Metropolis ARM CPU Examples

Trevor Meyerowitz

September 14, 2004

Technical Memorandum No. UCB/ERL M04/39

Chapter One

Introduction

This document explains the microarchitectural models of the Strongarm [2] and XScale [3] ARM Instruction Set [6] processors described in the Metropolis Meta Model (MMM) language [8, 1], and included in the examples directory of the Metropolis 1.0 release. These are trace-based models built using the YAPI [4] model of computation, and are base their delays and branch prediction schemes listed in the processor manuals. The models do not simulate the memory system, the bus, and interrupts. These are good non-trivial examples in Metropolis that illustrate the use of the YAPI platform, templates, and interfacing code generated by the Metropolis SystemC Simulator backend to native code. For a broader methodological view please read the design guidelines[7], and for a general reference see the metamodel document[8].

1.1 Directories

The processor models, support code, and trace-generating simulator can be found in the *metro/examples/yapi_cpus/arm* directory. Below are brief descriptions of the directory contents.

README.txt - A text file providing a brief overview and installation instructions.

arm_sim - The simulator used for generating traces.

object_files - A few small traces used to drive the models.

strongarm_yapi - The Strongarm model directory.

xscale_yapi - The XScale model directory.

1.2 Background Information

YAPI

YAPI stands for the Y-Chart API. It is an implementation of Kahn Process Networks (KPN) [5] with the addition of a non-deterministic select. Our models only use the KPN semantics, which amount to communication through unbounded FIFO's with blocking reads and non-blocking writes. Furthermore, we use the FIFO's to model pipeline delays by pre-loading them to the wanted length, and then maintaining the pipeline depth by reading-from and writing-to every FIFO each cycle.

XScale and Strongarm Processors

The XScale and Strongarm Microprocessors are both low-power embedded processors made by Intel that implement the ARM Instruction Set Architecture (ISA). The Strongarm processor has a five stage pipeline with static branch prediction, and has speed of up to 206 MHz. The XScale PXA-25x processor is the successor to the Strongarm, has a seven stage pipeline, dynamic branch predication, and has speeds up to 400 MHz.

Chapter Two

Processor Models

This chapter provides an overview of the different pieces used to construct the microarchitectural models. The first section provides a high level picture of the models and their behavior. The second section explains the trace format used by the models. The final section explains how the high-level model is customized for the Strongarm and XScale models. The next chapter provides a more detailed view of the code used in these models.

2.1 High Level Overview

A microarchitectural model executes an instruction trace, and returns the number of cycles that it takes to execute. To ensure accuracy the model must account for the delays of individual instructions, the interaction between them, and the impact of limited resources. We use YAPI in a cyclical manner, where each cycle every process reads one token from all of its input channels and writes a token to each of its output channels. In order to model a particular pipeline length, channels are pre-filled with the number of tokens equal to the pipeline length. As long as the cyclic assumption is maintained the pipeline behavior is guaranteed.

We use a two process model like the one pictured in figure 2.1. The two processes are a fetch process that handles the fetch and issue of instructions from execution trace, and an execution process that handles the execution, operand dependencies, and forwarding delays between

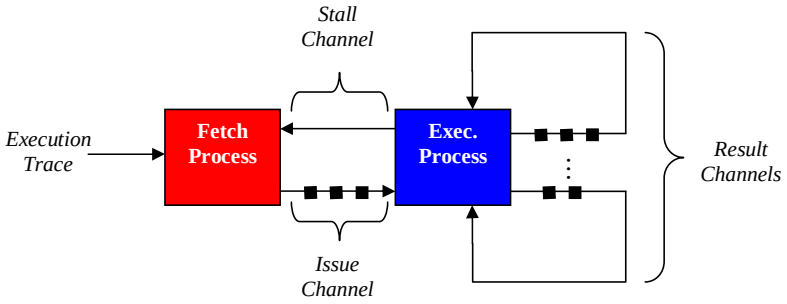

Figure 2.1: Model Overview

instructions. We use three types of YAPI channels: an *issue-channel* that passes instructions from the fetch process to the execute process, a *stall-channel* from the execute process to the fetch process, and one or more *result-channels* that model the execution delays of instructions by connecting the execution process to itself.

Each instruction in the execution trace has an associated *type* as well as *read_operands* and *write_operands*. Operands can be either registers or the condition codes. Each instruction type is classified by two delays, *issue_delay* and *results_delay*. The *issue_delay* of an instruction, represents the number of cycles that it takes for the fetch unit to write the instruction to the $issue - channel$ assuming that there is a cache hit. The *results_delay* of an instruction type is the number of cycles that the next instruction will have to wait if it depends on one or more write-operand of an instruction of that type.

Fetch Process

The fetch process begins execution by pre-loading the $issue - channel$ to model the pipeline between the two processes. This delay typically represents the delay of instruction fetch and decode.

For the main execution loop, which executes every cycle, it reads from the $stall - channel$, and if there is no stall then it writes either the fetched

instruction from the instruction trace or, if there is an issue stall, an bubble instruction to the issue-channel.

Execute Process

The Execute process's execution begins by pre-loading the $result-channels$ to their configured lengths, then the main loop is entered.

In the main loop, if there is no stall from operand dependencies, then the next instruction is read from the issue channel. After this the process reads from the result channels, the process checks to see if any of the operands of the current instruction are unavailable and if they are it sets stall to true. If there is no stall, then the appropriate result channel is selected for the instruction and it is written to it. After this, bubble instructions are written to all of the unselected result channels, and the calculated stall value is written to the stall channel. A stall happens in the case of operand or condition code dependencies.

2.2 Trace Format

The microarchitectural models are all driven by instruction traces generated by the functional Instruction Set Simulator (ISS). Each instruction issued by the ISS is turned into a trace entry that consists of three elements: a string that indicates if the instruction executes (i.e. "EX") or doesn't (i.e. "NOEX"), a hexadecimal value indicating the PC (Program Counter) address of the instruction, and the hexadecimal value of the instruction. Figure 2.2 shows the first 4 entries of the present the trace file `metro/examples/yapi_cpus/arm/object_files/fib.tr` in the first column, and the decoded entries in the second column.

Trace File	**Decoded Instructions**
NONOEX fffffff8 00000000	andeq r0, r0, r0
NOEX fffffffc 00000000	andeq r0, r0, r0
EX 00000000 ea000011	b 11
EX 0000004c e3a0d702	mov sp, #524288 ; 0x80000

Figure 2.2: Sample Trace File and its Decoding

2.3 Model Customization

Both the Strongarm and XScale models are derived from the same base models. They each have their own customized Fetch and Execute processes that extend the Fetch and Exec. Furthermore, they have their own netlists, and trace_interface C++ files that indicate the issue-stall values.

Chapter Three

Code Documentation

In this chapter we explain different pieces of the code in more detail than in the previous chapter, and then provide tips for interfacing Metropolis Models using the SystemC simulator with native code.

3.1 Instruction Class Specification

In this section we show some of the code used to represent the instruction data type, it comes from the file `MyDataTypes.mmm`. A segment of this file is shown in figure 3.1

The first line of variables has integers for the: instruction word, the instruction type, the instructions program counter, and the instruction number (i.e. number that it is in the trace). The second and third lines have variables for storing the read and write operands. The final variables relate to the number of issue cycles, whether or not the instruction issues, how it accesses condition codes, and data in its branches.

The default constructor produces non-executing "bubble" instructions with no operands. The second type takes in the instruction number, the instruction word, the program counter, and a boolean indicating execution. It calls a the $setValues$ method, which will call the trace interface code and determine the instruction type and set all of the appropriate values.

```
public class Instruction extends Object {
  public static final int REG_SIZE = 32;

  // Instruction specific variables
  public int IW, inst_type, PC, inst_num;
  public int num_op_regs, num_wr_regs;
  public int op_regs[], wr_regs[];
  public int extra_issue_cycles;
  public boolean executes, reads_cc, writes_cc;
  public boolean has_prediction, predict_taken, in_btb;
  private boolean is_branch;

  // empty (bubble) instruction constructor
  public Instruction() {...}

  // constructor for non-empty/non-bubble instruction
  public Instruction(int _inst_num, int _PC,
                     int _IW, boolean _exec) {
    super();
    op_regs = new int[REG_SIZE];
    wr_regs = new int[REG_SIZE];
    setValues(_inst_num, _PC, _IW, _exec);
  }
  ...
}
```

Figure 3.1: Instruction Class Code

3.2 Fetch Code

Figure 3.2 shows a portion of the fetch class declaration in the meta-model.

Ports

This model has 3 ports. $FetchedInstruction$ is an output port that passes the instruction created from the instruction trace on to the Exec process. $DoStall$ is an input port that reads stall information from the Exec process.

```
process Fetch extends
        yapiprocess-<Instruction;Instruction;
                     Instruction;Instruction>- {
  // PORTS
  ////////////////////////////////////
  port yapioutinterface-<Instruction>- FetchedInstruction;
  port yapiininterface-<yapiint>- DoStall;
  port yapioutinterface-<yapiint>- DoConfig;

  // PARAMETERS
  ////////////////////////////////////////
  parameter int ExecutionPipelineDepth;
  parameter boolean DO_PREDICTION; // do branch prediction
  parameter boolean REAL_BTB; // use real (or simulated) BTB
  ...
  Fetch(String n, int PD, boolean predict,
        boolean real_btb) {...}
  ...
}
```

Figure 3.2: Fetch Declaration with Ports and Parameters

$Config$ is an output port that passes the number instructions in the trace onto the Exec process.

Parameters

The model can be parameterized in terms of the pipeline depth between it and the Exec model (i.e. the length of the $FetchedInstruction$ channel), and in terms of branch prediction.

Pseudo-Code

Figure 3.3 shows the pseudo-code for the main execution of the Fetch process.

```
FetchExecCode() {
  preloadIssueChannel();

  // main loop
  while(true) {
     op_stall = Stall.read();
     if (issue_stall > 0) issue_stall--;
     if (!op_stall) {
        if (no-inst) {
           inst = trace.getNext();
           issue_stall = inst.issueStall();
        }
        if (issue_stall==0)
           issue_channel.write(inst);
        else
           issue_channel.write();
     }
  }
}
```

Figure 3.3: Fetch Pseudo-Code

3.3 Execute Code

Ports

This model has 3 ports connected to the Fetch process and two arrays of ports. $FetchedInstruction$ is an input port that receives instructions from the Fetch process. $DoStall$ is an output port used to pass stall information onto the Fetch process. $Config$ is an input port that receives the number instructions in the trace from the Fetch process. $ResultsIn$ and $ResultsOut$ are arrays of input and output YAPI ports that are used to represent execution delays within the model. Note that each i-th element of $ResultsIn$ and $ResultsOut$ connects to the same yapichannel and is connected at each end to the Exec process.

```
process Exec extends
        yapiprocess-<Instruction;Instruction;
                     Instruction;Instruction>- {
  // PORTS:
  ////////////////////////////
  // The instruction sent from the Fetch process
  port yapiininterface-<Instruction>- FetchedInstruction;
  port yapiininterface-<yapiint>- DoConfig;
  port yapiininterface-<Instruction>-[] ResultsIn;
  port yapioutinterface-<Instruction>-[] ResultsOut;
  port yapioutinterface-<yapiint>- DoStall;

  // PAREMETERS
  parameter int MispredictPenalty;
  parameter int num_commit_types;
  parameter int commit_lengths[];

  Exec(String n, int MP, int commit_types,
       int _commit_lengths[]) {...}
  ...
}
```

Figure 3.4: Execute Declaration with Ports and Parameters

Parameters

The Exec process is configured through an array of integers that specifies the lengths of the different commit types. It also is configured via a branch-misprediction penalty value.

Execution Pseudo-Code

Figure 3.3 shows the pseudo-code for the main execution of the Exec process.

```
ExecuteExecCode() {
  preloadResultChannels();

  // main loop
  while(true) {
     if (!stall) inst = issue_channel.read();

     ResultChannels.ReadAndUpdate();
     stall = ReadOperandsAndCheckStall(inst);

     if (!stall)
        ResulChannels.selectAndWrite(inst);

     ResultChannels.writeBubbles();
     stallChannel.write(stall);
  }
}
```

Figure 3.5: Execute Pseudo-Code

3.4 Interfacing with Native Code

While the microarchitectural models are primarily specified in the MMM, they do rely on trace interfaces and memory management done in non-native code. We will discuss the interfacing using blackbox statements, as well as modifying the makefiles to accommodate non-MMM code when building a model in Metropolis using the SystemC Simulator. Finally, we present our use of blackbox statements to manage memory in the model.

Blackbox Statements

Our models uses blackbox statements to insert into the generated SystemC simulator code. These are done using the below syntax:

```
blackbox(SystemCSim)%%
  // inserted blackbox C++/SystemC code
%%
```

Makefile Modifications

In order to interface with external code, this code must be added into the main makefile to be compiled, and also into the final linking file. To modify the linking we replace the generated *systemc_sim.mk* makefile with one that has the external code linked into the final execution. In our case the modified files are: `mmscs_xscale.mk` and `mmscs_strongarm.mk`.

Memory Management Considerations

At the current time the metamodel doesn't support memory deallocation, so any dynamic allocations of memory should be deleted using blackbox statements. Below is an example of the deallocation a variable of type $Instruction$.

```
blackbox(SystemCSim)%%
  delete ResultsInstruction;
%%
```

Chapter Four

Installation and Use

4.1 Running the Microarchitectural Models

These instructions assume that you have properly installed and configured the Metropolis framework.

Compiling the Models

In the *strongarm_yapi* or the xscale_yapi directory type "make" to build the appropriate executable (i.e. *strongarm.x* or *xscale.x*). For a model that has more verbose output type "make debug" this will produce the executables *strongarmd.x* or *xscaled.x*.

Running the Models

To run the strongarm model type "./strongarm.x [trace_file]" where trace file is the extension of an execution trace file. to run the XScale model use xscale.x instead of strongarm.x.

For example in the directory *metro/examples/yapi_cpus/arm/xscale_yapi*: Type: "./xscale.x ../object_files/fib.tr" to run an instruction trace from a Fibonacci program.

Once run, the execution statistics for that trace on the microarchitural model are printed out. The trace "fib.tr" executes in 1195 cycles on the Strongarm model and 1419 cycles on the XScale Model.

4.2 Building and Simulating Traces

To create new traces first new executables must be created using a cross compiler. Intel has binaries and source of the GNU compiler tools targeted for the XScale at their PXA developer site. The simulator only supports execution of ELF binary files.

Building the Functional Simulator

To generate new traces the simulator is run with a particular executable file with the below format:

"./arm-sim [executable file] [trace file name]"

This will create a trace with the name "trace file name" that can be used to drive the microarchitectural models.

Chapter Five

Acknowledgements

5.1 CPU Modeling Acknowledgements

This work is funded by the SRC and Intel Corporation. It builds upon work done with Sam Williams, with suggestions and mentoring from Luciano Lavagno and Kees Vissers. Yoshi Watanabe and Mike Kishinevsky provided additional mentoring and feedback. The ARM instruction set emulator is based on a modified version of the gdb-ARMulator modified by Christian Sauer. The trace-interface files are based on pieces of the SWARM ARM7 emulator written by Michael Dales.

5.2 Metropolis Acknowledgements

This work was supported in part by the following corporations:

- Cadence
- General Motors
- Intel
- Semiconductor Research Corporation (SRC)
- Sony
- STMicroelectronics

- and the following research projects:
  - NSF Award Number CCR-0225610 and the Center for Hybrid and Embedded Systems (CHESS, http://chess.eecs.berkeley.edu)
  - The MARCO/DARPA Gigascale Systems Research Center (GSRC, http://www.gigascale.org)

The Metropolis project would also like to acknowledge the research contributions by:

- The Project for Advanced Research of Architecture and Design of Electronic Systems (PARADES, http://www.parades.rm.cnr.it/) (in particular Alberto Ferrari)
- Politecnico di Torino
- Carnegie Mellon University
- University of California, Los Angeles
- University of California, Riverside
- Politecnico di Milano
- University. of Rome
- La Sapienza
- University of L'Aquila
- University of Ancona
- Scuola di Sant'Anna and University of Pisa

src/metropolis/metamodel Portions of the Java code were derived from sources developed under the auspices of the Titanium project, under funding from the DARPA, DoE, and Army Research Office. The Java code was further developed as part of the Ptolemy project. The Java code is released under Metropolis copyright.

src/metropolis/metamodel/frontend/Lexer Portions of JLexer are: "Copyright (C) 1995, 1997 by Paul N. Hilfinger. All rights reserved. Portions of this code were derived from sources developed under the auspices of the Titanium project, under funding from the DARPA, DoE, and Army Research Office."

src/metropolis/metamodel/frontend/parser/ptbyacc ptbyacc is in the public domain.

Bibliography

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer Magazine*, pages 45–52, April 2003.

[2] Intel Corporation. *SA-110 Microprocessor Technical Reference Manual*. Santa Clara, CA, 2000.

[3] Intel Corporation. *Intel Xscale Microarchitecture User's Manual*. Santa Clara, CA, 2003.

[4] W. J. M. Smits P. vd Wolf J.-Y. Brunel W. M. Kruijtzer P. Lieverse K. A. Vissers E. A. de Kock, G. Essink. Yapi: Application modeling for signal processing systems. *Proceedings of Design Automation Conference*, pages 402–405, 2000.

[5] G. Kahn. The semantics of a simple language for parallel programming. *Proceedings of the IFIP Congress*, August 1974.

[6] ARM Ltd. *ARM Architecture Reference Manual*. Cambridge, England, 2000.

[7] Alessandro Pinto. Metropolis design guidelines. In *Technical Memorandum UCB/ERL M04/40, University of California, Berkeley, CA 94720*, September 14, 2004.

[8] The Metropolis Design Team. The metropolis meta model version 0.4. In *Technical Memorandum UCB/ERL M04/38, University of California, Berkeley, CA 94720*, September 14, 2004.