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

MICRO 40 – December 4, 2007
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

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

**Can we improve compilers by making them more human?**
Humanizing Compilers

• First step: change our expectations of correctness

Current: An Omnipotent Being

New: An Expert Programmer

Zeus

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

1. Stream graph
2. Statement-level communication trace

Original Program → Annotated Program

Mark Potential Actor Boundaries
Run Dynamic Analysis

Satisfied with Parallelism?

Hand Parallelized Program
Communicate data by hand

Communicate based on trace
Yes → Auto Parallelized Program

No → Yes

Yes

Communicate based on trace

Yes

test and refine using multiple inputs
Stability of MPEG-2

MPEG-2 Decoder
Stability of MPEG-2 (Within an Execution)

Top 10 YouTube Videos
- 1.m2v
- 6.m2v
- 2.m2v
- 7.m2v
- 3.m2v
- 8.m2v
- 4.m2v
- 9.m2v
- 5.m2v
- 10.m2v

Unique Addresses Sent Between Partitions

Frame
Stability of MPEG-2 (Across Executions)

<table>
<thead>
<tr>
<th>Training File</th>
<th>Testing File 1.m2v</th>
<th>2.m2v</th>
<th>3.m2v</th>
<th>4.m2v</th>
<th>5.m2v</th>
<th>6.m2v</th>
<th>7.m2v</th>
<th>8.m2v</th>
<th>9.m2v</th>
<th>10.m2v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.m2v</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>9.m2v</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10.m2v</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Minimum number of training iterations (frames) needed on each video in order to correctly decode the other videos.
Stability of MPEG-2 (Across Executions)

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

5 frames of training on one video is sufficient to correctly parallelize any other video.
### Stability of MP3 (Across Executions)

<table>
<thead>
<tr>
<th>Training File</th>
<th>Testing File</th>
<th>1.mp3</th>
<th>2.mp3</th>
<th>3.mp3</th>
<th>4.mp3</th>
<th>5.mp3</th>
<th>6.mp3</th>
<th>7.mp3</th>
<th>8.mp3</th>
<th>9.mp3</th>
<th>10.mp3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>17900</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>2.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>3.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>4.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>5.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>6.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>7.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>8.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>9.mp3</td>
<td>1 1 1 1 1 1 1 1 1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5 5</td>
</tr>
<tr>
<td>10.mp3</td>
<td>5 5 5 5 5 5 5 5 5</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
</tr>
</tbody>
</table>

Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.
### Stability of MP3 (Across Executions)

<table>
<thead>
<tr>
<th>Training File</th>
<th>Testing File</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP3</strong></td>
<td>1.mp3 2.mp3 3.mp3 4.mp3 5.mp3 6.mp3 7.mp3 8.mp3 9.mp3 10.mp3</td>
</tr>
<tr>
<td>1.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>2.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>3.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>4.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>5.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>6.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>7.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>8.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>9.mp3</td>
<td>1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>10.mp3</td>
<td>5 5 5 5 5 5 5 5 5 5</td>
</tr>
</tbody>
</table>

Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.
**Stability of MP3 (Across Executions)**

<table>
<thead>
<tr>
<th>MP3</th>
<th>1.mp3</th>
<th>2.mp3</th>
<th>3.mp3</th>
<th>4.mp3</th>
<th>5.mp3</th>
<th>6.mp3</th>
<th>7.mp3</th>
<th>8.mp3</th>
<th>9.mp3</th>
<th>10.mp3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>6.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>7.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17900</td>
<td>—</td>
</tr>
<tr>
<td>10.mp3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Training File (MP3)</th>
<th>1.mp3</th>
<th>2.mp3</th>
<th>3.mp3</th>
<th>4.mp3</th>
<th>5.mp3</th>
<th>6.mp3</th>
<th>7.mp3</th>
<th>8.mp3</th>
<th>9.mp3</th>
<th>10.mp3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>2.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>4.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>6.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>7.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9.mp3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17900</td>
<td>—</td>
</tr>
<tr>
<td>10.mp3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.
## Stability of MP3 (Across Executions)

<table>
<thead>
<tr>
<th>Training File</th>
<th>Testing File</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MP3</strong></td>
<td><strong>1.mp3</strong></td>
</tr>
<tr>
<td><strong>1.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>3.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>4.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>5.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>6.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>7.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>8.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>9.mp3</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>10.mp3</strong></td>
<td>5</td>
</tr>
</tbody>
</table>

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

```
for (i=0; i<N; i++) {
    ...
    // stage 1
    ...
    // stage 2
    ...
    // stage 3
}
```

Original program

-- Insert Pipeline Annotations --

```
for (i=0; i<N; i++) {
    BEGIN_PIPELINED_LOOP();
    ...
    // stage 1
    PIPELINE();
    ...
    // stage 2
    PIPELINE();
    ...
    // stage 3
    END_PIPELINED_LOOP();
}
```

Annotated Program

• Serves as a fundamental API for pipeline parallelism
  – Comparable to OpenMP for data parallelism
  – Comparable to Threads for task parallelism
Dynamic Analysis

Record Who Produces / Consumes each Location

Legacy C Code

MP3 Decoding

while (!end_bs(&bs)) {
    BEGIN_PIPELINED_LOOP();
    for (ch=0; ch<stereo; ch++) {
        III_hufman_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++) { /* Polyphase synthesis */
            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]));
            }
        }
        out_fifo();
        /* Output PCM sample points for one granule */
        out_fifo("pcm_sample", 18, &fr_ps, done, musicout,
        &sample_frames);
    }
    END_PIPELINED_LOOP();
}

Implemented Using Valgrind

Build Block Diagram

Huffman() —> Dequantize() —> Antialias() —> Hybrid() —> Polyphase() —> out_fifo()
Exploiting the Parallelism

Stateless stage (data parallel)
- Dequantize()
- Antialias()
- Hybrid()
- Polyphase()
- out_fifo()

Stateful stage (sequential)
- Huffman()
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();
    ....
}

Stateful stage (sequential)
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

![Diagram showing pipeline parallelism with Dequantize, Mem, pipe, Antialias, and Mem]
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

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

Programmer assistance needed for:
- malloc’d data
- nested loops
- reduction vars
Outline

• Analysis Tool
• Case Studies
# Extracted Stream Graphs

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

Extracted with tool:

From GMTI specification:

Figure courtesy of J. Lebak, R. Haney, A. Reuther, & J. Klapner, MIT Lincoln Laboratories
Ground Moving Target Indicator (GMTI)

Extracted with tool:

From GMTI specification:
Audio and Video Codecs

MP3 Decoder

MPEG-2 Decoder
SPEC Benchmarks

197.parser

256.bzip2 (decompression)

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

On two AMD 270 dual-core processors

Speedup:
4 cores vs. 1 core

GMTI, MP3, MPEG-2, 197.parser, 256.bzip2, 456.hmmr, GEOMEAN
Results

Profiled for 10 iterations of training data
Ran for complete length of testing data
Only observed unsoundness: MP3
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
  – Karkowski and Corporaal (1997) – focus on data parallelism

• Inspector/executor for DOACROSS parallelism
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**