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

Trevor Meyerowitz, Phd Candidate, UC Berkeley

tcm@eecs.berkeley.edu

February 26, 2008

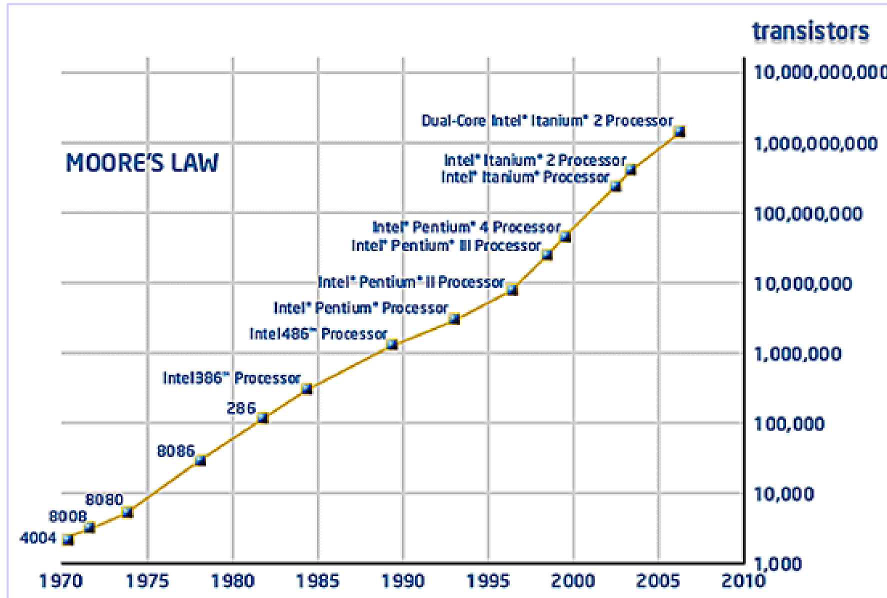
Advisor: Alberto Sangiovanni-Vincentelli

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

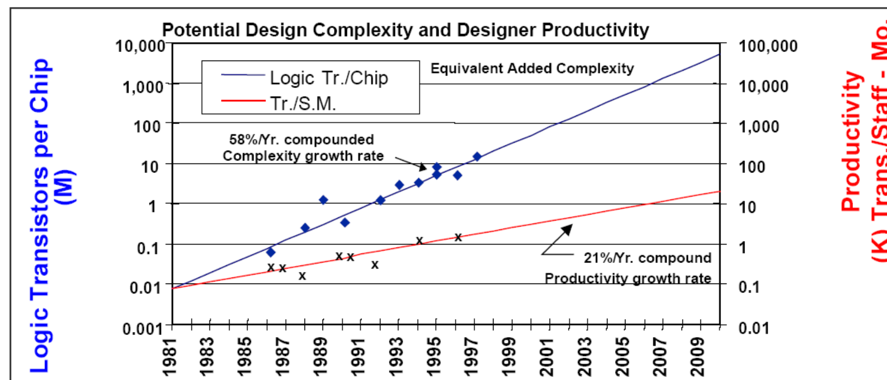
Outline

- Introduction
  - Motivation
  - The Performance Modeling Problem
  - Levels of Abstraction
  - Metropolis
  - Our Work
- Uniprocessor Modeling
- Multiprocessor Modeling
- Timing Annotation
- Conclusions and Future Work

Motivating Trends



Moore's Law (Source: Intel)



The Design Productivity Gap (Source: 1999 ITRS Roadmap)

- **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)**

Modeling Approaches

- **Traditional Approaches**
  - Cycle-Accurate Models
    - Unacceptable Performance
    - Long Development Time
    - Hard to Retarget
  - Prototypes and Emulation
    - Great Performance
    - Very Expensive
    - Very Long Development Time
    - Not Flexible
- **Abstract Approaches**
  - Sacrifices some Detail and Accuracy for Speed of Development and Simulation
  - Analytical Models
    - Algebraic Models
    - Statistical Models
  - Transaction-Level Models (TLM)
    - Higher Than RTL, lower than Analytical Models
    - Separation of Communication and Computation
    - Communication via function-calls

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

Algorithmic Models

Transaction-Level Models

Register Transfer Level Models

Logic Gates

Actual Gates

Layout

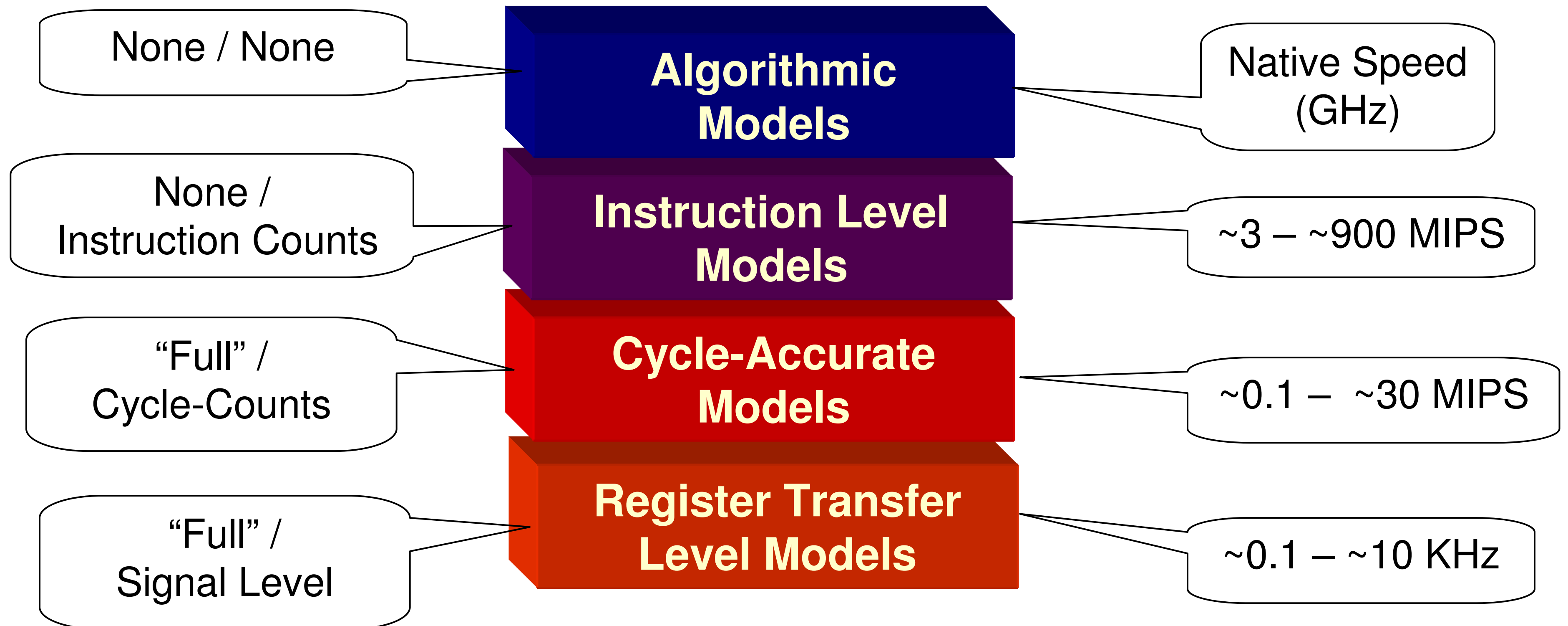
Languages:

C/C++, Matlab

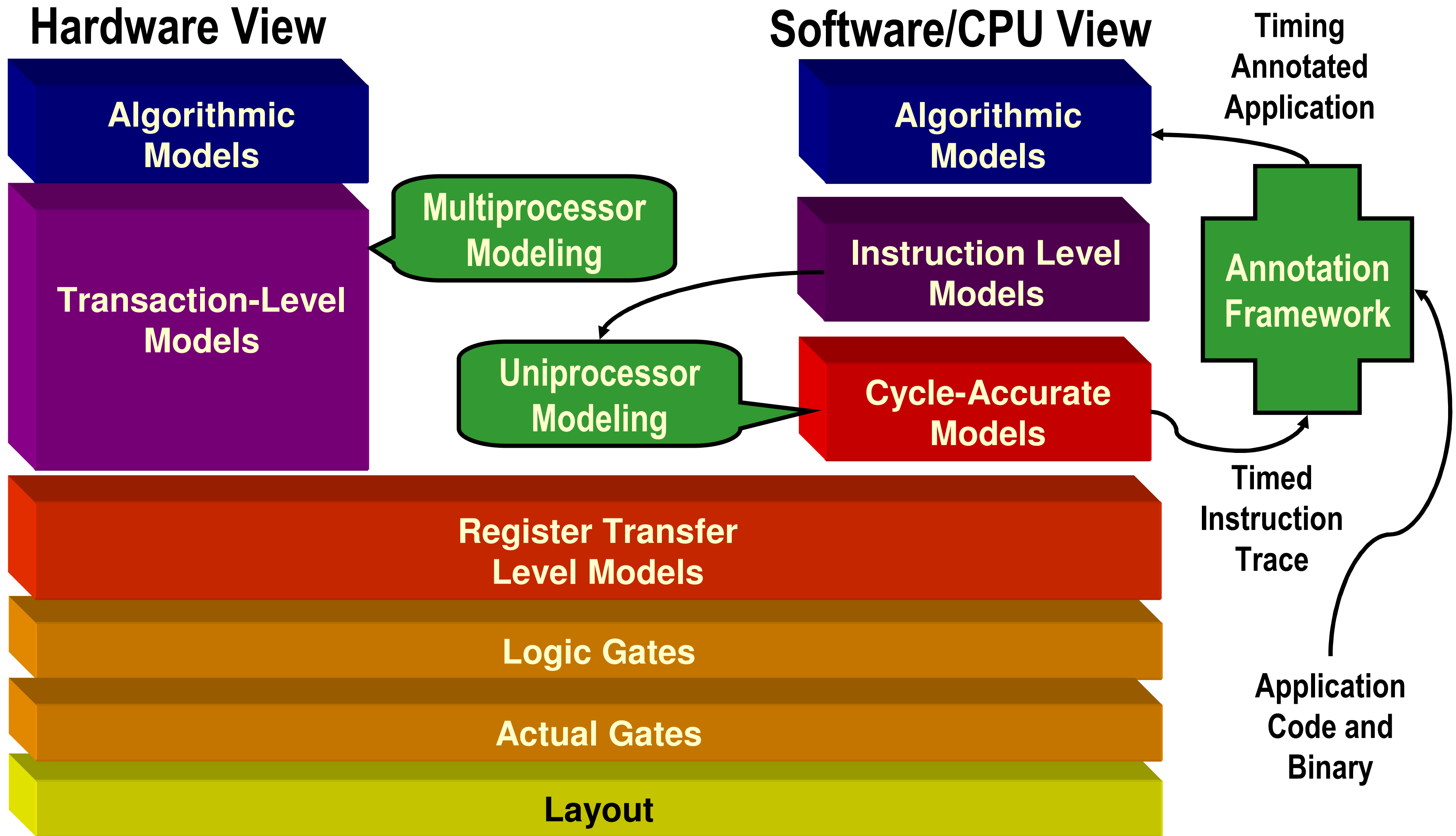
SystemC, SpecC, Metropolis

Verilog, VHDL

Levels of Abstraction: A Software/Processor View

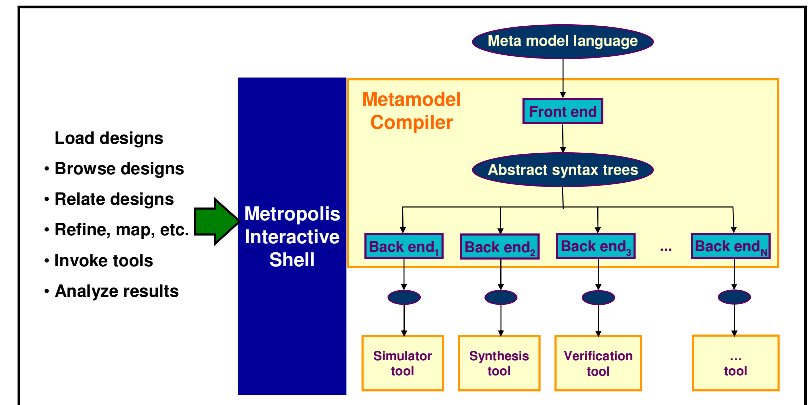


My Work in Terms of Hardware and Software Views

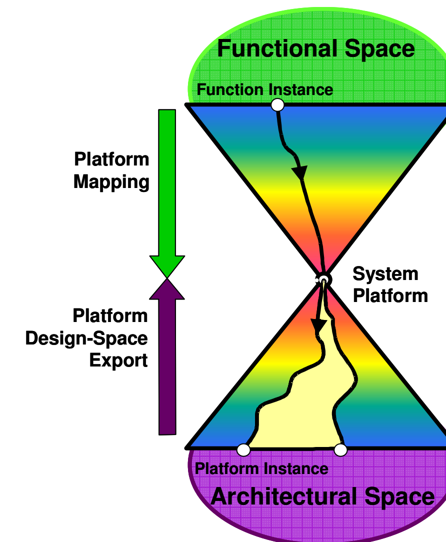


Metropolis Framework – Key Features

- 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



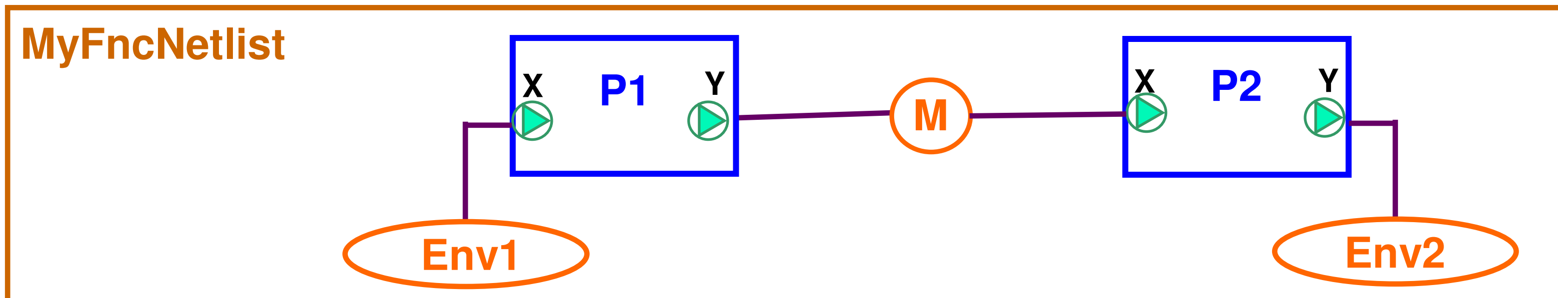
Metropolis Tool Framework



Platform-Based Design (ASV Triangles)

Homepage: http://embedded.eecs.berkeley.edu/metropolis

Metropolis : Functionality Example




```
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(){ ... }
}
```

Source: Yosinori Watanabe

Metropolis: Architecture Components

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

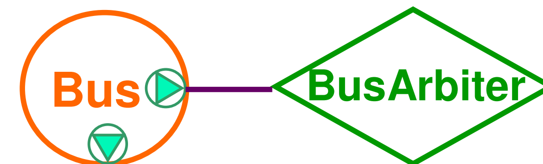
- what it *can* do : interfaces
- how much it *costs* : quantities, annotation, logic of constraints

```
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 }
 }
```



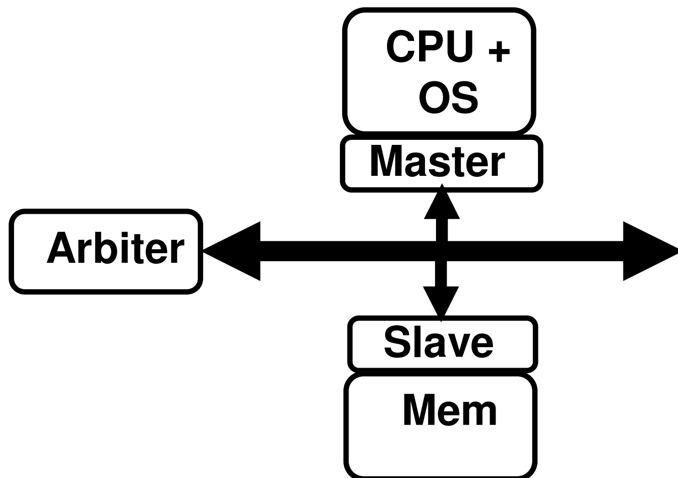
Source: Yosinori Watanabe

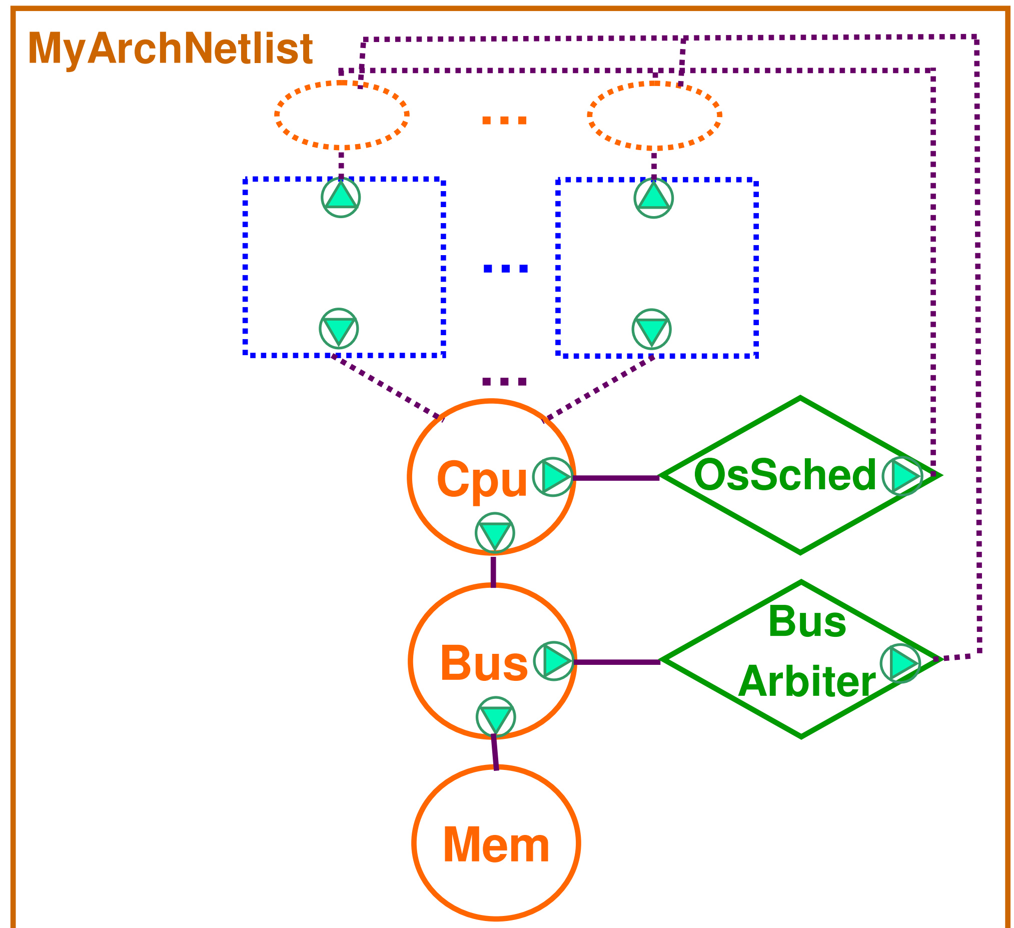
Metropolis: Architecture Netlist

Architecture netlist specifies configurations of architecture components.

Each constructor

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





Source: Yosinori Watanabe

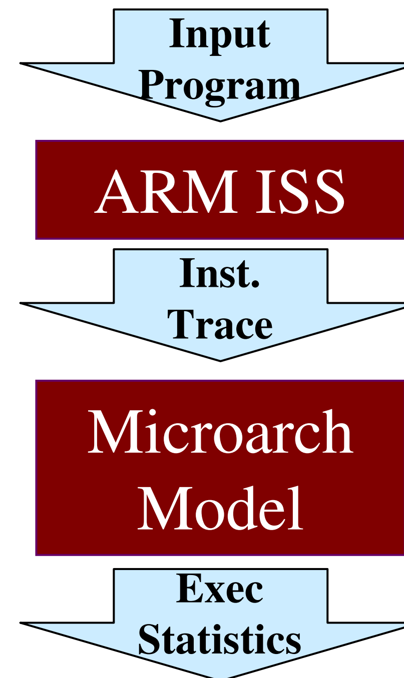
Outline

- 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

Tool Flow



Uniprocessor Modeling Related Work

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

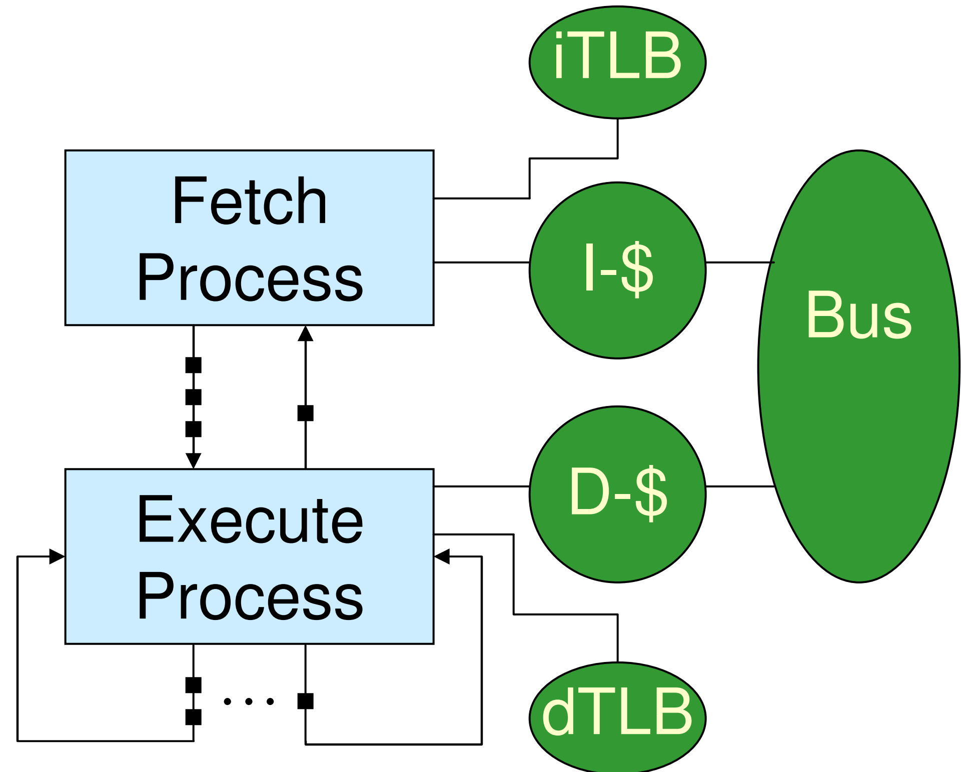
- **ISS Technologies**
  - **Vendor Provided ISS's**
  - **Compiled-Code ISS (e.g. VAST, LISAtek)**

- **Architecture Description Languages (ADL's)**
  - **Expression, LISAtek, Tipi, Etc**

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

Double Process Model

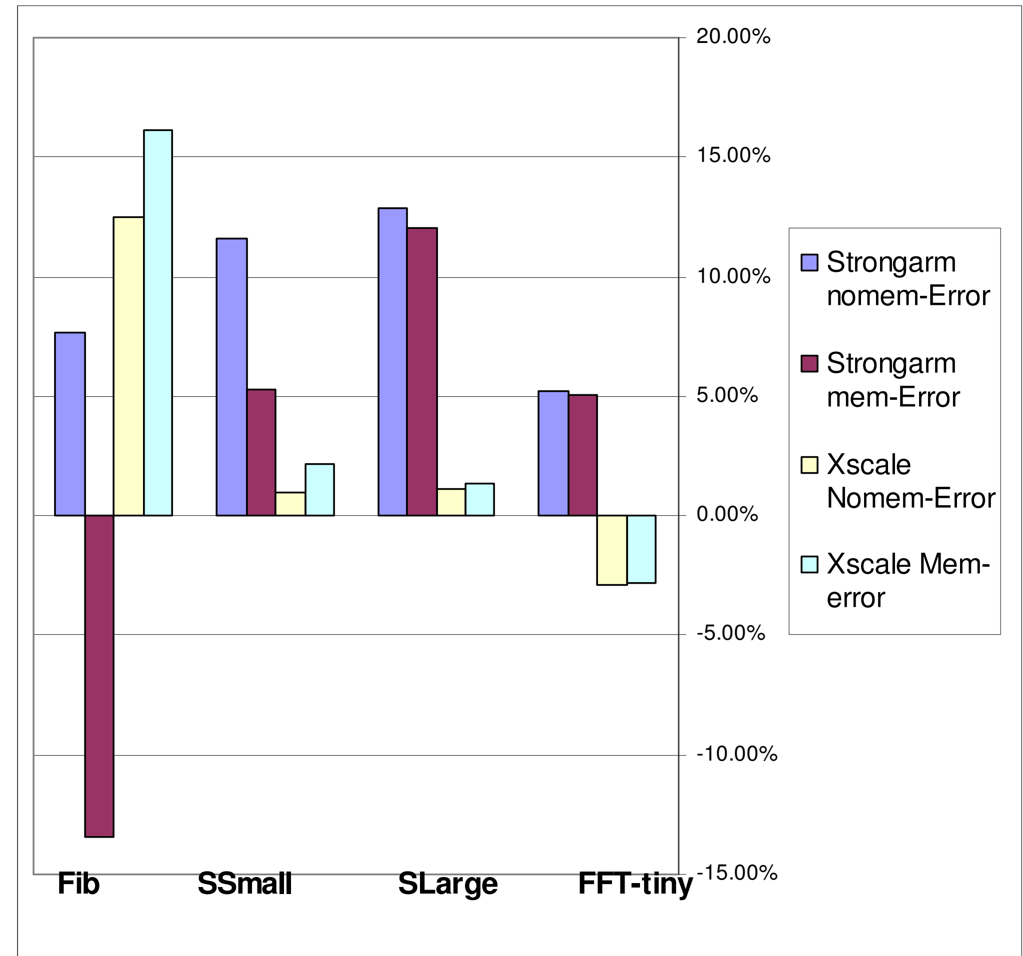
- **Needed For:**
  - **Modeling Forwarding**
  - **Modeling Variable Instruction Latencies**
- **Leverages FIFO's for modeling delays**
  - **Pre-execution Delay**
  - **Execution Delay**
  - **Synchronization**
  - **Stalls**
    - **Issue Stalls**
    - **Result Stalls**



T. Meyerowitz, A. Sangiovanni-Vincentelli,"High Level CPU Micro-architecture Models Using Kahn Process Networks", SRC Techcon Extended Abstract, October 2005, Portland, OR

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***



*Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. **"MiBench: A free, commercially representative embedded benchmark suite."**, Proc. 4th IEEE International Workshop on Workload Characterization, 2001, pp.3-14.

Performance vs SimpleScalar

	Base Metropolis	1. Optimized Metropolis	2. Optimized SystemC	Simplescalar ARM
StrongARM Base	1997	5798	65537	916666
Xscale Base	2019	6086	67031	1013591
StrongARM Mem	1501	5865	60443	809229
Xscale Mem	1479	6036	60207	891007

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

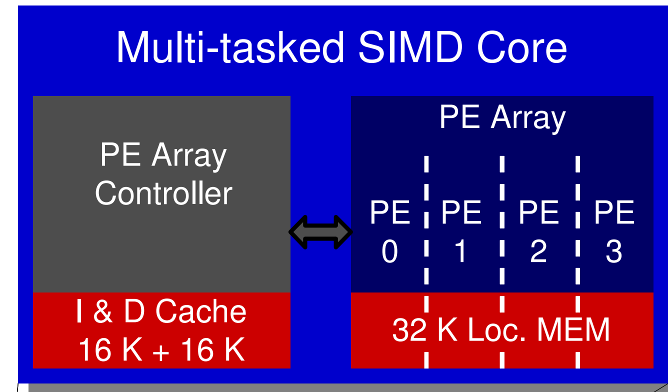
Outline

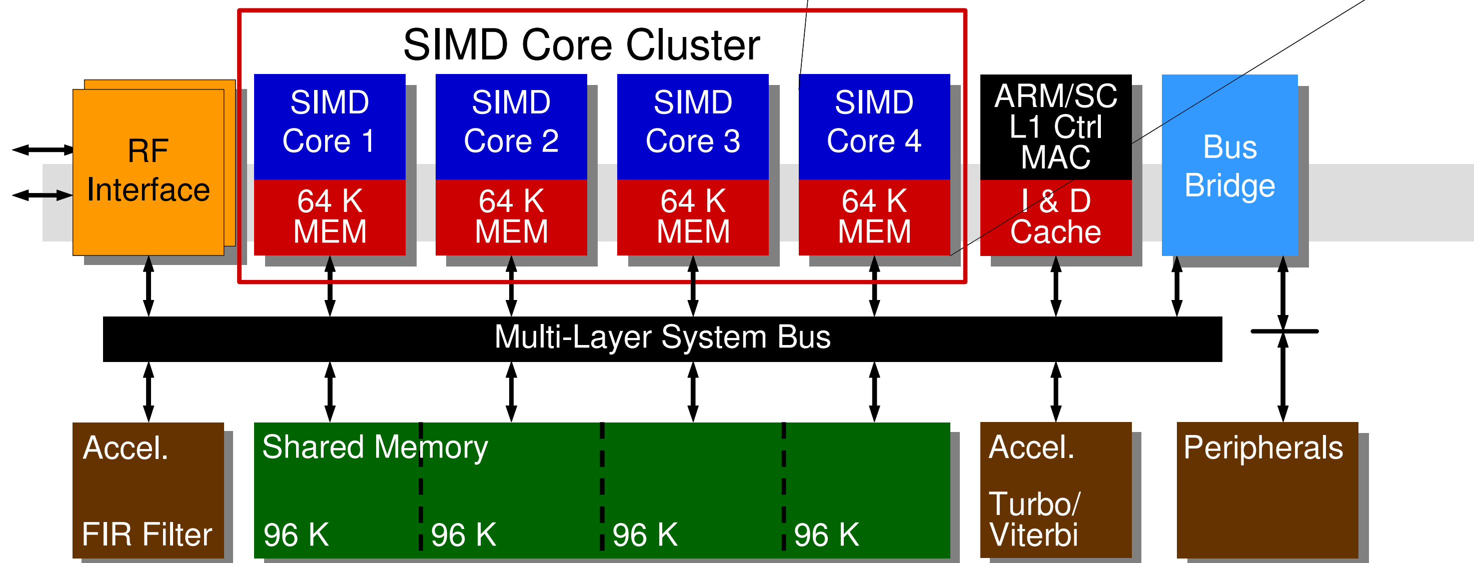
- Introduction
- Uniprocessor Modeling
- Multiprocessor Modeling
  - MuSIC Multiprocessor
  - Architecture Model
  - Functional Model
  - Mapping
- Timing Annotation
- Conclusions and Future Work

Baseband Processing Platform (MuSIC)

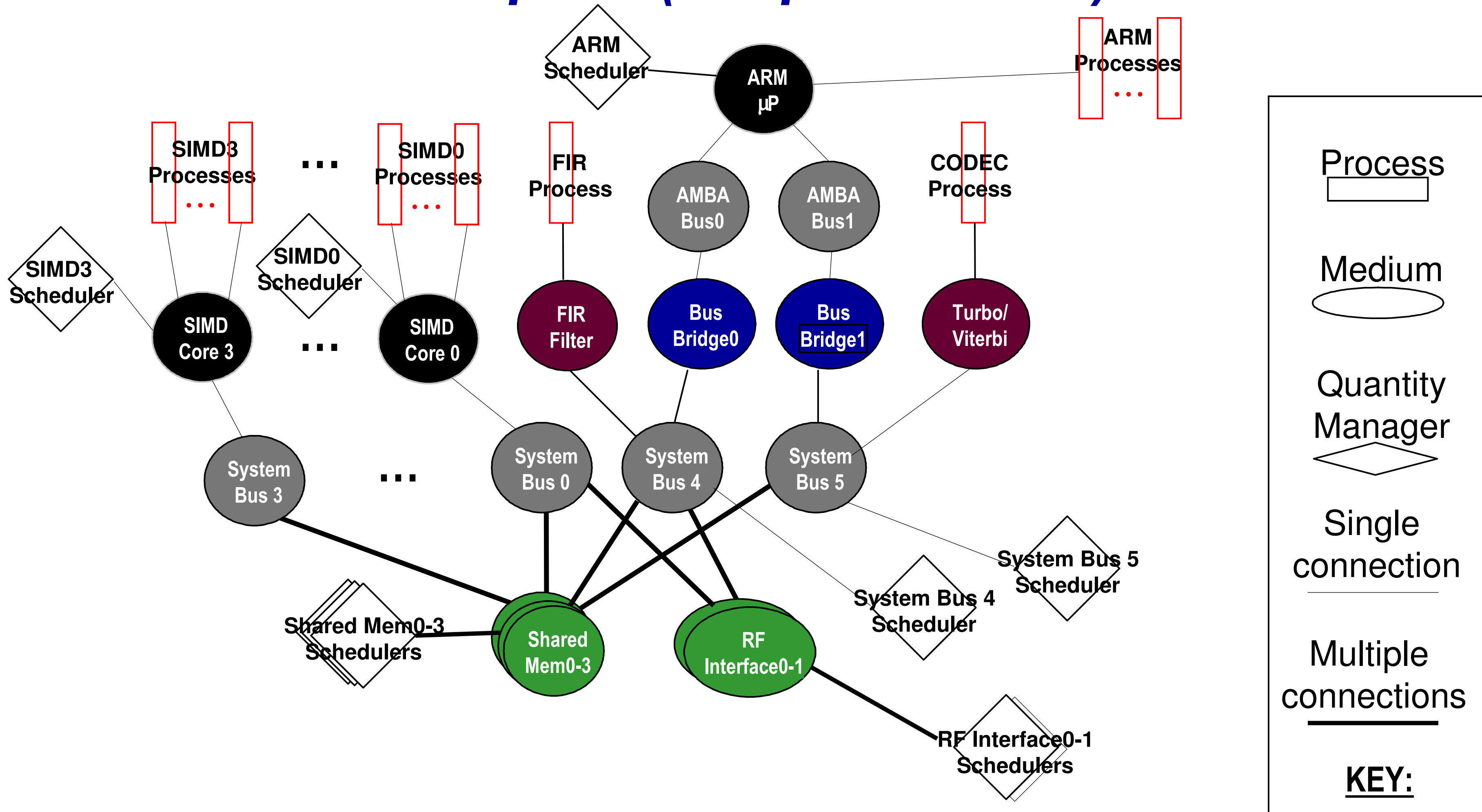
- Each SIMD Core has 4 Control Processors
- MuSIC has a total of 19 Control Processors
- Control processors run a Multiprocessor OS
- Each Control Processor can run 1 thread at a time (i.e. Processor ~= Thread)

Multiple SIMD Cores (MuSIC)





Baseband Processing Platform Architecture Modeled in Metropolis (Simplified View)



•Not Shown: Global Time, State Media, Sync Bus

Functionality: 802.11 Receive Payload Processing

Coder0 Spreader0 Modulator0

MAC

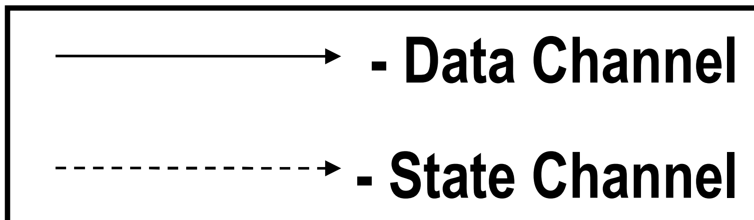
Rx_Splitter

Processing chain0

Processing chain1

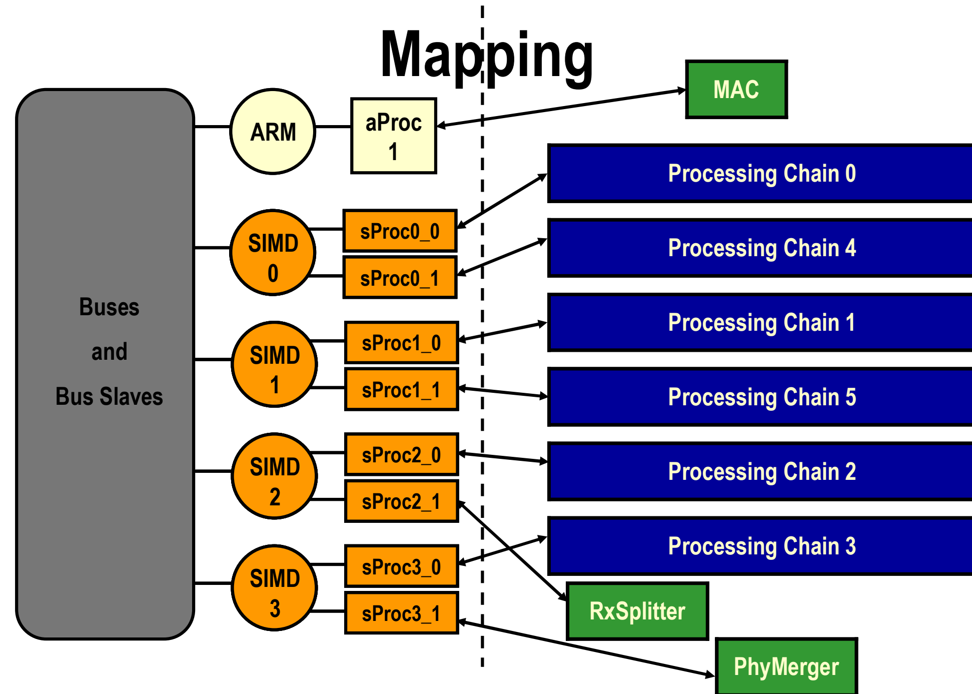
Processing chain5

PHY Merger



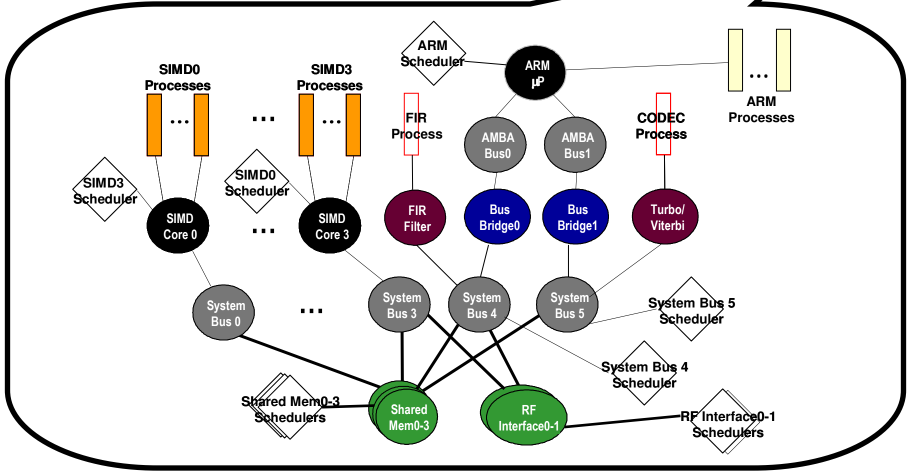
Mapping Functionality onto Architecture

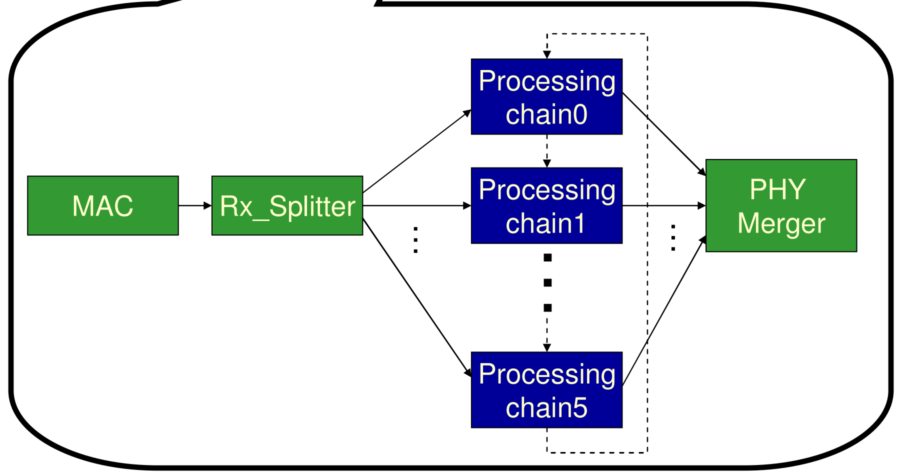
- Mapping done via Synchronization



Architecture

Function



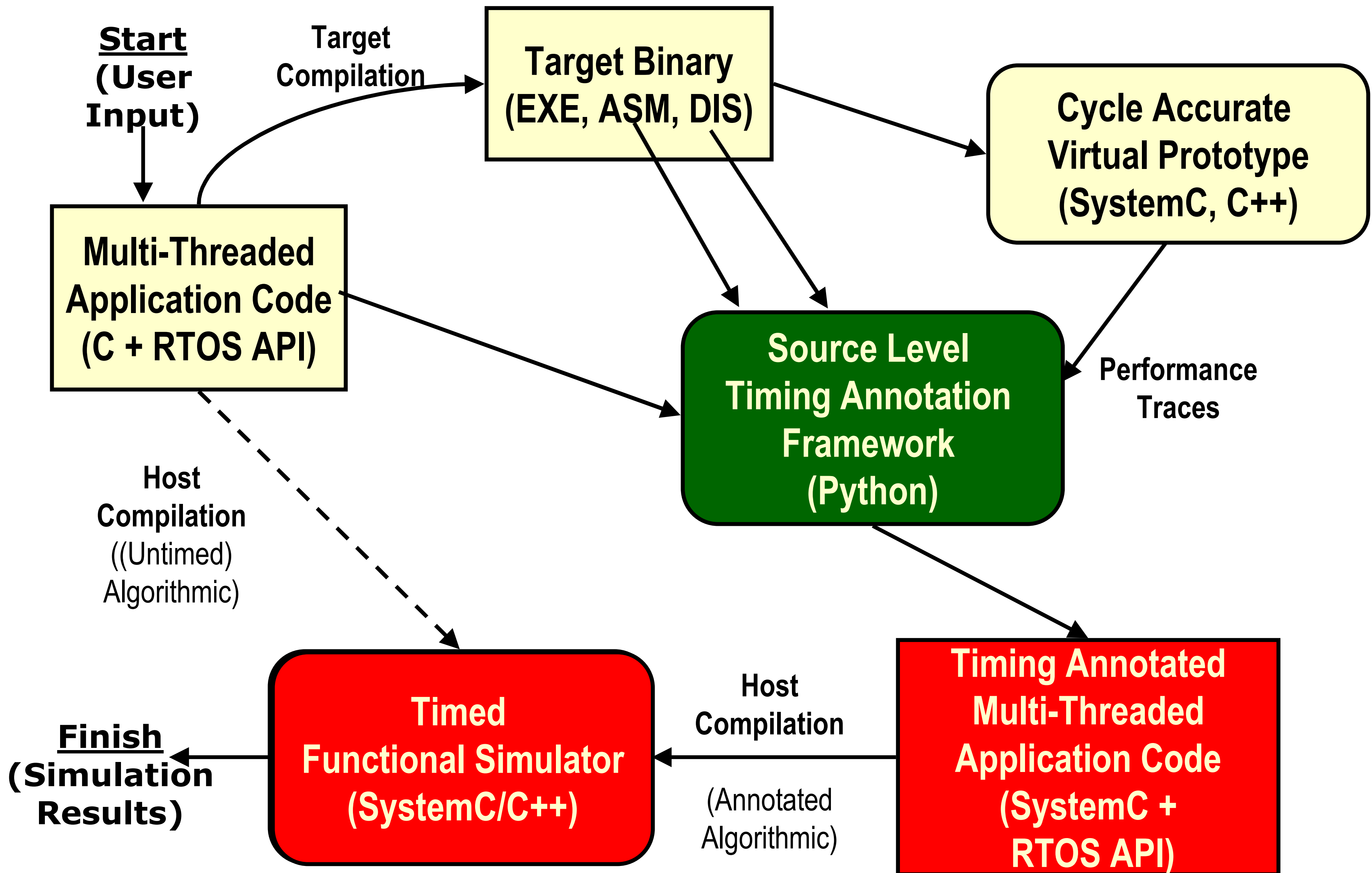


Outline

T. Meyerowitz, M. Sauermann, D. Langen, A. Sangiovanni-Vincentelli, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor", To appear in DATE 2008

- 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



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**

Annotation Example: Original Source Files

```
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
   …
```

Sample Code (fft64_test.c) Excerpt

```
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
  …
```

Annotated Code Excerpt (annotated_code/fft54_test.c)

Annotation Example: Assembly and Disassembly

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
            nop
            nop
            mov r12, r2
; **line 24
            pgen2 r2, 7
            mov r3, 4
            mov r4, 2
            …
```

Disassembly File

```
[00020200] <.Fmain>:           2f3c        sub  r15, 28
[00020202]:           e6de        push  r8..r14
[00020204]:           f7ff 1f34   add  r15, r15, -0x00cc

[00020208]:           0287        pgen2  r2, 7
[0002020a]:           f534        mov  r3, 4
[0002020c]:           f542        mov  r4, 2
[0002020e]:           ecc1 0b78   jl  .FAlloc_Mem:0x0216f0
[00020212]:           0000        nop
[00020214]:           0000        nop
[00020216]:           0000        nop
[00020218]:           0000        nop
[0002021a]:           0000        nop
[0002021c]:           0000        nop
[0002021e]:           fac2        mov  r12, r2

[00020220]:           0287        pgen2  r2, 7
[00020222]:           f534        mov  r3, 4
[00020224]:           f542        mov  r4, 2
…
```

Annotation Example: Processor Execution Trace

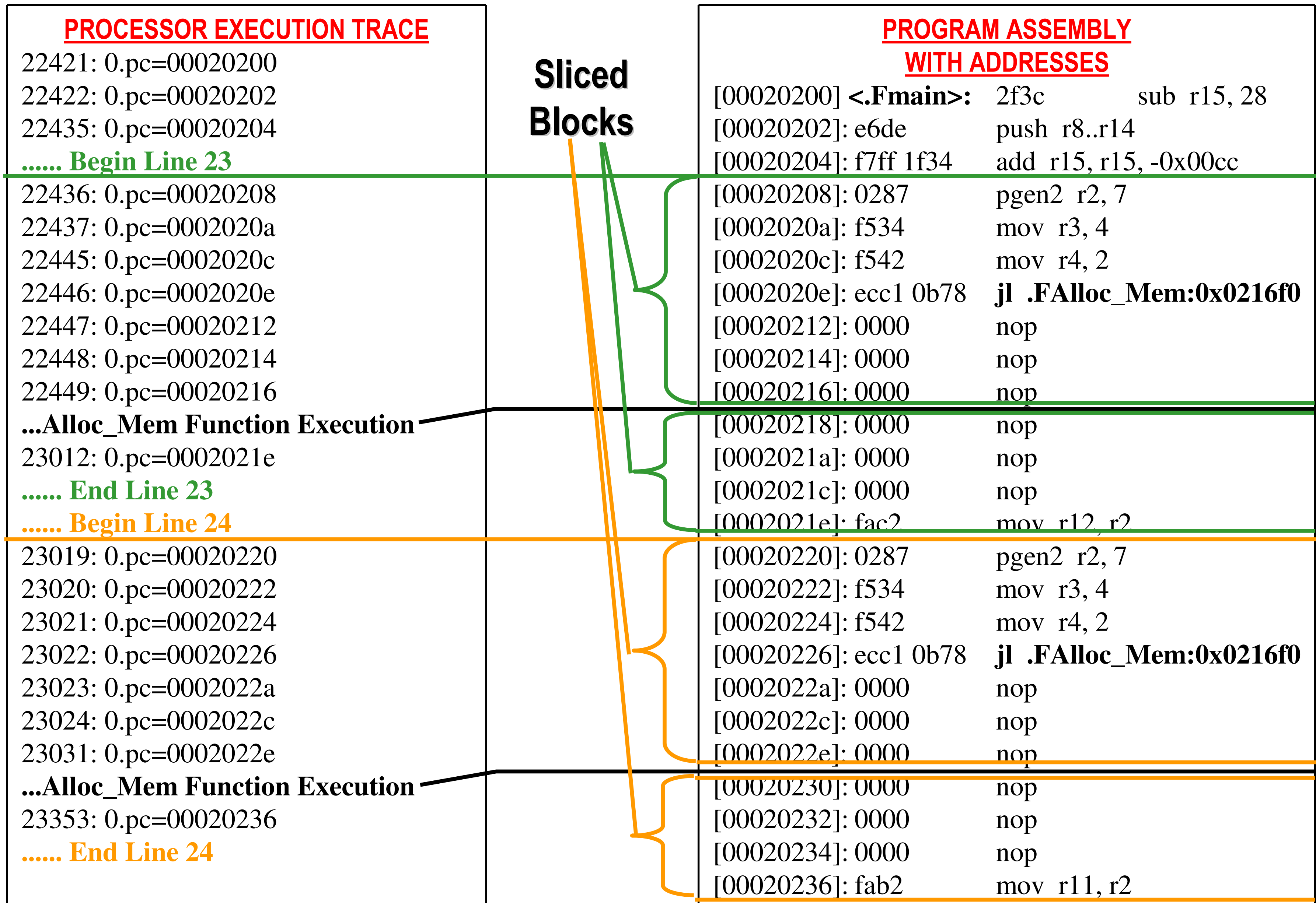
PROCESSOR EXECUTION TRACE

```
22421: 0.pc=00020200
22422: 0.pc=00020202
22435: 0.pc=00020204
...... Begin Line 23
22436: 0.pc=00020208
22437: 0.pc=0002020a
22445: 0.pc=0002020c
22446: 0.pc=0002020e
22447: 0.pc=00020212
22448: 0.pc=00020214
22449: 0.pc=00020216
...Alloc_Mem Function Execution
23012: 0.pc=0002021e
...... End Line 23
...... Begin Line 24
23019: 0.pc=00020220
23020: 0.pc=00020222
23021: 0.pc=00020224
23022: 0.pc=00020226
23023: 0.pc=0002022a
23024: 0.pc=0002022c
23031: 0.pc=0002022e
...Alloc_Mem Function Execution
23353: 0.pc=00020236
...... End Line 24
```

"22421: 0.pc=0020200" means:

Processor 0 fetches instruction 0x0020200 at cycle 22421

Annotation Example: Block Slicing at Jumps



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

```
Delay_Thread(D_statement1);
<statement1>
Delay_Thread(D_statement2);
<statement2>
...
```

Normal Case

```
Delay_Thread(D_test);
while (<test>) {
    Delay_Thread(D_test);
    Delay_Thread(D_body);
   <body>
}
```

While Loop Case

```
do {
    Delay_Thread(D_body);
    <body>
    Delay_Thread(D_test);
} while (<test>) ;
```

Do-While Loop Case

(Normal Case)

```
Delay_Thread(D_init + D_test);
for( <init>; <test>; <update>) {
    Delay_Thread(D_body);
    <body>
    Delay_Thread(D_update + D_test);
};
```

For Loop Case

Multiprocessor Annotation Algorithm

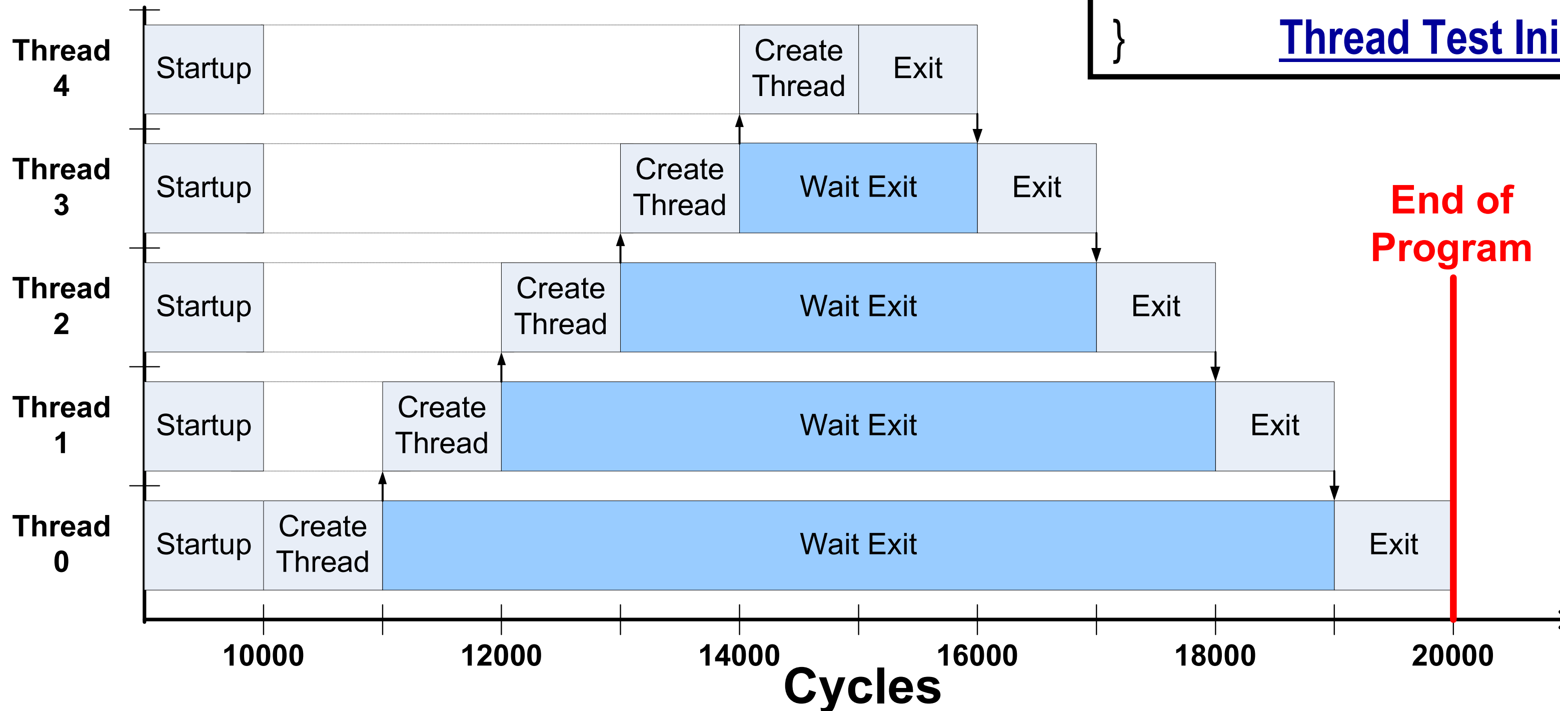
1. **Calculate each processor's block and line-level annotations as before except:**
   - **Special care with startup delays**
   - **Special handling of inter-processor functions**
     - **Ignore their delays**
     - **Substitute in characterized delays**
2. **Unify line-level annotations from all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



Multiprocessor Annotation: Handling Startup Delays

```
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
   }
   ...
}
```

Initial Code

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

```
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
   }
   ...
}
```

Fixed Annotated Initial Code

Multiprocessor Annotation: Why Characterization? (1 of 2)

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

Initial Code

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();
}
```

Direct Measurement Annotated Code

Results:
Delay = 14000
Error = 40%

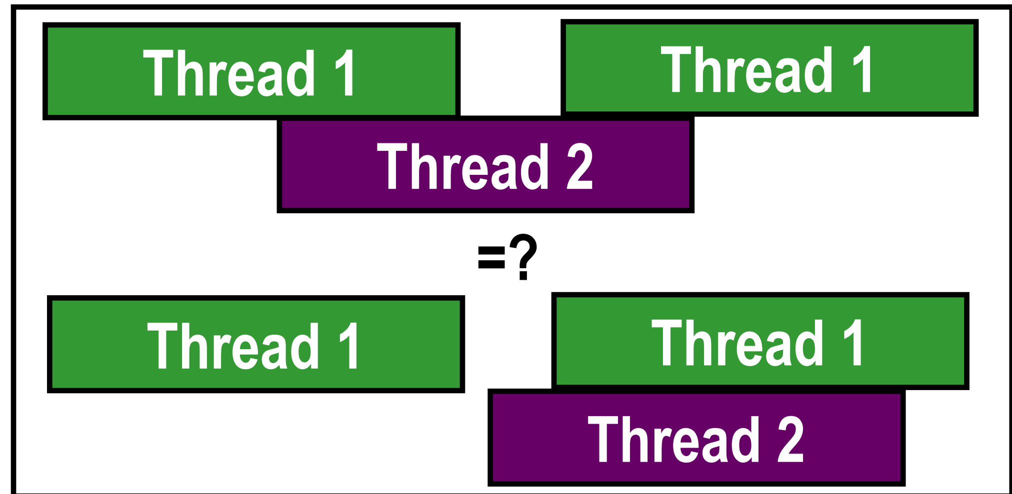
```
void main(void) {
   thread* next_thread;
   …
   next_thread = Create_Thread(main);

   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



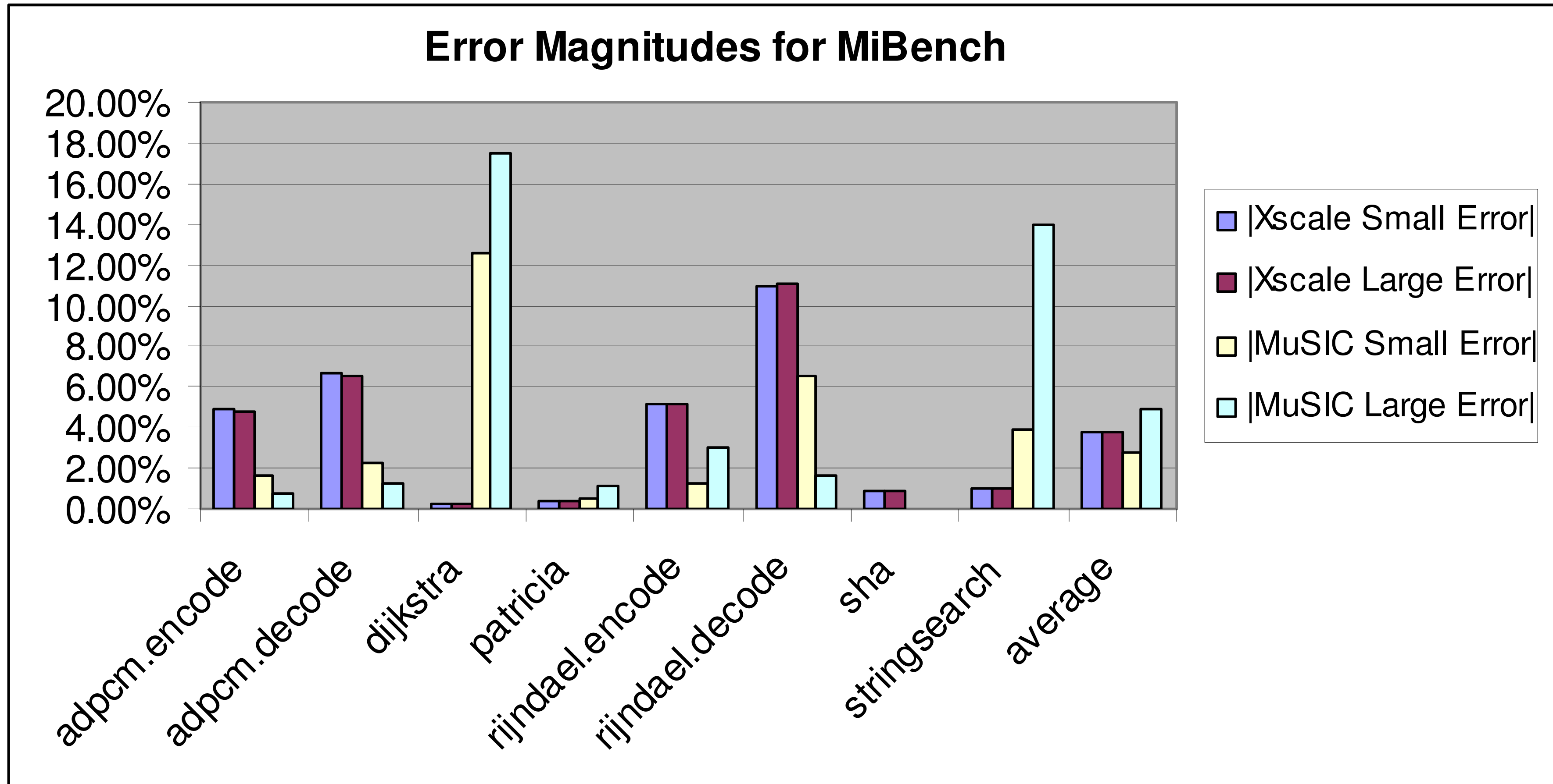
- Processor Execution Trace is approximate
  - Only gives fetch times
  - No pipelining
  - This means that intrinsic instructions can magnify errors

```
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();
}
```

Direct Measurement Annotated Code (v2)

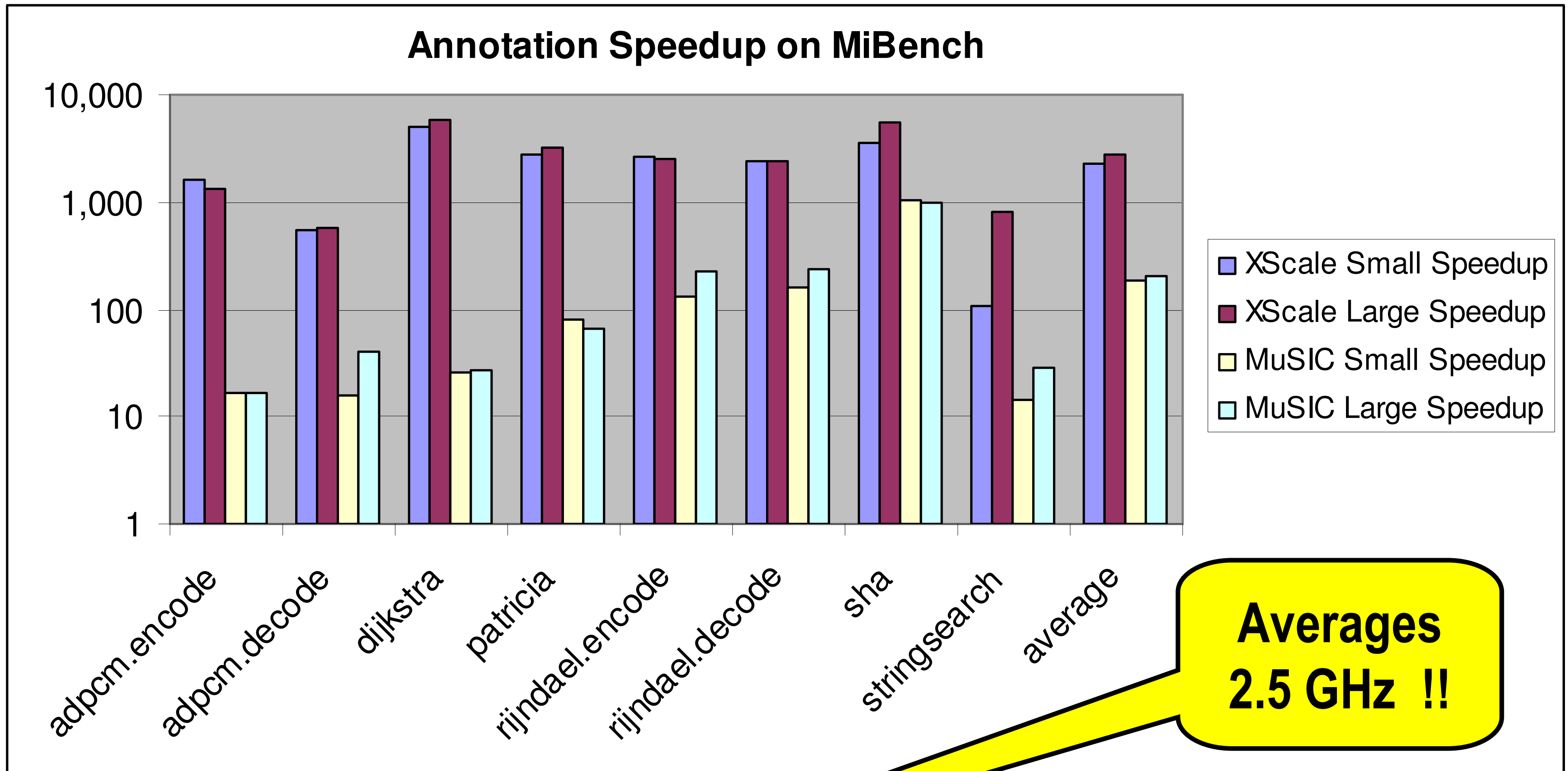
Results:
Delay = 34000
Error = 240%

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 (vs. VP)			Annotation Slowdown (vs. Native)		
	Average	Minimum	Maximum	Average	Minimum	Maximum
XScale	2250	110	5900	2	1	8
MuSIC	190	16	1030	290	1900	6

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

Test	Thread Count	Direct Meas. Error %	Char. Based Error %	Char. Based. Speedup
message test	3	0.00%	0.21%	80.1
streaming test	4	48.77%	0.18%	207.5
thread test	19	767.24%	-2.24%	526.9
avg. magnitude		*272.01%*	*0.87%*	*271.5*
max. magnitude		*767.24%*	*2.24%*	*526.9*

Internal Multiprocessor Results

Num. of Rows and Columns	Error	Speedup
32	-8.18%	14.00
64	-7.56%	18.49
160	-7.39%	17.03
avg. magnitude	7.71%	16.51
max magnitude	8.18%	18.49

JPEG results for different image sizes

Scaling of Annotations

Streaming Test for Different #'s of Worker Threads

Normalized Value

3.000
2.500
2.000
1.500
1.000

1
6

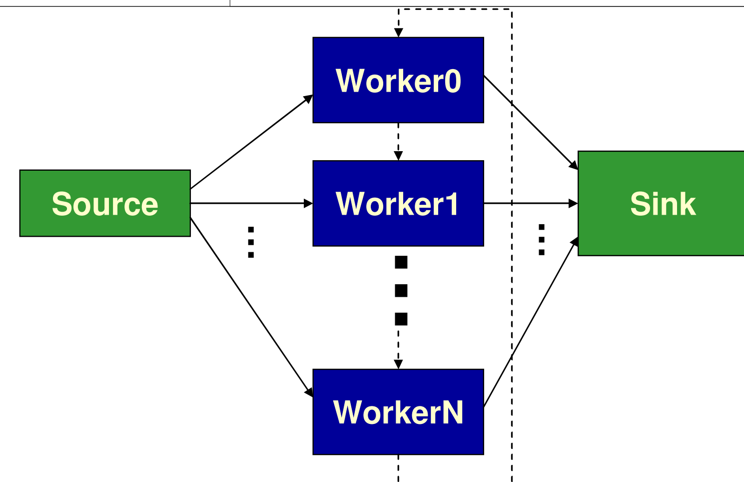
Number of Worker Threads

Normalized VP Execution Time

Normalized Char. Based Execution Time

Normalized Annotation Time

- **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**



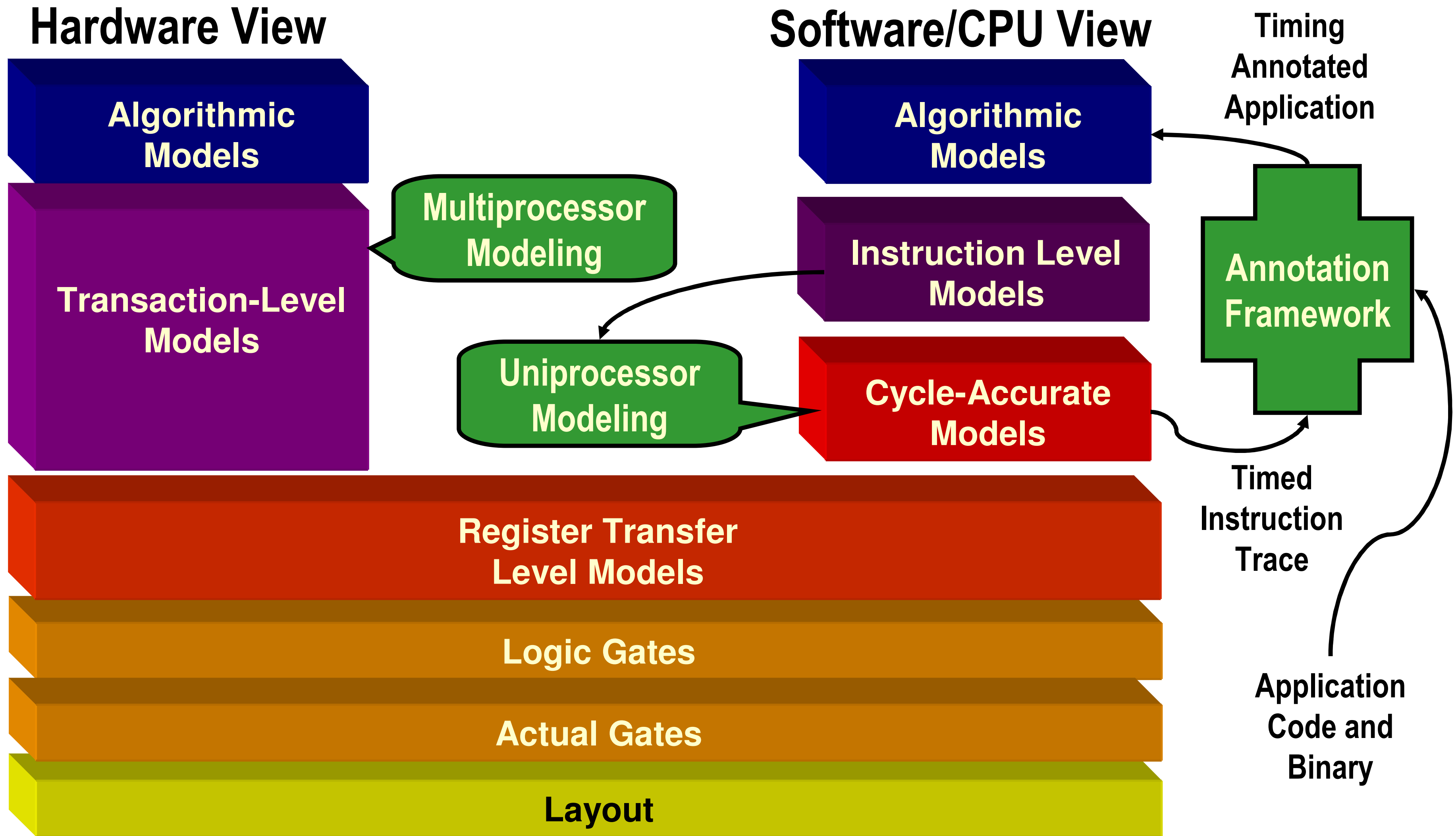
Timing Annotation Related Work

- 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, LISAtek, etc)
  - Fast and Accurate
  - Time consuming to create and modify
  - Multi-processors can still be slow
- Profiling
  - General Profiling (e.g. Gprof)
    - Insturmentation + 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

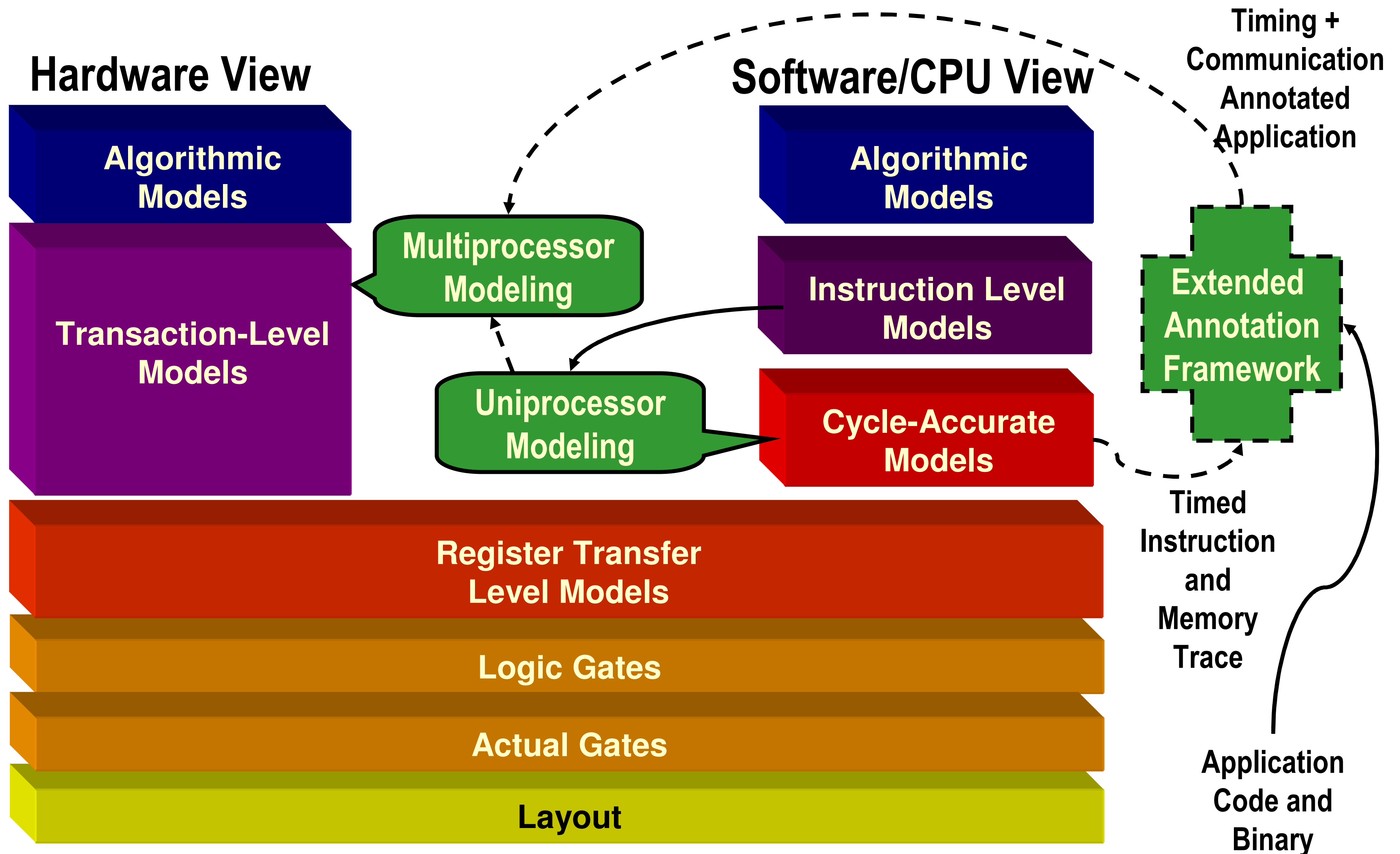
Contributions

- **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



Future Work in Terms of Hardware and Software Views



Acknowledgements

- Microprocessor Modeling
  - Intel Corporation for SRC sponsorship
    - Summer Internship
    - Mentors: Timothy Kam and Mike Kishinevsky
    - Berkeley Liason: 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
  - Infineon SDR Group: Wolfgang Raab, Dominik Langen, Cyprian Grassman, Mirko Sauermann, Matthias Richter, Alfonso Troya, Jens Harnisch, et al.
  - 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
  - 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

Annotation Example: Annotation Results

```
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

Level	Speed / Accuracy	Function / Architecture
Algorithmic	Excellent / None	Concurrent / None
Annotated Algorithmic	Excellent / Good	Concurrent + Delays / None
Abstract (TLM) Model	Good / Poor	Concurrent / Timed Resources
Annotated Abstract Model	Good / Very Good	Concurrent + Delays / Timed Resources
Cycle-Level (Virtual Prototype)	OK / Cycle Level	ASM / SystemC
RTL-level	Horrible / Signal Level	ASM / SystemC

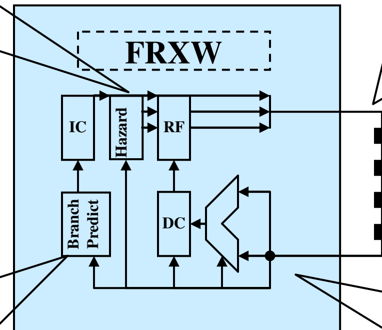
Our Focus

Annotations

Single Process Model*

•Add hazard detection and bubble insertion (stalls)

•Parameterize the pipeline depth



•Add a branch predictor

•Pass prediction and PC down pipeline

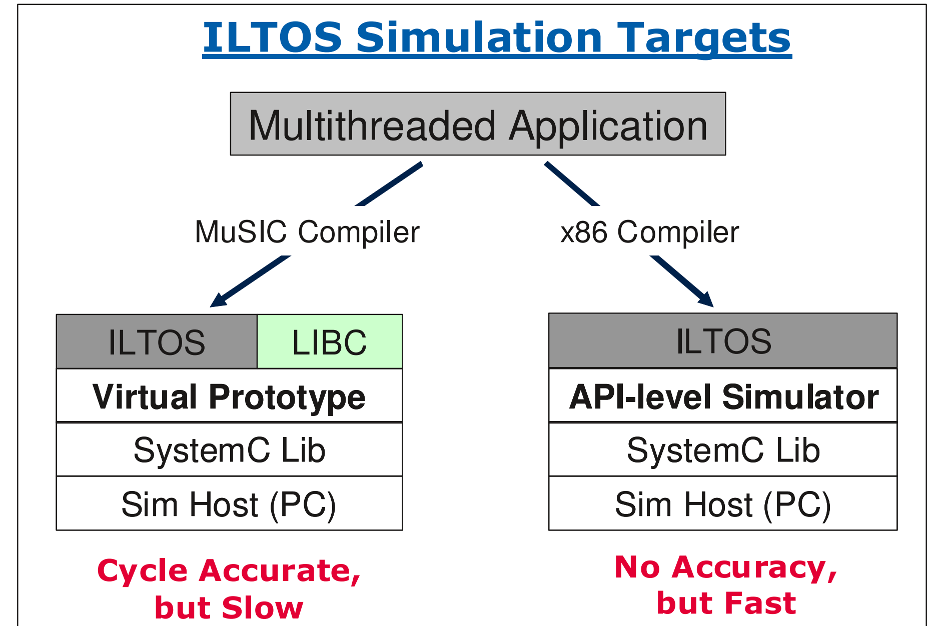
•Resolve branch when it commits

- **Single Process Execution Order**
  1. Read operands
  2. Execute
  3. Write to register file
- **Synchronous Assumption**

* Collaboration With: Sam Williams

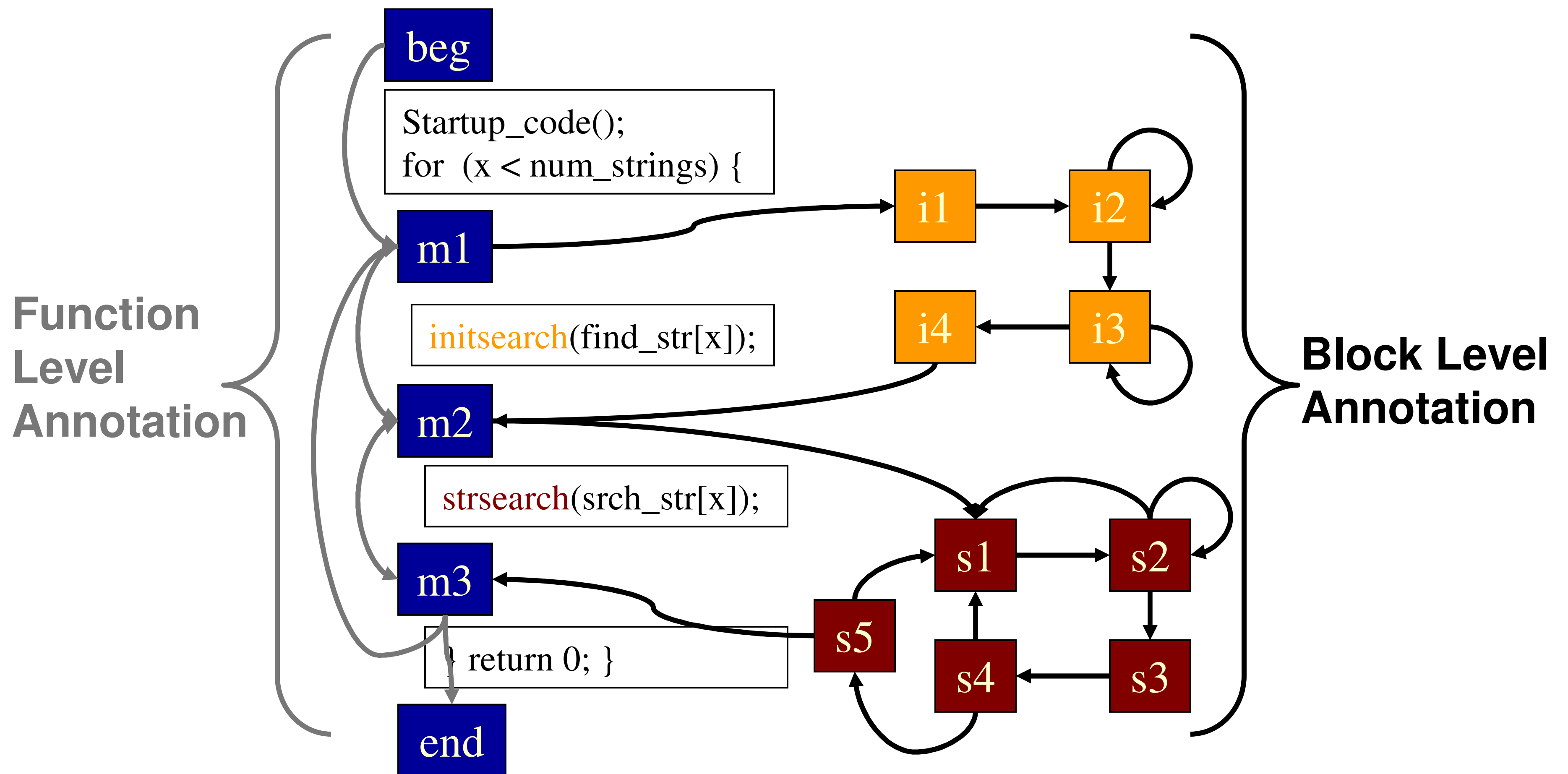
MuSIC Platform Software

- 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



* Collaboration With: Mirko Sauermann @ Infineon

Performance Backwards Annotation: An Example



Characterization Example Actual Execution

Thread Id	Create Thread Begin	Wait Exit Begin	Wait Exit End	Exit End	Full Wait Delay
0	0	1000	9000	10000	8000
1	1000	2000	8000	9000	6000
2	2000	3000	7000	8000	4000
3	3000	4000	6000	7000	2000
4	4000	5000	5000	6000	0
Avg.					4000

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;
   …
   next_thread = Create_Thread(main);
   wait_exit(next_thread);
   exit();
}
```

Initial Code

Characterization Example Direct Measurement Execution

Thread Id	Create Thread Begin	Wait Exit Begin	Wait Exit End	Exit End
0	0	1000	13000	14000
1	1000	2000	12000	13000
2	2000	3000	11000	12000
3	3000	4000	10000	11000
4	4000	5000	9000	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();
}
```

Direct Measurement Annotated Code

Results:
Delay = 14000
Error = 40%

Characterization Example Characterization-Based Execution

Thread Id	Create Thread Begin	Wait Exit Begin	Wait Exit End	Exit End
0	0	1000	9000	10000
1	1000	2000	8000	9000
2	2000	3000	7000	8000
3	3000	4000	6000	7000
4	4000	5000	5000	6000

```
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

Thread Id	Create Thread Begin	Wait Exit Begin	Wait Exit End	Exit End
0	0	1000	28000	34000
1	1000	2000	22000	28000
2	2000	3000	16000	22000
3	3000	4000	10000	16000
4	4000	5000	5000	10000

```
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();
}
```

Direct Measurement Annotated Code (v2)

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%)

Benchmark	Direct Meas. Error %	Char. Based Error %	Char. Based Speedup
alloc_test	0.00%	-0.59%	33
dhrystone	-0.09%	-0.21%	47
libc_test	-0.46%	-0.46%	94
payload1	-0.02%	4.41%	33
ratematch	-0.04%	-0.04%	11
syncmng_test	0.00%	1.59%	13
udt_test	-0.02%	-0.02%	25
fft64	0.00%	-0.03%	48
cck_test	-0.15%	-0.15%	139
udt_test2	0.00%	0.00%	23
avg. magnitude	***0.08%***	***0.75%***	*47*
max. magnitude	***0.46%***	***4.41%***	*139*

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)

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

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!