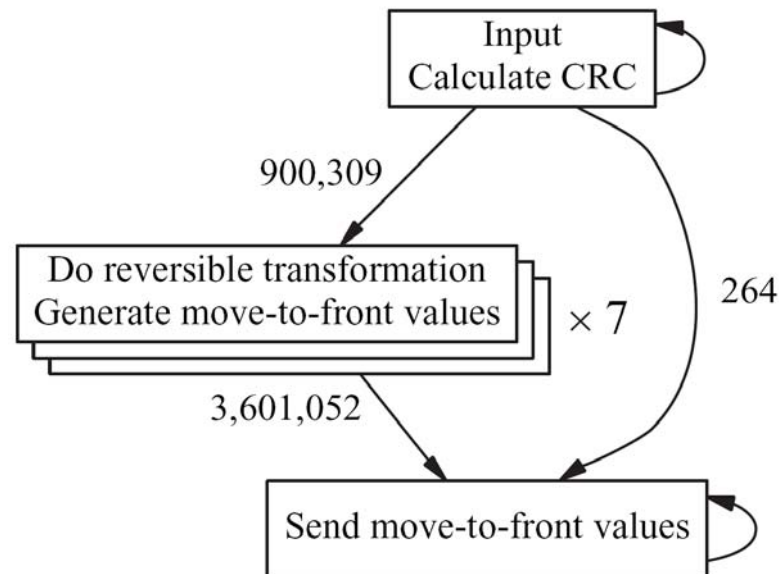
197.parser



256.bzip2 (decompression)



456.hmmmer



256.bzip2 (compression)

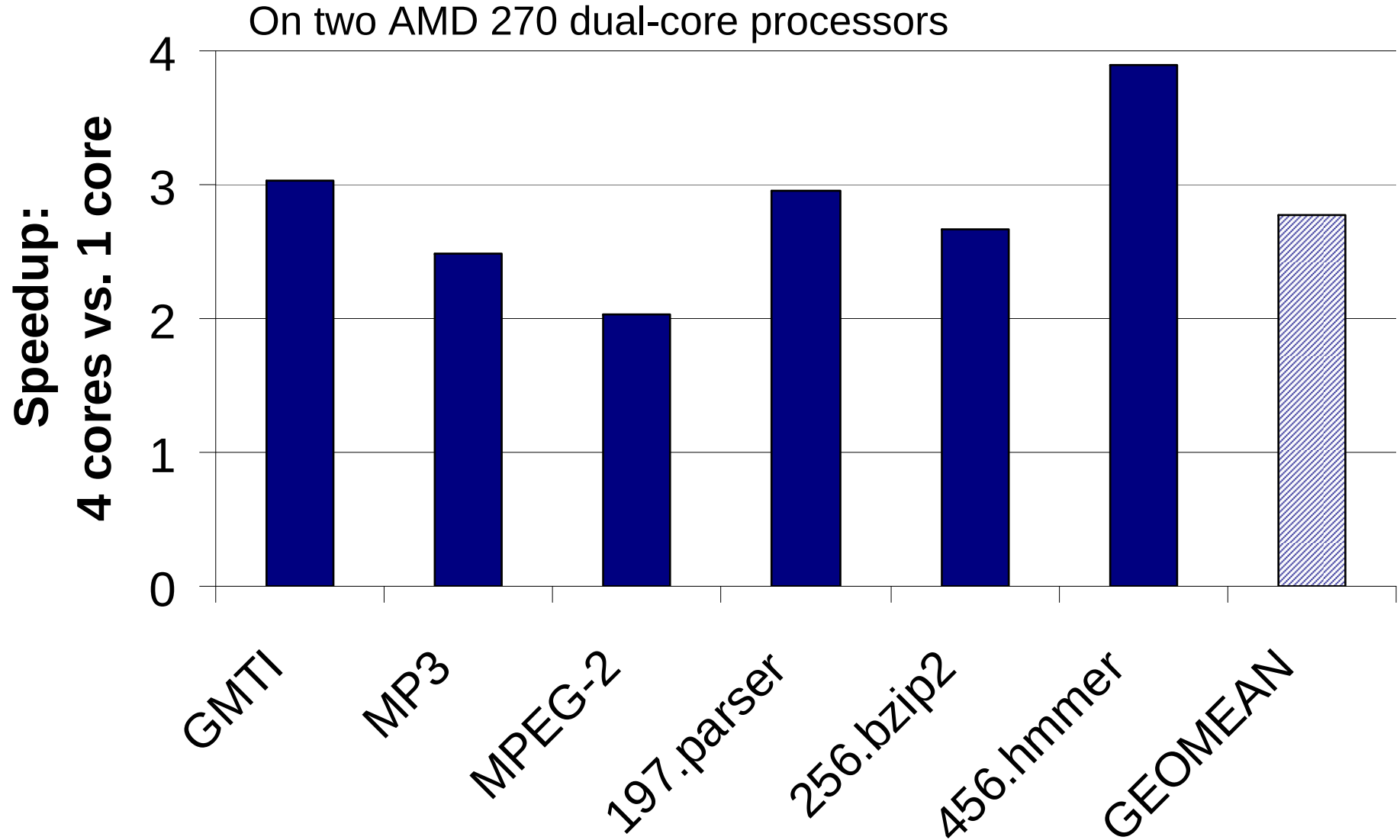
Interactive Parallelization Process

- **Analysis tool exposed serializing dependences**
 - As annotated back-edges in stream graph (main.c:9 → fft.c:5)
- **How to deal with serializing dependences?**
 1. Rewrite code to eliminate dependence, or
 2. Instruct the tool to ignore the dependence
- **Lesson learned:**

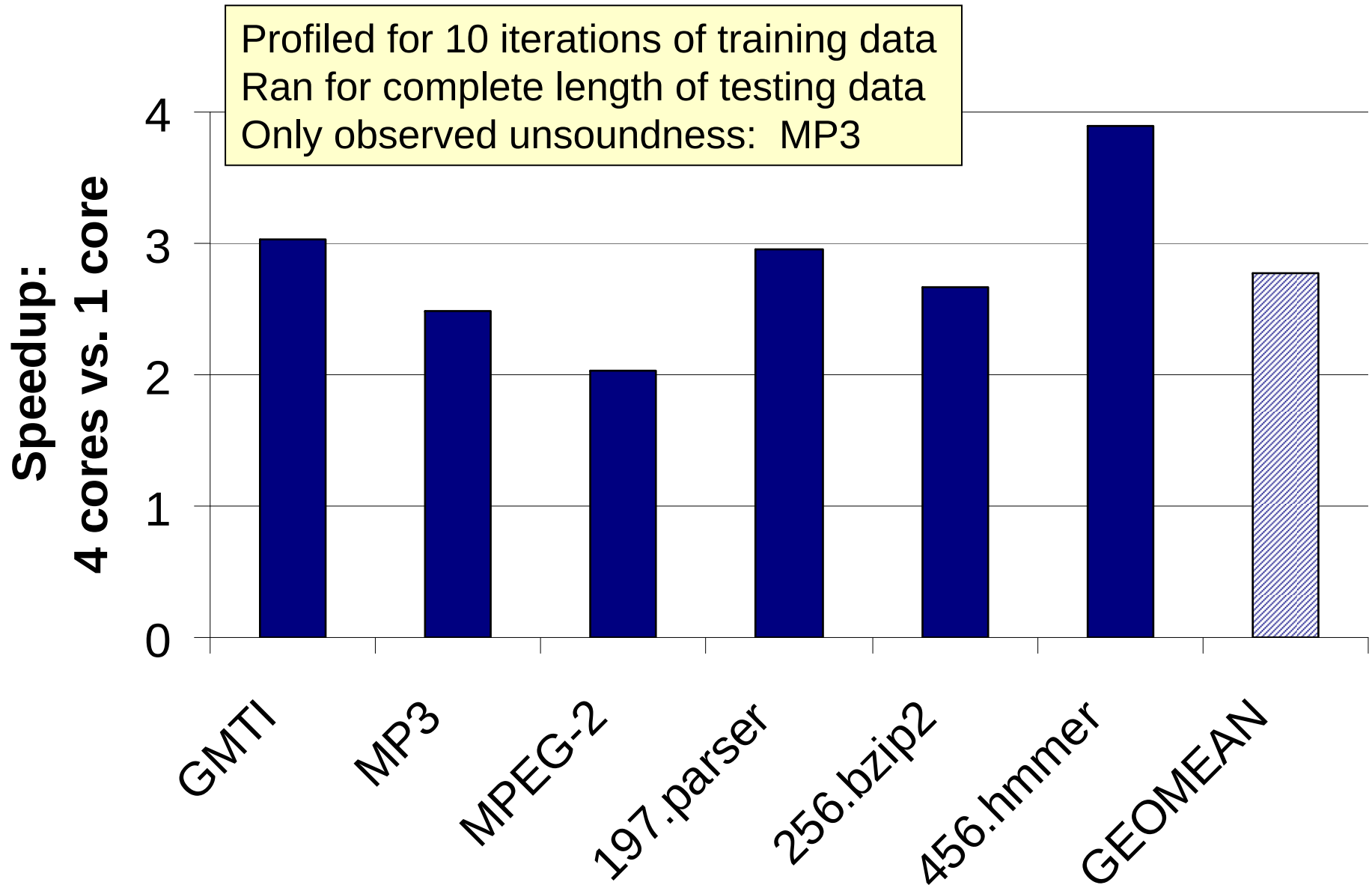
Many memory dependences can be safely ignored!

 - Allow malloc (or free) to be called in any order (GMTI, hmmer)
 - Allow rand() to be called in any order (hmmer)
 - Ignore dependences on uninitialized memory (parser)
 - Ignore ordering of demand-driven buffer expansion (hmmer)

Results



Results



How to Improve Soundness?

- **Revert to sequential version upon seeing new code (fixes MP3)**
- **Hardware support**
 - Mondriaan memory protection (Witchel et. al)
 - Versioned memory (used by Bridges et al.)
 - Would provide safe communication, but unsafe parallelism
- **Rigorous testing with maximal code coverage**
- **Programmer review**

Related Work

- **Revisiting the Sequential Programming Model for Multi-Core** (Bridges et al., yesterday)
 - Same pipeline-parallel decompositions of parser, bzip2
 - Like *commutative* annotation, we tell tool to ignore dependences
 - But since we target distributed memory, annotation represents privatization rather than reordering
- **Dynamic analysis for understanding, parallelization**
 - Rul et. al (2006) – programmer manages communication
 - Redux (2003) – fine-grained dependence visualization
 - Karkowski and Corporaal (1997) – focus on data parallelism
- **Inspector/executor for DOACROSS parallelism**
 - Rauchwerger (1998) – survey

Conclusions

- **Dynamic analysis can be useful for parallelization**
 - Our tool is simple, transparent, and one of the first to extract coarse-grained pipeline parallelism from C programs
 - Primary application: program understanding
 - Secondary application: automatic parallelization
- **Future work in improving soundness, automation**