Single and Multi-CPU Performance Modeling for Embedded Systems (Dissertation Talk)

Trevor Meyerowitz, Phd Candidate, UC Berkeley
tcm@eecs.berkeley.edu
February 26, 2008

Advisor: Alberto Sangiovanni-Vincentelli
Dissertation Committee: Prof. Alper Atamturk, Prof. Rastislav Bodik, Prof. Alberto Sangiovanni-Vincentelli (chair)
Outline

- **Introduction**
  - Motivation
  - The Performance Modeling Problem
  - Levels of Abstraction
  - Metropolis
  - Our Work

- **Uniprocessor Modeling**

- **Multiprocessor Modeling**

- **Timing Annotation**

- **Conclusions and Future Work**
Motivating Trends

2004 IRTS Update
- Key system-level design challenges include:
  - Design space exploration
  - System-level estimation

Intel CTO Greg Spirakis
2003 EMSOFT Keynote
- Software is 80% of embedded system development cost
- more advanced research needed
  - Increase simulation speed of HLM for architectural exploration and HW/SW codesign

Multicore Explosion makes high level modeling even more important
- Cycle-Level Multicore simulation is HARD (and SLOW)
Focus of This Work

Modeling Approaches

中国传统方法
- 循环精确模型
  - 不可接受的性能
  - 长开发时间
  - 难以重定位
- 原型和仿真
  - 优秀的性能
  - 非常昂贵
  - 非常长的开发时间
  - 不灵活

抽象方法
- 放弃一些细节和准确性以换取开发和模拟的速度
- 分离通信和计算
- 通过函数调用进行通信

分析模型
- 算法模型
- 统计模型

事务层次模型（TLM）
- 高于 RTL，低于分析模型
- 通信和计算的分离

Focus of This Work
The Performance Modeling Problem

◆ Major Factors to Consider
  ♦ Accuracy
  ♦ Speed
  ♦ Creation Effort

◆ Other Factors to Consider
  ♦ Retargetability
  ♦ Flexibility
  ♦ Modularity
  ♦ Modeling other quantities (e.g. Power, Area, etc)
Levels of Abstraction: The Hardware View

Communication:
- Shared Variables
- Method Calls to Channels
- Wires and Registers

Languages:
- C/C++, Matlab
- SystemC, SpecC, Metropolis
- Verilog, VHDL
Levels of Abstraction: A Software/Processor View

Microarchitecture / Timing:
- None / None
- None / Instruction Counts
- “Full” / Cycle-Counts
- “Full” / Signal Level

Model Speed:
- Native Speed (GHz)
- ~3 – ~900 MIPS
- ~0.1 – ~30 MIPS
- ~0.1 – ~10 KHz
**My Work in Terms of Hardware and Software Views**

**Hardware View**
- Algorithmic Models
- Transaction-Level Models

**Software/CPU View**
- Algorithmic Models
- Instruction Level Models
- Cycle-Accurate Models

- Multiprocessor Modeling
- Uniprocessor Modeling

**Annotation Framework**
- Timing Annotated Application

**Register Transfer Level Models**
- Logic Gates
- Actual Gates
- Layout
**Metropolis Framework – Key Features**

- **Orthogonalization of Concerns**
  - Function – Architecture
  - Computation – Communication
  - Behavior – Performance

- **Flexible and Formal Semantics**
  - Can Represent a Wide Number of Models of Computation (MoCs)

- **Constraints**

- **Quantities**

- **Mapping**

Homepage: [http://embedded.eecs.berkeley.edu/metropolis](http://embedded.eecs.berkeley.edu/metropolis)
process P{
    port reader X;
    port writer Y;
    thread()
        while(true){
            ...
            z = f(X.read());
            Y.write(z);
        }
}

interface reader extends Port{
    update int read();
    eval int n();
}

interface writer extends Port{
    update void write(int i);
    eval int space();
}

medium M implements reader, writer{
    int storage;
    int n, space;
    void write(int z){
        await(space>0; this.writer ; this.writer)
        n=1; space=0; storage=z;
    }
    word read(){ ... }
}
**Metropolis: Architecture Components**

An architecture component specifies *services*, i.e.

- **what it *can* do**: interfaces
- **how much it *costs***: quantities, annotation, logic of constraints

```java
interface BusMasterService extends Port {
    update void busRead(String dest, int size);
    update void busWrite(String dest, int size);
}

interface BusArbiterService extends Port {
    update void request(event e);
    update void resolve();
}

medium Bus implements BusMasterService {
    port BusArbiterService Arb;
    port MemService Mem; ...
    update void busRead(String dest, int size) {
        if(dest== ... ) Mem.memRead(size);
        [[Arb.request(B(thisthread, this.busRead));
        GTime.request(B(thisthread, this.memRead),
            BUSCLKCYCLE +
            GTime.A(B(thisthread, this.busRead))));
        ]]
    }
    ...
}

scheduler BusArbiter extends Quantity implements BusArbiterService {
    update void request(event e){ ... }
    update void resolve() { //schedule }
}
```

Source: Yosinori Watanabe
**Metropolis: Architecture Netlist**

Architecture netlist specifies configurations of architecture components.

Each constructor
- instantiates arch. components,
- connects them,
- takes as input *mapping processes*.

Source: Yosinori Watanabe
Outline

- Introduction
- Uniprocessor Modeling
  - Overview
  - Related Work
  - Double Process Models
  - Results
- Multiprocessor Modeling
- Timing Annotation
- Conclusions and Future Work
Uniprocessor Modeling Overview

◆ Goals:
  - Extensibility and Flexibility
  - Simplicity
    - Be “Data Sheet” accurate
  - Separate Timing from Function
◆ Base MoC: Kahn Process Networks
  - Concurrent Processes
  - Communication via unbounded FIFOs
    - Blocking Reads / Unblocking Writes
  - Fully deterministic
  - Untimed
◆ Scalar ARM Processors Modeled
  - Strongarm – 5 stage pipeline
  - XScale – 7 stage pipeline
Uniprocessor Modeling Related Work

- Microarchitectural Simulators / Frameworks
  - SimpleScalar (Austin, et al)
  - Liberty (August, et al)
  - Operation State Machine (Simit-ARM) (Qin, Malik)

- ISS Technologies
  - Vendor Provided ISS’s
  - Compiled-Code ISS (e.g. VAST, LISAték)

- Architecture Description Languages (ADL’s)
  - Expression, LISAték, Tipi, Etc

- System Level Design Environments
  - e.g. Polis, Cosyma, VCC, CoWare
**Double Process Model**

- **Needed For:**
  - Modeling Forwarding
  - Modeling Variable Instruction Latencies

- **Leverages FIFO’s for modeling delays**
  - Pre-execution Delay
  - Execution Delay
  - Synchronization
  - Stalls
    - Issue Stalls
    - Result Stalls

---

Accuracy vs SimpleScalar

- Compared XScale and Strongarm models to SimpleScalar-ARM
  - With and Without Memory
  - Instruction traces from modified Sim-Safe ISS

- Based on four benchmarks
  - Three from MiBench*

## Performance vs SimpleScalar

<table>
<thead>
<tr>
<th></th>
<th>Base Metropolis</th>
<th>1. Optimized Metropolis</th>
<th>2. Optimized SystemC</th>
<th>Simplescalar ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrongARM Base</td>
<td>1997</td>
<td>5798</td>
<td>65537</td>
<td>916666</td>
</tr>
<tr>
<td>Xscale Base</td>
<td>2019</td>
<td>6086</td>
<td>67031</td>
<td>1013591</td>
</tr>
<tr>
<td>StrongARM Mem</td>
<td>1501</td>
<td>5865</td>
<td>60443</td>
<td>809229</td>
</tr>
<tr>
<td>Xscale Mem</td>
<td>1479</td>
<td>6036</td>
<td>60207</td>
<td>891007</td>
</tr>
</tbody>
</table>

**Model Performance (Cycles/Sec on a 3.0 GHz Xeon, Search-Large Benchmark)**

- **Optimizations**
  1. Results channels changed to fixed arrays
  2. Ported models to SystemC

- **Still ~15x performance gap**

- **Many optimizations are still possible**
  - Caching of previously decoded instructions
  - Trace preprocessing
Outline

◆ Introduction
◆ Uniprocessor Modeling
◆ Multiprocessor Modeling
  ◆ MuSIC Multiprocessor
  ◆ Architecture Model
  ◆ Functional Model
  ◆ Mapping
◆ Timing Annotation
◆ Conclusions and Future Work
**Baseband Processing Platform (MuSIC)**

- Each SIMD Core has 4 Control Processors
- MuSIC has a total of 19 Control Processors
- Control processors run a Multiprocessor OS
- Each Control Processor can run 1 thread at a time (i.e. Processor ~= Thread)

Multiple SIMD Cores (MuSIC)
Baseband Processing Platform Architecture Modeled in Metropolis (Simplified View)

- ARM Scheduler
- ARM µP
- ARM Processes
- Bus Bridge0
- Bus Interface0-1
- FIR Process
- CODEC Process
- Turbo/Viterbi
- RF Interface0-1
- RF Interface0-1 Schedulers
- RF Interface0-1
- Single connection
- Multiple connections
- System Bus 0
- System Bus 4
- System Bus 5
- System Bus 3
- System Bus 5 Schedulers
- System Bus 4 Schedulers
- Shared Mem0-3 Schedulers
- Shared Mem0-3
- SIMD Core 3
- SIMD Core 0
- SIMD Core 3 Processes
- SIMD Core 0 Processes
- SIMD3 Scheduler
- SIMD0 Scheduler
- Not Shown: Global Time, State Media, Sync Bus

KEY:
- Process
- Medium
- Quantity Manager
- Single connection
- Multiple connections
**Functionality: 802.11 Receive Payload Processing**

- **Sequence**: Coder0, Spreader0, Modulator0
- **Processing Chain**: Processing chain0, Processing chain1, Processing chain5
- **MAC** → **Rx_Splitter** → **Processing Chain** → **PHY Merger**
- **Channels**:
  - Data Channel
  - State Channel
Mapping Functionality onto Architecture

Mapping done via Synchronization
Outline

◆ Introduction
◆ Uniprocessor Modeling
◆ Multiprocessor Modeling
◆ Multiprocessor Timing Annotation
  ◦ Tool Flow
  ◦ Uniprocessor Timing Annotation – XScale and MuSIC
  ◦ Multiprocessor Timing Annotation – MuSIC Only
  ◦ Results
  ◦ Related Work
◆ Conclusions and Future Work

**Annotation Tool Flow**

- **Start (User Input)**
- **Multi-Threaded Application Code (C + RTOS API)**
- **Target Binary (EXE, ASM, DIS)**
- **Cycle Accurate Virtual Prototype (SystemC, C++)**
- **Source Level Timing Annotation Framework (Python)**
- **Timed Functional Simulator (SystemC/C++)**
- **Timing Annotated Multi-Threaded Application Code (SystemC + RTOS API)**

- **Host Compilation**
  - ((Untimed) Algorithmic)
- **Performance Traces**
- **Finish (Simulation Results)**
  - Host Compilation
  - (Annotated Algorithmic)
Uniprocessor Annotation Algorithm

1. Unify Assembly and Disassembly information into Blocks, Lines, Functions, and Files

2. Slice Blocks at Jumps in Processor Execution Trace

3. Calculate Block-Level Annotations
   - Add each block-level annotation to its line’s annotation

4. Generate Annotated Source Code
### Sample Code (fft64_test.c) Excerpt

```c
void main (int argc, char** argv) {
    ...
    short *InputI = (short *)Alloc_Mem(sizeof(short)*64); // line 23
    short *InputQ = (short *)Alloc_Mem(sizeof(short)*64); // line 24
    ...
}
```

### Annotated Code Excerpt (annotated_code/fft54_test.c)

```c
void main (int argc, char** argv) {
    ...
    Delay_Thread(577); // line 23 annotation
    short *InputI = (short *)Alloc_Mem(sizeof(short)*64); // line 23
    Delay_Thread(351); // line 24 annotation
    short *InputQ = (short *)Alloc_Mem(sizeof(short)*64); // line 24
    ...
}
```
### Assembly File
```
.Fmain:
.L1:
  sub r15, 0x1c
  push r8..r14
  add r15, r15, -0xcc

; End of Prologue
; **file '…/fft64_test.c', line 23
  pgen2 r2, 7
  mov r3, 4
  mov r4, 2
  jl .FAlloc_Mem
  nop
  nop
  nop
  nop
  mov r12, r2

; **line 24
  pgen2 r2, 7
  mov r3, 4
  mov r4, 2
```

### Disassembly File
```
[00020000] <.Fmain>:
  2f3c  sub r15, 28
[00020002]:
  e6de  push r8..r14
[00020004]:
  f7ff 1f34  add r15, r15, -0x00cc

[00020008]:
  0287  pgen2 r2, 7
[0002000a]:
  f534  mov r3, 4
[0002000c]:
  f542  mov r4, 2
[0002000e]:
  ecc1 0b78  jl .FAlloc_Mem:0x0216f0
[00020012]:
  0000  nop
[00020014]:
  0000  nop
[00020016]:
  0000  nop
[00020018]:
  0000  nop
[0002001a]:
  0000  nop
[0002001c]:
  0000  nop
[0002001e]:
  fac2  mov r12, r2

[00020200]:
  0287  pgen2 r2, 7
[00020202]:
  f534  mov r3, 4
[00020204]:
  f542  mov r4, 2
[00020208]:  ...
"22421: 0.pc=0020200" means:
Processor 0 fetches instruction 0x0020200 at cycle 22421
### Annotation Example: Block Slicing at Jumps

<table>
<thead>
<tr>
<th>PROCESSOR EXECUTION TRACE</th>
<th>PROGRAM ASSEMBLY WITH ADDRESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>22421: 0.pc=00020200</td>
<td>[00020200] &lt;.Fmain&gt;:</td>
</tr>
<tr>
<td>22422: 0.pc=00020202</td>
<td>2f3c sub r15, 28</td>
</tr>
<tr>
<td>22435: 0.pc=00020204</td>
<td>[00020202]: e6de push r8..r14</td>
</tr>
<tr>
<td>...... Begin Line 23</td>
<td>[00020204]: f7ff 1f34 add r15, r15, -0x00cc</td>
</tr>
<tr>
<td>22436: 0.pc=00020208</td>
<td>[00020208]: 0287 pgen2 r2, 7</td>
</tr>
<tr>
<td>22437: 0.pc=0002020a</td>
<td>[0002020a]: f534 mov r3, 4</td>
</tr>
<tr>
<td>22445: 0.pc=0002020c</td>
<td>[0002020c]: f542 mov r4, 2</td>
</tr>
<tr>
<td>22446: 0.pc=0002020e</td>
<td>[0002020e]: ecc1 0b78 jal .FAlloc_Mem:0x0216f0</td>
</tr>
<tr>
<td>22447: 0.pc=00020212</td>
<td>[00020212]: 0000 nop</td>
</tr>
<tr>
<td>22448: 0.pc=00020214</td>
<td>[00020214]: 0000 nop</td>
</tr>
<tr>
<td>22449: 0.pc=00020216</td>
<td>[00020216]: 0000 nop</td>
</tr>
<tr>
<td>...Alloc_Mem Function Execution</td>
<td>[00020218]: 0000 nop</td>
</tr>
<tr>
<td>23012: 0.pc=0002021e</td>
<td>[0002021a]: 0000 nop</td>
</tr>
<tr>
<td>...... End Line 23</td>
<td>[0002021c]: 0000 nop</td>
</tr>
<tr>
<td>...... Begin Line 24</td>
<td>[0002021e]: fac2 mov r12, r2</td>
</tr>
<tr>
<td>23019: 0.pc=00020220</td>
<td>[00020220]: 0287 pgen2 r2, 7</td>
</tr>
<tr>
<td>23020: 0.pc=00020222</td>
<td>[00020222]: f534 mov r3, 4</td>
</tr>
<tr>
<td>23021: 0.pc=00020224</td>
<td>[00020224]: f542 mov r4, 2</td>
</tr>
<tr>
<td>23022: 0.pc=00020226</td>
<td>[00020226]: ecc1 0b78 jal .FAlloc_Mem:0x0216f0</td>
</tr>
<tr>
<td>23023: 0.pc=0002022a</td>
<td>[0002022a]: 0000 nop</td>
</tr>
<tr>
<td>23024: 0.pc=0002022c</td>
<td>[0002022c]: 0000 nop</td>
</tr>
<tr>
<td>23031: 0.pc=0002022e</td>
<td>[0002022e]: 0000 nop</td>
</tr>
<tr>
<td>...Alloc_Mem Function Execution</td>
<td>[00020230]: 0000 nop</td>
</tr>
<tr>
<td>23353: 0.pc=00020236</td>
<td>[00020232]: 0000 nop</td>
</tr>
<tr>
<td>...... End Line 24</td>
<td>[00020234]: 0000 nop</td>
</tr>
<tr>
<td></td>
<td>[00020236]: fab2 mov r11, r2</td>
</tr>
</tbody>
</table>
Line-Level Annotations

- Internal and External Costs for Cycles
  - Internal:
    - Cycles of blocks inside of the line for the current iteration
  - External:
    - Cycles before this line (and after the previous block) and
    - Cycles between blocks for the current iteration

- Detecting Iteration Count: (Approximate Solution)
  - Track internal blocks’ iteration counts, and make line iteration count equal to the maximum of these iteration counts

- Handling Loops
  - Search source code for lines beginning with “for” or “while”
  - Detect initialization statement and treat it separately
Generate Annotated Source Code: Different Cases

- **Normal Case**
  
  ```c
  Delay_Thread(D_{\text{statement1}});
  \langle\text{statement1}\rangle
  Delay_Thread(D_{\text{statement2}});
  \langle\text{statement2}\rangle
  ...
  ```

- **While Loop Case**
  
  ```c
  Delay_Thread(D_{\text{test}});
  while (\langle\text{test}\rangle) {
    Delay_Thread(D_{\text{test}});
    Delay_Thread(D_{\text{body}});
    \langle\text{body}\rangle
  }
  ```

- **Do-While Loop Case (Normal Case)**
  
  ```c
  do {
    Delay_Thread(D_{\text{body}});
    \langle\text{body}\rangle
    Delay_Thread(D_{\text{test}});
  } while (\langle\text{test}\rangle);
  ```

- **For Loop Case**
  
  ```c
  Delay_Thread(D_{\text{init}} + D_{\text{test}});
  for( \langle\text{init}\rangle; \langle\text{test}\rangle; \langle\text{update}\rangle) {
    Delay_Thread(D_{\text{body}});
    \langle\text{body}\rangle
    Delay_Thread(D_{\text{update}} + D_{\text{test}});
  };
  ```
**Multiprocessor Annotation Algorithm**

1. Calculate each processor’s block and line-level annotations as before except:
   - Special care with startup delays
   - Special handling of inter-processor functions
     - Ignore their delays
     - Substitute in characterized delays

2. Unify line-level annotations from all processors

3. Generate annotated source code
Sample Multiprocessor Application

◆ Creates Threads until limit is reached
◆ Motivates Special handling of
  • Startup Delays
  • Inter processor communication

static int val = 0;
void main(void) {
    thread* next_thread;
    val++; // line 20
    Delay_Thread(<line22_delay>);
    if (val < NUM_PROCS) { // line 22
        next_thread = Create_Thread(main); // line 24
    }
    ...  
}  
Thread Test Initial Code

End of Program
**Multiprocessor Annotation: Handling Startup Delays**

**Initial Code**

```c
static int val = 0;

void main(void) {
    thread* next_thread;
    Delay_Thread(<startup_delay>);
    Delay_Thread(<line20_delay>);
    val++;    // line 20
    Delay_Thread(<line22_delay>);
    if (val < NUM_PROCS) { // line 22
        Delay_Thread(<line24_delay>); next_thread =
        Create_Thread(main); // line 24
    }
    ...
}
```

**Problem: Startup delay will occur with each thread creation!!**

**Fixed Annotated Initial Code**

```c
static int val = 0;
static int started_up = 0;

void main(void) {
    thread* next_thread;
    if (started_up == 0) {
        started_up = 1;
        Delay_Thread(<startup_delay>);
    }
    Delay_Thread(<line20_delay>);
    val++;    // line 20
    Delay_Thread(<line22_delay>);
    if (val < NUM_PROCS) { // line 22
        Delay_Thread(<line24_delay>); threads[val] =
        Create_Thread(main); // line 24
    }
    ...
}
```
void main(void) {
    thread* next_thread;
    ...
    next_thread = Create_Thread(main);
    wait_exit(next_thread);
    exit();
}

Program Properties:
- Wait_exit, Startup delays = 0
- Create_Thread, Exit delays = 1000
- System has 5 total processors
- Based on this Actual Delay = 10000

void main(void) {
    thread* next_thread;
    ...
    Delay_Thread(1000);
    next_thread = Create_Thread(main);

    Delay_Thread(4000);
    wait_exit(next_thread);

    Delay_Thread(1000);
    exit();
}

Results:
- Delay = 14000
- Error = 40%

void main(void) {
    thread* next_thread;
    ...
    next_thread = Create_Thread(main);

    wait_exit(next_thread);

    Delay_Thread(1000);
    exit();
}

Characterization-Based Annotated Code

Results:
- Delay = 10000
- Error = 0%
Multiprocessor Annotation: Why Characterization? (2 of 2)

- Overlap between multiple processor threads isn’t captured in annotation

- Processor Execution Trace is approximate
  - Only gives fetch times
  - No pipelining
  - This means that intrinsic instructions can magnify errors

```c
void main(void) {
    thread* next_thread;
    ...
    Delay_Thread(1000);
    next_thread = Create_Thread(main);
    wait_exit(next_thread);
    Delay_Thread(4000);
    Delay_Thread(1000);
    exit();
}
```

Results:
- Delay = 34000
- Error = 240%

Direct Measurement Annotated Code (v2)
MiBench Uniprocessor Accuracy for XScale and MuSIC

**XScale Error Magnitudes**
- Average: 3.76%
- Maximum: 11.06%

**MuSIC Error Magnitudes**
- Average: 3.85%
- Maximum: 17.46%
**MiBench Uniprocessor Speedup for XScale and MuSIC**

### Annotation Speedup on MiBench

<table>
<thead>
<tr>
<th>Function</th>
<th>XScale Small Speedup</th>
<th>XScale Large Speedup</th>
<th>MuSIC Small Speedup</th>
<th>MuSIC Large Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm encode</td>
<td>2250</td>
<td>5900</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>adpcm decode</td>
<td>110</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>dijkstra</td>
<td>2</td>
<td>1</td>
<td>290</td>
<td>1900</td>
</tr>
<tr>
<td>patricia</td>
<td>5900</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>rijndael encode</td>
<td>16</td>
<td>1030</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>rijndael decode</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sha</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>stringsearch</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>average</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

### Averages

2.5 GHz
Multiprocessor Results

◆ Internal Tests
  • Direct Measurement is much less accurate than Characterization Based because of double counting.
  • For streaming_test benchmark
    ♦ Different data |error| <1.0%
    ♦ Different # of worker threads |error| <3.1%

◆ 5 Thread JPEG Encoder
  • Max. and Avg. |error| are: ~8%
  • Speedup of 14 – 18x
  • Changing image sizes didn’t impact error

◆ Annotated Code was 2x – 5x slower than Non-Annotated Code

<table>
<thead>
<tr>
<th>Test</th>
<th>Thread Count</th>
<th>Direct Meas. Error %</th>
<th>Char. Based Error %</th>
<th>Char. Based. Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>message test</td>
<td>3</td>
<td>0.00%</td>
<td>0.21%</td>
<td>80.1</td>
</tr>
<tr>
<td>streaming test</td>
<td>4</td>
<td>48.77%</td>
<td>0.18%</td>
<td>207.5</td>
</tr>
<tr>
<td>thread test</td>
<td>19</td>
<td>767.24%</td>
<td>-2.24%</td>
<td>526.9</td>
</tr>
</tbody>
</table>

| avg. magnitude | 272.01% | 0.87% | 271.5 |
| max. magnitude  | 767.24% | 2.24% | 526.9 |

<table>
<thead>
<tr>
<th>Num. of Rows and Columns</th>
<th>Error</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>-8.18%</td>
<td>14.00</td>
</tr>
<tr>
<td>64</td>
<td>-7.56%</td>
<td>18.49</td>
</tr>
<tr>
<td>160</td>
<td>-7.39%</td>
<td>17.03</td>
</tr>
</tbody>
</table>

| avg. magnitude | 7.71% | 16.51 |
| max magnitude  | 8.18% | 18.49 |

Internal Multiprocessor Results

JPEG results for different image sizes
Scaling of Annotations

For MuSIC: Annotator runtime ranged from 0.5x to 3x of VP runtime

For XScale: Annotator runtime ranged from 7.5x to 75x of VP runtime
Timing Annotation Related Work

◆ Software Performance Estimation
  - Polis (UC Berkeley)
    ♦ Based on CFSM and S-Graphs
  - CABA – Lavagno and Lazarescu
    ♦ Virtual Compilation
    ♦ Object Code Based

◆ Worst-Case Execution Time Analysis
  - Cinderella (Li and Malik)
  - AbsInt (Wilhelm, et al)

◆ MESH – Cassidy, Paul, Thomas
  - High Level Simulation Framework for Multiprocessor Systems
  - Hand-annotation of programs

◆ Compiled Code ISS (VaST, LISAtrek, etc)
  - Fast and Accurate
  - Time consuming to create and modify
  - Multi-processors can still be slow

◆ Profiling
  - General Profiling (e.g. Gprof)
    ♦ Instrumentation + sampling
  - Micro-profiling (Kempf, Meyr, et al)
    ♦ Instrumentation at an IR Level
    ♦ ~9x Faster than Instruction Level Simulation
    ♦ 80% - 98% accuracy

◆ Other
  - FastVeri from Interdesign Technologies
Contributions

◆ Uniprocessor Modeling
  ◆ Intuitive, Accurate, and Retargetable
  ◆ Still needs more optimization

◆ Multiprocessor Modeling
  ◆ Simple, generic models that are highly reusable
  ◆ But, needs a direct connection pre-existing models

◆ Multiprocessor Source-Level Timing Annotation
  ◆ Framework is Highly Generic and Retargetable
  ◆ Extension of Functional Simulator to SystemC
  ◆ Achieved Good Speedup and Accuracy vs. Virtual Prototype
My Work in Terms of Hardware and Software Views

Hardware View
- Algorithmic Models
- Transaction-Level Models

Multiprocessor Modeling
Uniprocessor Modeling

Software/CPU View
- Algorithmic Models
- Instruction Level Models
- Cycle-Accurate Models

Annotation Framework
Timed Instruction Trace
Application Code and Binary

Register Transfer Level Models
Logic Gates
Actual Gates
Layout
Future Work in Terms of Hardware and Software Views

- **Hardware View**
  - Algorithmic Models
  - Transaction-Level Models
  - Multiprocessor Modeling
  - Uniprocessor Modeling
  - Register Transfer Level Models
  - Logic Gates
  - Actual Gates
  - Layout

- **Software/CPU View**
  - Algorithmic Models
  - Instruction Level Models
  - Cycle-Accurate Models
  - Extended Annotation Framework
  - Timing + Communication
  - Annotated Application
  - Timed Instruction and Memory Trace
  - Application Code and Binary
Acknowledgements

◆ **Microprocessor Modeling**
  - Intel Corporation for SRC sponsorship
    - Summer Internship
    - Mentors: Timothy Kam and Mike Kishinevsky
    - Berkeley Liaison: John Moondanos
  - Other Mentors
    - Luciano Lavagno, Yoshi Watanabe, *Cadence Berkeley Labs*
    - Kees Vissers
  - Student Collaborators
    - Sam Williams, Min Chen, Qi Zhu, and Haibo Zeng

◆ **Annotation Work**
  - Direct Collaboration with Mirko Sauermann and Dominik Langen at Infineon
  - Software Licenses from CoWare
Thank You!
**SystemC**

- IEEE Standardized C++-based Library for modeling Hardware and/or Software Systems
  - Open Source Reference Implementation available at: [http://www.systemc.org](http://www.systemc.org)
  - Widely Supported in EDA and IP-Exchange Industries
- **Version 1.0** – Primarily for accelerating RTL-level simulations
  - Primitive Channels – Wires, Registers
- **Version 2.2** – Support for complicated channels and other higher level concepts
  - Higher-level Interface-based Channels
  - Dynamic Process Control Constructs
- **Official Libraries for:**
  - Verification and Testing
  - Transaction-Level Modeling
void main (int argc, char** argv)
{
    ...
    if (started_up == 0) {
        started_up = 1; Delay_Thread(22436); // processor startup delay
    }
    Delay_Thread(577); // line 23 annotation
   short *InputI = (short *)Alloc_Mem(sizeof(short)*64); // line 23
    Delay_Thread(351); // line 24 annotation
    short *InputQ = (short *)Alloc_Mem(sizeof(short)*64); // line 24
    ...

Annotated Code Excerpt (annotated_code/fft54_test.c)

- Annotated Delay Within 0.1% of actual results
## Levels of Abstraction

<table>
<thead>
<tr>
<th>Level</th>
<th>Speed / Accuracy</th>
<th>Function / Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic</td>
<td>Excellent / None</td>
<td>Concurrent / None</td>
</tr>
<tr>
<td>Annotated Algorithmic</td>
<td>Excellent / Good</td>
<td>Concurrent + Delays / None</td>
</tr>
<tr>
<td>Abstract (TLM) Model</td>
<td>Good / Poor</td>
<td>Concurrent / Timed Resources</td>
</tr>
<tr>
<td>Annotated Abstract Model</td>
<td>Good / Very Good</td>
<td>Concurrent + Delays / Timed Resources</td>
</tr>
<tr>
<td>Cycle-Level (Virtual Prototype)</td>
<td>OK / Cycle Level</td>
<td>ASM / SystemC</td>
</tr>
<tr>
<td>RTL-level</td>
<td>Horrible / Signal Level</td>
<td>ASM / SystemC</td>
</tr>
</tbody>
</table>
Single Process Model*

- Add hazard detection and bubble insertion (stalls)
- Parameterize the pipeline depth
- Add a branch predictor
  - Pass prediction and PC down pipeline
  - Resolve branch when it commits
- Single Process Execution Order
  1. Read operands
  2. Execute
  3. Write to register file
- Synchronous Assumption

* Collaboration With: Sam Williams
**MuSIC Platform Software**

- **ILTOS Multiprocessor RTOS**
  - Architecture Specific Functions
  - Basic Thread Operations (Create, Exit)
  - Inter-Thread Messaging
  - Synchronization Primitives (Events, Mutexes, Etc)

- **API-Compatible Simulator**
  - ILTOS API implemented on Windows API
  - Ported to SystemC for timing annotation*

- **Highly Optimized SystemC Cycle Accurate Virtual Prototype**
  - Parameterizable (# CPUs, Tracing, Accuracy of Models, etc)
  - Uses JIT Cache-Compiled simulator from CoWare

---

**ILTOS Simulation Targets**

- **Multithreaded Application**
  - MuSIC Compiler
  - x86 Compiler

- **Virtual Prototype**
  - SystemC Lib
  - Sim Host (PC)
  - Cycle Accurate, but Slow

- **API-level Simulator**
  - SystemC Lib
  - Sim Host (PC)
  - No Accuracy, but Fast

---

* Collaboration With: Mirko Sauermann @ Infineon
Performance Backwards Annotation: An Example

```
Startup_code();
for (x < num_strings) {
    initsearch(find_str[x]);
    strsearch(srch_str[x]);
}
return 0;
```
# Characterization Example Actual Execution

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>Create Thread Begin</th>
<th>Wait Exit Begin</th>
<th>Wait Exit End</th>
<th>Exit End</th>
<th>Full Wait Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>9000</td>
<td>10000</td>
<td>8000</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>2000</td>
<td>8000</td>
<td>9000</td>
<td>6000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>3000</td>
<td>7000</td>
<td>8000</td>
<td>4000</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
<td>4000</td>
<td>6000</td>
<td>7000</td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
<td>5000</td>
<td>5000</td>
<td>6000</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4000</td>
</tr>
</tbody>
</table>

**Program Properties:**
- Wait_exit, Startup delays = 0
- Create_Thread, Exit delays = 1000
- System has 5 total processors
- Based on this Actual Delay = 10000

```c
void main(void) {
    thread* next_thread;
    ...
    next_thread = Create_Thread(main);
    wait_exit(next_thread);
    exit();
}
```

*Initial Code*
Characterization Example Direct Measurement Execution

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>Create Thread Begin</th>
<th>Wait Exit Begin</th>
<th>Wait Exit End</th>
<th>Exit End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>13000</td>
<td>14000</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>2000</td>
<td>12000</td>
<td>13000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>3000</td>
<td>11000</td>
<td>12000</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
<td>4000</td>
<td>10000</td>
<td>11000</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
<td>5000</td>
<td>9000</td>
<td>10000</td>
</tr>
</tbody>
</table>

void main(void) {
    thread* next_thread;
    ...
    Delay_Thread(1000);
    next_thread = Create_Thread(main);
    Delay_Thread(4000);
    wait_exit(next_thread);
    Delay_Thread(1000);
    exit();
}  

Results:
Delay = 14000
Error = 40%
Characterization Example Characterization-Based Execution

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>Create Thread Begin</th>
<th>Wait Exit Begin</th>
<th>Wait Exit End</th>
<th>Exit End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>9000</td>
<td>10000</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>2000</td>
<td>8000</td>
<td>9000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>3000</td>
<td>7000</td>
<td>8000</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
<td>4000</td>
<td>6000</td>
<td>7000</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
<td>5000</td>
<td>5000</td>
<td>6000</td>
</tr>
</tbody>
</table>

```c
void main(void) {
    thread* next_thread;
    ...
    next_thread = Create_Thread(main);
    wait_exit(next_thread);
    Delay_Thread(1000);
    exit();
}  
Characterization-Based Annotated Code
```

Results:
- Delay = 10000
- Error = 0%
### Characterization Example Direct Measurement Execution: With Wait Delay Pushed to After the Wait Function

<table>
<thead>
<tr>
<th>Thread Id</th>
<th>Create Thread Begin</th>
<th>Create Thread Exit</th>
<th>Create Thread Exit End</th>
<th>Exit End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>28000</td>
<td>34000</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>2000</td>
<td>22000</td>
<td>28000</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>3000</td>
<td>16000</td>
<td>22000</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
<td>4000</td>
<td>10000</td>
<td>16000</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
<td>5000</td>
<td>5000</td>
<td>10000</td>
</tr>
</tbody>
</table>

```c
void main(void) {
    thread* next_thread;
    ...
    Delay_Thread(1000);
    next_thread = Create_Thread(main);
    wait_exit(next_thread);
    Delay_Thread(4000);
    Delay_Thread(1000);
    exit();
}
```

**Results:**
- Delay = 34000
- Error = 240%
Internal Uniprocessor MuSIC Tests

- Compares Direct Measurement and Characterization-Based Approaches
- Speedup ranges from 11x to 139x
- For Different Data
  - Good for larger # of iterations (within 3%)
  - Smaller # causes bad estimation due to cache impact (within 50%)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Direct Meas. Error %</th>
<th>Char. Based Error %</th>
<th>Char. Based Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_test</td>
<td>0.00%</td>
<td>-0.59%</td>
<td>33</td>
</tr>
<tr>
<td>dhrystone</td>
<td>-0.09%</td>
<td>-0.21%</td>
<td>47</td>
</tr>
<tr>
<td>libc_test</td>
<td>-0.46%</td>
<td>-0.46%</td>
<td>94</td>
</tr>
<tr>
<td>payload1</td>
<td>-0.02%</td>
<td>4.41%</td>
<td>33</td>
</tr>
<tr>
<td>ratematch</td>
<td>-0.04%</td>
<td>-0.04%</td>
<td>11</td>
</tr>
<tr>
<td>syncmng_test</td>
<td>0.00%</td>
<td>1.59%</td>
<td>13</td>
</tr>
<tr>
<td>udt_test</td>
<td>-0.02%</td>
<td>-0.02%</td>
<td>25</td>
</tr>
<tr>
<td>fft64</td>
<td>0.00%</td>
<td>-0.03%</td>
<td>48</td>
</tr>
<tr>
<td>cck_test</td>
<td>-0.15%</td>
<td>-0.15%</td>
<td>139</td>
</tr>
<tr>
<td>udt_test2</td>
<td>0.00%</td>
<td>0.00%</td>
<td>23</td>
</tr>
<tr>
<td>avg. magnitude</td>
<td>0.08%</td>
<td>0.75%</td>
<td>47</td>
</tr>
<tr>
<td>max. magnitude</td>
<td>0.46%</td>
<td>4.41%</td>
<td>139</td>
</tr>
</tbody>
</table>
**Uniprocessor MiBench Results**

**Information**
- Run on benchmarks that compiled with minimal changes and ran properly
- Ran on small and large data sets

**Speedup of 15x-1030x vs. VP**

**Accuracy**
- For same data |error| is: max 1%, avg. ~3%
- For different data |error| stayed almost the same (<0.1% change)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Small Results</th>
<th></th>
<th>Large Results</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error (%)</td>
<td>Speedup</td>
<td>Error (%)</td>
<td>Speedup</td>
</tr>
<tr>
<td>adpcm.encode</td>
<td>-1.67%</td>
<td>16.3</td>
<td>-0.70%</td>
<td>16.3</td>
</tr>
<tr>
<td>adpcm.decode</td>
<td>-2.25%</td>
<td>15.6</td>
<td>-1.31%</td>
<td>40.1</td>
</tr>
<tr>
<td>dijkstra</td>
<td>-12.63%</td>
<td>25.7</td>
<td>-17.46%</td>
<td>27.5</td>
</tr>
<tr>
<td>patricia</td>
<td>-0.47%</td>
<td>81.6</td>
<td>-1.18%</td>
<td>65.5</td>
</tr>
<tr>
<td>rijndael.encode</td>
<td>-1.26%</td>
<td>129.9</td>
<td>-2.96%</td>
<td>229.6</td>
</tr>
<tr>
<td>rijndael.decode</td>
<td>-6.52%</td>
<td>159.1</td>
<td>-1.69%</td>
<td>234.5</td>
</tr>
<tr>
<td>sha</td>
<td>0.00%</td>
<td>1,030.8</td>
<td>0.00%</td>
<td>984.4</td>
</tr>
<tr>
<td>stringsearch</td>
<td>3.85%</td>
<td>14.6</td>
<td>-13.95%</td>
<td>28.7</td>
</tr>
<tr>
<td>avg. magnitude</td>
<td>2.80%</td>
<td>184.2</td>
<td>4.91%</td>
<td>203.3</td>
</tr>
<tr>
<td>max. magnitude</td>
<td>12.63%</td>
<td>1,030.8</td>
<td>17.46%</td>
<td>984.4</td>
</tr>
</tbody>
</table>

MiBench results for large and small data sets
# Assumptions and Limitations

## Assumptions

- Execution delays of individual lines are close to fixed.
- All code of interest available as:
  - C source code
  - Debug-compiled ASM and EXE
- Good coding practices are used
  - No goto’s
  - 1 statement per line
  - If, for, do-while all have { }’s

## Limitations

- Doesn’t handle optimized code
- Some annotations can be placed illegally
- Memory traffic and value-dependent execution times are not modeled
- Trace files can get HUGE!