RC 21968 (98739) 20 February 2001 **Computer Science**

IBM Research Report

Turandot Users's Guide (Microarchitecture Exploration Toolset)

Jaime H. Moreno, Mayan Moudgill

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Turandot's User's Guide
The Microarchitecture Exploration Toolset (The MET)

Jaime H. Moreno, Mayan Moudgill
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
<jhmoreno@us.ibm.com, mayan@watson.ibm.com>

Abstract

Turandot *is a Power-PC based highly parameterized superscalar processor model suitable for exploring limits and potentials of alternative features in microprocessors. The model supports trace-driven simulation with static and dynamically generated traces, is capable of taking into account the effects of instructions executed speculatively, and allows exploring many features such as instruction-level parallelism, issuing policy, issue width, number and size of the resources, cache sizes, and many more. The model is characterized by its speed (in excess of 100 million simulated processor cycles per hour), and its flexibility within the domain of superscalar processor organizations considered.*

This User's Guide provides basic information on models built using Turandot which are already available, how to execute these models, how to interpret the results generated by the models, and how to build new models with different microarchitecture features.

Table of Contents

1. Introduction

Contemporary microprocessors are classified as superscalar engines because of their ability to process more than one instruction at a time. Many such processors issue instructions out-of-order, support speculative execution, reorder load and store operations, rely on branch prediction, and so on. Such processor features make processor microarchitectures increasingly complex, and these are only some options from a large design space. The associated design complexity raises questions regarding the potential performance benefits obtained from the features. Since the interaction among microarchitecture features is often counterintuitive, early, accurate, and timely modeling are required to ensure proper design trade-offs. In other words, the benefits of various microarchitecture features should be quantified properly so that those leading to effective and efficient implementations can be identified. Moreover, the benefits of such features differ among the various areas of application; for example, the characteristics of commercial applications differ dramatically from those of numeric-intensive, CPU-intensive or embedded applications, making it necessary to understand the effects of all such workloads on processor performance.

This report is a Users' Guide for *Turandot*, a highly parameterized PowerPC-based superscalar processor model suitable for exploring a range of microarchitecture features. *Turandot* is a member of *The MET*, a set of tools for supporting fast simulation of microprocessor configurations. These tools permit throughput in excess of 100 million simulated processor cycles per hour (on a contemporary workstation circa 2000). We first provide an overview of the simulation environment, then directions to find the source code and prebuilt processor models, how to execute the prebuilt models, how to interpret the results generated by the models, and finally how to build new models with different features and/or microarchitectures.

Turandot is a tool for developing an understanding of the limits and potentials of PowerPC-based superscalar processors. Consequently, *Turandot* does not attempt to accurately model any specific processor implementation; instead, it attempts to model a generalized (perhaps idealized) processor suitable to explore some of the many variables involved. The results provided by *Turandot* should be used to *determine trends* in performance values as features are changed, not as precise performance predictions of a specific configuration. Models may *approximate* the features of existing PowerPC processors, but the degree of approximation is dependent on how closely the features of those processors can be mapped to *Turandot's* capabilities and parameters. Users of the models should be aware of these limitations, and exercise caution when comparing the results generated by *Turandot* with those obtained from measurements in actual hardware. *Turandot* has been validated in detail for some specific processors, in particular a pre-silicon model of Power4.

Turandot supports the exploration of microarchitecture features through extensive parameterization. Parameters include changing the size of the various resources, the number and latency of functional units, the number of pipeline stages, enabling/disabling features, and so on. All the parameters available in *Turandot* are listed in Appendices A and B.[1]

Results obtained from using The MET and models based on *Turandot* have been reported in [1-3]. Further details on the microarchitecture characteristics implemented by *Turandot* are described in [4].

The instructions given in this guide for building and running models to be used with *Aria* (dynamic trace generation) assume that the compilation process is performed under AIX 4.3 but using the *Aria* methodlogy for AIX 4.1. The resulting models can be executed under AIX 4.3, though. Achieving this behavior requires the installation of *Aria* invoking the `make41` script found in the *Aria* installation directory (after execution the script `configure`).

1. Although many combinations of parameters in Turandot have been exercised, not all possible combinations have been fully tested. As a result, it is conceivable that some combinations of parameters may fail to operate properly at simulation time. Users beware.

2. Overview of the simulation environment

Turandot can be used in two possible modeling scenarios:

Static-tracing, wherein the processor model is fed an existing instruction execution trace, instructions and associated addresses, in FF52 format. This trace only contains the instructions actually executed to completion by the program (e.g., only instructions in the taken path of the program).

The static-trace modeling scenario uses the following components:

- a *Turandot*-based processor model;
- a FF52 instruction execution trace to be fed to the processor model;
- a *pseudo-xcoff* file generated from the FF52 trace, which corresponds to a “binary image” of the program that generated the trace; and
- a file with predecoded information obtained from the *pseudo-xcoff* file.

Dynamic-tracing, wherein the processor model is fed an execution trace generated on-the-fly, under control of the processor model. This trace may include the execution of predicted paths, even if those paths are not actually taken by the program, so the effects of mispredicted instructions can be taken into account by the processor model controlling the generation of the trace.

The dynamic-trace modeling scenario uses the following components:

- *Aria*, the dynamic trace generator;
- a *Turandot*-based processor model;
- the object (*xcoff*) file of the program to be used in the modeling session, with its corresponding inputs; and
- a file with predecoded information obtained from the *xcoff* file of the program to be used.

Both modeling scenarios collect summary data regarding the utilization of processor resources, and generate periodic reports indicating the number of instructions processed up to that point. Moreover, both scenarios can also generate a cycle-by-cycle description of the state of the processor and the flow of instructions through the pipeline.

Aria is the component of *The MET* which dynamically generates the execution trace analyzed by the *Turandot*-based processor model. *Aria* dynamically instruments the instructions which are about to be executed by the input program, on a basic-block by basic-block basis. The instrumented code generates the execution trace representing the effects of the original instructions. Translated blocks are saved, so they are not translated repeatedly. Additional information about *Aria* is given in [5] as well as in *Aria*’s documentation in the software distribution.

Predecoding step

The file with predecoded information mentioned above, both for static and dynamic trace modeling, is used for reducing the instruction decoding overhead during simulation. The file includes the type of each instruction in the program (e.g., in the *xcoff* or *pseudo-xcoff* file), its latency, the resources required by the instruction, and so on. In addition, the file contains predecoded information regarding the placement of the instructions within their corresponding instruction cache line, which is used to reduce the overhead in simulating the fetching of instructions. More specifically, the predecoded information addresses the following features of the processor model:

Latency of operations: Number of cycles required by the execution of an operation. For the case of non-pipelined operations, such as divide and square-root, this is also the occupancy of the corresponding functional unit.

Availability: Number of cycles required prior to use the result from an operation as input to another one. This feature captures the behavior of long bypasses, wherein the result from one operation cannot be fed as input to another operation in the same cycle.

Extra availability: Number of additional cycles required by some operations to allow using its results as input to another operation. This feature captures the behavior of some operations that deviate from the normal operation of a similar base instruction (such as a sign-extended load with respect to a zero-extended load instruction). The value specified corresponds to the extra cycles beyond the normal availability.

Functional unit: Assignment of an operation to a specific functional unit. This feature captures the ability to execute the various classes of operations either in dedicated or general-purpose functional units.

I-cache: Organization of instruction cache: line size, alignment and size of fetch block.

I-expansion: Expansion (decomposition) of specific instructions into simpler instructions.

Predication: Features for evaluating predicated execution in PowerPC.

Note that some of the predecoded information depends only on the instruction set architecture, whereas other depends on the implementation of the processor. In general, the file with predecoded information is specific to the processor configuration being modeled, thus it should be generated individually for each processor model.

The description of all the arguments to the predecoding step, and the functionality of these arguments, is given in Appendix B. Some of the arguments need to be consistent with compilation parameters given when building models; the processor model checks for proper consistency at run time, whenever needed.

Predecoded information cannot be generated for dynamically linked libraries (DLLs) that may be invoked during the execution of a program under the dynamic tracing scenario. In such a case, the predecoded information is generated the first time that a given DLL is encountered.

3. Source code and prebuilt *Turandot* models

The source code necessary for creating new models using *Turandot*, as well as a few prebuilt models and a collection of support tools, are available at the MET root directory. This directory is location-dependent. At IBM T.J. Watson Research Center, this directory is found at

```
$METROOT = /.../watson.ibm.com/fs/projects/MET/ExtDist
```

The directory structure at this place contains the following subdirectories:

`/Source`	source code for *Aria*, *Turandot*, and other related tools.
`/Source/bin`	executable files for various support tools used in conjunction with *Turandot*.
`/Models`	prebuilt versions of *Turandot* for some generic processor configurations.
`/Scripts`	sample scripts used to build and run models, in either one of the modelings scenarios.
/Docs	PDF and text files describing some features of the toolset. Detailed documentation on *Aria* is available in Source/aria/Docs.

3.1 Prebuilt models

The Models subdirectory contains prebuilt models of some generic processor configurations. In particular, this subdirectory contains the following models:

`io4`	generic in-order four-issue superscalar processor;
`io4infcache`	same as io4 but with infinite size caches and infinite size TLBs;
`oo4`	generic out-of-order four issue superscalar processor;
`oo4infcache`	same as oo4 but with infinite size caches and infinite size TLBs.

There are several possible executable files for these models, whose name differ by a prefix, as follows:

`no prefix`	model to be used with *Aria*, in a dynamic-trace modeling session;
`tr`	model to be used with *Aria*, in a dynamic-trace modeling session, but generating detailed information at every processor cycle regarding the status of the processor and the flow of instructions through the entire pipeline;
`pp`	model to be used with *Aria,* in a dynamic-trace modeling session, but generating a timeline of the flow of each instruction through the pipeline, indicating when each instruction enters the major pipeline stages;
`ff`	model to be used with existing instruction execution traces in FF52 format, in a static-trace modeling session;
`fftr`	model to be used with existing instruction execution traces in FF52 format, in a static-trace modeling session, but generating detailed information at every processor cycle regarding the status of the processor and the flow of instructions through the entire pipeline;
`ffpp`	model to be used with existing instruction execution traces in FF52 format, in a static-trace modeling session, but generating a timeline of the flow of each instruction through the pipeline, indicating when each instruction enters the major pipeline stages.

4. Running a static-trace modeling experiment

Using a processor model with an execution trace in FF52 format implies the following steps (see Figure 1; the contents in this figure are available in the file `/Scripts/Run_ff` in the MET distribution):

1. Create a *pseudo-xcoff* file from the FF52 trace, using the tool `ff2pseudo` available in the `$METROOT/bin` subdirectory. The corresponding command line is

   ```
   ff2pseudo [-dm] [-ds] [-h] [-t] [-o <pseudo-file>] [--] <FF52-trace>
   ```

   wherein the optional arguments have the following function:

`-dm`	show progress message every 1M words processed from the trace file
`-ds`	print statistics regarding the entire trace
`-h`	display help message
`-t`	trace processing of the FF52 trace
`-o`	specify the name of the pseudo-xcoff file created

   The `ff2pseudo` command needs to be invoked only once on a given trace; the resulting file is usable in any modeling session.

2. Create the file with predecoded information, using the pseudo-xcoff file generated in (1) as input. The corresponding command line is

   ```
   ffdep_prep [-h] [-predecoded-arguments] [-o <dep-file>] <pseudo-file>
   ```

   wherein the optional arguments have the following function:

`-h`	display help message, including a description of all optional arguments
`-o`	specify the name of the predecoded file created

   The predecoded-arguments is a list indicating the value of parameters to be used in the generation of the predecoded file. These include the type of each instruction, its latency, the resources required by the instruction, the placement of instructions with respect to the instruction cache line, and so on. The complete list of these arguments is given in Appendix B.

3. Initiate the simulation. The corresponding command line is

   ```
   <model-name> -dep <dep-file> [-help] [-cutoff cutoff-value]
         [-skip skip-value] [-sample skip-value sample-period]
         < <FF52-trace> |& tee <log-file>
   ```

   The processor model receives the input FF52 trace through redirection of standard input (the < operator). The arguments have the following function:

   `-h|-help` display help message, including a description of all optional arguments

   `-d|-dep <dep-file>`
   : specify the name of the file with predecoded information

   `-c|-cutoff cutoff-value`
   : specify the number of processor cycles to be simulated

   `-s|-skip skip-value`
   : specify a number of instructions to skip before performing simulation

   `-m|-sample skip-value sample-period`
   : specify a sampling period composed of number of instructions to skip followed by the number of instructions in the sampling period

   `<log file>`
   : specify a file where to save the results that are produced to standard output (using the "tee" operator).

Since the input FF52 trace is provided to the model through standard input, a compressed trace can be decompressed and provided through a pipe to the model, for example as follows:

```
zcat <FF52-trace> | <model-name> -d <dep-file> [-help]
      [-c cutoff-value] [-s skip-value]
      [-m skip-value sample-period] |& tee <log-file>
```

```
#!/bin/csh -xef
# Steps for using Turandot in static-trace modeling with model ffoo4
# Processor model from distribution directory
#
# Distribution directory
setenv METROOT            /.../watson.ibm.com/fs/projects/MET/ExtDist
setenv METBIN             $METROOT/Source/bin
setenv MODELDIR           $METROOT/Models
#
# Input trace to be analyzed
setenv FFTRACE            trace_path_name       # FF52 trace file path name
setenv FFNAME             trace_name            # FF52 trace file name
#
# Processor model
setenv MODEL              ffoo4                 # model name
#
# Intermediate files
setenv PXFILE             $FFNAME.px            # pseudo-xcoff file name
setenv DEPFILE            $FFNAME.dep           # predecoded file name
setenv LOG                logfile_name          # name of log file for results
setenv CUTOFF             cutoff_value          # max. number of cycles to simulate
#
# This is the entire preparation and simulation process
#
# Create the pseudo-xcoff file from the trace
#$METBIN/ff2pseudo -o $PXFILE $FFTRACE
# or, in the case of a compressed trace
zcat $FFTRACE | $METBIN/ff2pseudo -o $PXFILE
#
# Create $DEPFILE, the preprocessed file for input trace
# Optional arguments added as needed (see Appendix B)
$METBIN/ffdep_prep -o $DEPFILE $PXFILE
#
# Start Turandot; input from trace file
#$MODELDIR/$MODEL -dep $DEPFILE -cutoff $CUTOFF < $FFTRACE |& tee $LOG
# or, in the case of a compressed trace
zcat $FFTRACE | $MODELDIR/$MODEL -dep $DEPFILE -cutoff $CUTOFF |& tee $LOG
```

Figure 1: Summary of static-trace modeling session with prebuilt model ffoo4

5. Running a dynamic-trace modeling experiment

Using a processor model with an execution trace generated dynamically by the companion tool *Aria* implies the following steps (see Figure 2; the contents in this figure are available in the file `/Scripts/Run_aria` in the MET distribution):

1. Create the file with predecoded information from the *xcoff* file of the program to be used in the simulation. The corresponding command line is

   ```
   dep_prep [-h] [-predecoded-arguments] [-o <dep-file>] <xcoff-file>
   ```

   wherein the optional arguments have the following function:

`-h`	display help message, including a description of all optional arguments
`-o`	specify the name of the predecoded file created

   The predecoded-arguments is a list indicating the value of parameters to be used in the generation of the predecoded file. These include the type of each instruction, its latency, the resources required by the instruction, the placement of instructions with respect to the instruction cache line, and so on. The complete list of these arguments is given in Appendix B.

2. Set the read/write permissions of the *xcoff* file and the processor model executable file so they are readable/writable only by owner and group (not accessible by others).

3. Set the environment variable `LIBPATH` to point to the directory containing the Aria library `libaria41.a` (this library is available in the `/aria/lib` directory in the MET repository).

4. Initiate the simulation. The corresponding command line is

   ```
   aria41 --MODEL <model-name> -dep <dep-file> [-help] [-cutoff cutoff-value]
            [-skip skip-value] [-sample skip-value sample-period]
          --PROGRAM <xcoff-file> [prog-args] |& tee <log-file>
   ```

   wherein the optional arguments have the following function:

`-h|-help`	display help message, including a description of all optional arguments
`-d|-dep <dep-file>`	specify the name of the file with predecoded information
`-c|-cutoff cutoff-value`	specify the number of processor cycles to be simulated
`-s|-skip skip-value`	specify a number of instructions to skip before performing simulation
`-m|-sample skip-value sample-period`	specify a sampling period composed of number of instructions to skip followed by the number of instructions in the sampling period
`prog-args`	arguments used as input to the program being used to generate the trace
`<log file>`	specify a file where to save the results that are produced to standard output (using the “tee” operator).

The <model-name> is the full path to the corresponding executable file containing the model. For simulation purposes, this file cannot be globally readable; consequently, the user carrying out the simulation must at least be in the same group as the owner of the file with the model.

```
#!/bin/csh -xef
# Steps for using Turandot in dynamic-trace modeling with model oo4
# Processor model from distribution directory
#
# Distribution directory
setenv METROOT          /.../watson.ibm.com/fs/projects/MET/ExtDist
setenv METBIN           $METROOT/Source/bin
setenv ARIABIN          $METROOT/Source/aria/bin
setenv LIBPATH          $METROOT/Source/aria/lib
setenv MODELDIR         $METROOT/Models
#
# Input program to be analyzed and its inputs
setenv PROGPATH         prog_path_name        # xcoff file path name
setenv PROGNAME         prog_name             # xcoff file name
setenv PROGINPS         prog_inputs           # program inputs
#
# Processor model
setenv MODEL            oo4                   # proc. model name from distribution dir.
#
# Intermediate files
setenv DEPFILE          $PROGNAME.dep         # predecoded file name
setenv LOG              logfile_name          # name of log file for results
setenv CUTOFF           cutoff_value          # max. number of cycles to simulate
#
# This is the entire preparation and simulation process
#
# Create $DEPFILE, the preprocessed file for input program
# Optional arguments added as needed (see Appendix B)
$METBIN/dep_prep -o $DEPFILE $PROGPATH
#
# Start Aria+Turandot, with program and its inputs
$ARIABIN/aria41 --MODEL $MODELDIR/$MODEL -dep $DEPFILE -cutoff $CUTOFF \
                --PROGRAM $PROGPATH $PROGINPS \& tee $LOG
```

Figure 2: Summary of dynamic-trace modeling session with prebuilt model oo4

6. Interpreting the results

Turandot produces various results during its execution, as well as extensive summaries upon completion of execution. These results are in a simple text format, so they are amenable for input to post-processing scripts for further analysis and summarization, as described next.

6.1 Periodic results

Periodically, *Turandot* produces lines containing the current value for the measures listed below, in the following format (wherein the two @@ symbols in each line are used to distinguish an output line generated by the model from output lines generated by the program that is being traced, when applicable):

```
       cycle   fetch   total   retir   mispr   ldstc   dline  [hd tl]
   @@  nnnnnn  nnnnnn  nnnnnn  nnnnnn  nnnnnn  nnnnnn  nnnnnn   nn nn
```

The specific measures are current values for:

`cycle`	number of cycles executed;
`fetch`	number of PowerPC instructions fetched;
`total`	number of primitive operations executed (after decode/expansion);
`retir`	number of primitive operations retired;
`mispr`	number of mispredicted branches;
`ldstc`	number of load/store reorder conflicts; and
`dline`	number of data cache lines transferred.

(The rightmost two columns contain number associated to internal structures of the model.)

6.2 Results at end of execution

At the end of execution, *Turandot* displays the following information:

- Values of all the parameters used during simulation (see Appendix A and B)
- Summary results (see Table 1)
- Histograms regarding utilization of queues and pipeline resources (see Table 2), consisting of entries with the following format:

```
            resource name
      size:cycles   n
```

  The tuple `size:cycles` corresponds to the number of cycles that the corresponding resource had the associated size (length). The number *n* is the percentage of total cycles at that size.

- Frequency of reasons for failing to retire instructions (traumas, see Table 3). These data list the number of cycles when fewer than the maximum possible number of operations were retired, classified by the reason attributed to that condition.

6.3 Scripts to summarize and analyze the results

The results generated by Turandot at the end of execution are given in a simple text format that is amenable for post-processing by scripts for summarizing and analysis, as well as for transformation into other formats suitable for inputs to other tools such as a spreadsheet. The MET subdirectory `/Scripts` contains various scripts (in Perl) that summarize and display relevant statistics and information from a simulation run. In most cases, the script is simply invoked followed with the name of the file containing the output generated by the processor model (the log file), as follows:

```
script_name log_file
```

The available scripts are prefixed with “met”. Some scripts accept multiple log file names in the invocation, displaying the results for each file specified. Available scripts perform the following functions:

`metcpi`	Display the cycles per instruction (CPI) and instructions per cycle (IPC).
`metcfg`	Display the configuration of the processor modeled.
`metsum`	Display a one-page summary of relevant performance and resource usage statistics.
`metpars`	Displays all the parameters used in the simulation.

6.4 Detailed results at every cycle

Turandot can be used to build models which provide a detailed description of the flow of instructions through the pipeline at every processor cycle. (Models such as these are named with the prefix “tr” in the subdirectory `/Models` in the MET repository.) The information generated at every cycle includes:

- state of the memory subsystem;
- current sizes of various queues;
- register rename status;
- detailed information about instructions currently executing, including pipeline stage, rename, mispredicted status, etc.

A lot of information is generated at every cycle; the main purpose of such detailed results is for debugging a processor model. To facilitate the process, this mode of operation of the models accepts two additional optional parameters after the predecoded file name in the command line. For the case of a static-trace modeling session, the command line is as follows:

```
<model-name> -dep <dep-file> [start-trace] [stop-trace] < <FF52-trace>
      |& tee <log-file>
```

This invocation of the model will simulate until `start-trace` cycles without reporting the detailed information, and then will report detailed information until `stop-trace` cycles, at which point the simulation will be terminated. The command line format is similar for the case of a dynamic-trace modeling session.

The description of all the information reported every cycle is given in Table 4.

6.5 Pipeline timeline display

Turandot can be used to build models which provide a graphical description (in text format) of the flow of each instruction through the pipeline, as a timeline chart. (Models such as these are named with the prefix “pp” in the subdirectory `/Models` in the MET repository.) The format of each output line in this case is as follows:

```
000000002 [.F.DE.di0.f.....c...........] 000000016 fff00100 00000000   addi      r1,r0,1000
000000003 [.F.DE.d..i3.h.f.c...........] 000000016 fff00104 000007d0   lwzux     r2,r1,r1
000000003 [...DE.d..i1.f...c...........] 000000016 fff00104 000007d4 + lwzux     r2,r1,r1
```

Using the first line as example, the meaning of the data in the timeline is as follows:

000000002 instruction sequence number (this number does not start at 1 for arbitrary reasons)

`[.F.DE.di0.f.....c...........]`
this is the actual timeline displaying the flow of the instruction through the pipeline, wherein the symbols correspond to the various pipeline stages:

F	fetch;
D	decode;
E	decode (stage 2), if applicable;
d	dispatch;
i	issue for execution;
#	execute in unit number #;
f	finish execution, if applicable;
c	complete (retire);
h	hit in cache;
!	
s	

`00000016` cycle when instruction completed execution;

`fff00100` instruction address;

`000007d0` data address (when applicable, otherwise set to all zeroes);

+ indicator that original PowerPC instruction has been decomposed into multiple internal operations (eg., “cracked”);

`addi   r1,r0,1000`
instruction in assembly language representation;

Note that the timeline is a "modulo window" into the execution trace; that is, this window rolls around from the rightmost end to the lefmost end every N cycles, where N is the number of characters in the timeline.

7. Building new models

Building a new model using *Turandot* requires compiling the source code. The parameters are specified as compile time options, using the -D flag to the compiler. This process is illustrated in Figure 3, for the case of dynamic-tracing and static-tracing (prefix “ff”) modeling scenarios. The figure shows a Makefile containing a set of microarchitecture parameters in the variable CFGFLAGS, which are passed in the command line invoking the compiler. The contents of this figure are available in the file `/Scripts/make_models` in the MET distribution.

All the parameters available in *Turandot* are listed in Appendix A; all of them have a default value, listed in the appendix, which is used whenever no -D flag is given at compile time. In other words, the -D compile time flag overrides the default value listed in the appendix.

```
# This Makefile is an example for building Turandot-based models
# Variations in the model (with respect to the default parameters)
# can be specified in the variable CFGFLAGS below
#
# The model is built by invoking
#     make turandot     - build a model for use with Aria
#     make ffturandot   - build a model that reads a ff52 trace
#
# The name of the model built is specified in the variable TARGET;
# the ff52 version name is automatically prefixed with "ff"
#
# For faster compilation time, use -O in STD_CFLAGS; for faster execution, use -O2
# Source directories
METROOT     = /.../watson.ibm.com/fs/projects/MET/ExtDist
ARIADIR     = $(METROOT)/Source/aria
OPCDIR      = $(ARIADIR)/opcode
OPCLIBDIR   = $(ARIADIR)/lib
TRANSDIR    = $(ARIADIR)/translate
TRANSLIBDIR = $(ARIADIR)/lib
TURANDIR    = $(METROOT)/Source/turandot
SRCDIR      = $(TURANDIR)/src
# Name of model to be created
TARGET      = infcache
# Parameters defining the processor model that differ from the default values
CFGFLAGS    = -DINFINITE_CACHE=TRUE
#Compilation flags
ARIAINCDIRS = -I$(OPCDIR) -I$(OPCDIR)/include \
              -I$(TRANSDIR) -I$(TRANSDIR)/include \
              -I$(TURANDIR)/reader -I$(TURANDIR)/aria_reader \
              -I$(TURANDIR)/deps
FFINCDIRS   = -I$(OPCDIR) -I$(OPCDIR)/include \
              -I$(TURANDIR)/ffreader -I$(TURANDIR)/deps
LIBDIRS     = -L$(OPCLIBDIR) -L$(TRANSLIBDIR)
LIBS        = -lopc41
CC         = xlc
STD_CFLAGS  = -O -qspill=1024 -qmaxmem=4000 -DNDEBUG=1
DEBUG_CFLAGS= -O -qspill=1024 -qmaxmem=4000
TRACE_CFLAGS= -O -DNDEBUG=TRUE -DTRACE=TRUE

LDFLAGS        =
```

Figure 3: Sample Makefile for building processor models based on *Turandot*

```
LINKER      = $(CC)

FFAUX_SRCS  = $(TURANDIR)/deps/ffdep_prep_info.c \
              $(TURANDIR)/deps/ffdep_prep_info_table.c

XAUX_SRCS   = $(TURANDIR)/reader/reader_prep_info.c \
              $(TURANDIR)/deps/dep_prep_info.c

SRCS        = $(AUX_SRCS) turandot.c

FF52AUX_OBJS= $(FFAUX_SRCS:.c=.o) $(TURANDIR)/ffreader/trace_ff52reader.o
ARIAAUX_OBJS= $(XAUX_SRCS:.c=.o)  $(TURANDIR)/aria_reader/aria_reader.o

turandot:  turandot.o $(ARIAAUX_OBJS) $(TRANSLIBDIR)/libaria41.a
          $(LINKER) -o $(TARGET) $(LDFLAGS) $(LIBDIRS) turandot.o \
          $(ARIAAUX_OBJS) $(LIBS) -laria
          chmod 750 $(TARGET)
          rm turandot.o

ffturandot:ffturandot.o $(FF52AUX_OBJS)
          $(LINKER) -o ff$(TARGET) $(LDFLAGS) $(LIBDIRS) ffturandot.o \
          $(FF52AUX_OBJS) $(LIBS)
          rm ffturandot.o

turandot.o:sources
          $(CC) -o $@ $(STD_CFLAGS) $(ARIAINCDIRS) $(CFGFLAGS) \
          -DUSING_SPECULATION=1 -DUSING_ARIA=1 \
           -c $(SRCDIR)/turandot.c

ffturandot.o:sources
          $(CC) -o $@ $(STD_CFLAGS) $(FFINCDIRS) $(CFGFLAGS) \
          -DUSING_FF52_SEGS=1 -DUSING_FF=1 \
           -c $(SRCDIR)/turandot.c

sources:                          \
          $(SRCDIR)/turandot.c           \
           ...
           ...
```

Figure 3 (cont.): Sample Makefile for building processor models based on *Turandot* (cont.)

8. References

[1] M. Moudgill, J-D. Wellman, J.H. Moreno, "Environment for PowerPC microarchitecture exploration," *IEEE Micro*, Vol. 19, No. 3 , pp. 15-25, May/June 1999.

[2] M. Moudgill, P. Bose, J.H. Moreno, "Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration," *IEEE International Performance, Computing and Communications Conference* (IPCCC), February 1999.

[3] J.H. Moreno, M. Moudgill, J-D. Wellman, P. Bose, L. Trevillyan, "Trace-driven performance exploration of a PowerPC 601 OLTP workload on wide superscalar processors," *IBM Research Report* RC20962, 1997.

[4] M. Moudgill, J.H. Moreno, “Turandot, a PowerPC-based wide-issue superscalar processor model for microarchitecture exploration,” in preparation.

[5] J.D. Wellman, “Aria, an execution-simulation environment for microarchitectural analysis tools,” in preparation.

Table 1: Summary results reported at end of simulation

Line	Line contents	Description
Totals	cycles=	cycles measured
	insns=	instructions executed during those cycles (after expansion)
	memops=	memory operations completed
	retired=	instructions retired (excludes expansion)
	totalret=	operations retired (includes expansion)
	totalexecd=	operations executed (includes expansion)
	totaldispd=	operations dispatched (includes expansion)
I-cache	probes=	I-cache accesses
	l1miss=	I-cache misses
	prehit=	prefetch buffer hits
	l15 miss=	I-cache misses that were also L1.5 misses
	l2miss=	I-cache misses that were also L2 misses
	tlb1 =	first level TLB misses generated by instruction fetch
	tlb2 =	second level TLB misses generated by instruction fetch
D-cache	probes=	D-cache accesses
	l1lines_started=	D-cache lines accesses started
	l1lines_fetched=	D-cache lines transferred into L1 (actual cache misses)
	l2lines=	D-cache lines transferred into L2 (actual cache misses)
	l1miss_all=	D-cache “misses”: total (includes misses resolved by earlier misses)
	l1miss_taken=	D-cache “misses”: taken path only
	l2miss_all=	D-cache “misses “that were also L2 misses: total
	l2miss_taken=	D-cache “misses “that were also L2 misses: taken path
	trail =	D-cache probes to lines being brought in (trailing-edge effect)
	cast =	D-cache lines cast out
D-TLB	tlb1_all=	first level TLB misses generated by data accesses: total
	tlb1_taken=	first level TLB misses generated by data accesses: taken path
	tlb2_all=	second level TLB misses generated by data accesses: total
	tlb2_taken=	second level TLB misses generated by data accesses: taken path
Interleave		(These results are applicable only for the case of 2 load/store units)
	conflicts=	simultaneous accesses to the same data cache bank but to different cache lines
	samebank=	simultaneous accesses to the same data cache bank and same cache line
I-fetch0 (no instructions were fetched in a cycle)	tlb=	TLB miss
	cache=	I-cache miss
	nfa=	NFA mispredict
	mispredict=	no instructions in trace (only for static trace simulation)
	ibuf=	instruction buffer full
	fchblk=	maximum number of fetch blocks exceeded
	sync=	sync instruction
	max_pred=	maximum number of predicted branches exceeded
	partial=	decomposed complex instruction does not fit in instruction buffer
	inflight=	maximum number of instructions in-flight exceeded
	end=	end of trace
Decode0 (no instructions were decoded)	empty=	instruction buffer empty
	stall=	
	multi=	completing decoding of complex operation
	string=	completing decoding of string operation
Dispatch0 (no instructions were dispatched)	groups=	maximum number of groups exceeded
Prefetch buffer	hits=	prefetch buffer hits
	total=	total lines prefetched
Expansion	insns=	instructions that were expanded
	total=	operations after expansion

Table 1: Summary results reported at end of simulation

Line	Line contents	Description
String operations	insns=	load/store string instructions processed
	total=	primitive operations produced from the load/store string instructions
	stalls=	stall cycles due to string instructions not being decoded because the length field in XER was not yet available
NFA	link=	NFA predictions based on Link Register stack
	nfa=	NFA predictions based on BTAC
	mis=	NFA predictions which do not match branch prediction
	mispred =	NFA mispredictions on taken path
Branch (all paths, in branch predictor)	total =	branch instructions (includes unconditional)
	stalls=	cycles fetching from incorrect path
	full=	cycles with no instructions fetched from incorrect path
	cond_mis=	mispredicts in conditional branch instructions
	cond_tot=	conditional branch instructions
	link_mis=	mispredicts in branch through link register instructions
	link_tot=	branch through link register instructions
	ctr_mis=	mispredicts in branch through counter register instructions
	ctr_tot=	branch through counter register instructions
Actual branch (taken path only, in branch pre-dictor)	total =	branch instructions (includes unconditional)
	stalls=	cycles fetching from incorrect path
	full=	cycles with no instructions fetched from incorrect path
	cond_mis=	mispredicts in conditional branch instructions
	cond_tot=	conditional branch instructions
	link_mis=	mispredicts in branch through link register instructions
	link_tot=	branch through link register instructions
	ctr_mis=	mispredicts in branch through counter register instructions
	ctr_tot=	branch through counter register instructions
Branches executed (in branch unit)	total =	branch instructions (includes unconditional)
	stalls=	cycles fetching from incorrect path
	full=	cycles with no instructions fetched from incorrect path
	cond_mis=	mispredicts in conditional branch instructions
	cond_tot=	conditional branch instructions
	link_mis=	nmispredicts in branch through link register instructions
	link_tot=	branch through link register instructions
	ctr_mis=	mispredicts in branch through counter register instructions
	ctr_tot=	branch through counter register instructions
Prediction	early =	mispredicted branches that were evaluated early
	mispredict=	mispredicted branches
	total =	predicted branches (i.e. all but unconditional branches)
Others	flushes=	pipeline flushes (due to any reason)
	groups=	groups dispatched
Stalls	ibuf=	fetch stall cycles due to I-buffer full
	inflight=	fetch stall cycles due to in-flight limit
	dmissq=	memory unit stall cycles due to full miss queue
	cast=	memory unit stall cycles due to full castout queue
	storeq=	memory unit stall cycles due to full store queue
	reorderq=	memory unit stall cycles due to full reorder queue
	resv=	dispatch stall cycles due to full issue queues
	rename=	rename stall cycles due to lack of instructions
Store queue	total=	insertions into store queue
	forward=	data forwarded to load instructions (load instructions hitting store queue)
	partial=	partial matches in queue (byte/halfword/word)
	data_wait=	load operations waiting for data to be forwarded
Reorder buffer	total =	number of insertions into reorder buffer
	conflicts=	number of conflicts detected

Table 1: Summary results reported at end of simulation

Line	Line contents	Description
mtsr	total= ipurge= dpurge=	mtsr instructions executed I-TLB invalidates arising from mtsr instructions D-TLB invalidates arising from mtsr instructions
nops	dispd= retd=	no-op instructions (ori Rx,Ry,0) dispatched no-op instructions (ori Rx,Ry,0) retired

Table 2: Histograms of resource utilization (reported at end of simulation)

Name	Description
ibuf	Instruction buffer utilization
fchblk	Size of blocks fetched
retireq	Retirement queue
fix_rsv	Integer issue queue (single or first cluster)
fix1_rsv	Integer issue queue (second cluster, if present)
fpu_rsv	Floating-point issue queue (single or first cluster)
fp1_rsv	Floating-point issue queue (second cluster, if present)
mem_rsv	Memory issue queue (single or first cluster)
mem1_rsv	Memory issue queue (second cluster, if present)
br_rsv	Branch unit issue queue
log_rsv	Logical unit issue queue
cmplx_rsv	Complex unit issue queue
tot_rsv	Total across all issue queues
gpr	Free physical general-purpose registers
fpr	Free physical floating point registers
spr	Free physical special-purpose registers
ccr	Free physical condition-code field registers
storeq	Store queue
reorderq	Reorder queue
dmissq	Data cache miss queue
lines	Data cache lines transferred
l2dmissq	L2 D-cache miss queue
l2lines	L2 D-cache lines transferred
dcast	Data cache castout queue
st_commit	Store operations committed (removed from store buffer, after retirement)
retired	Operations retired
st_retd	Store operations retired
fix_exd	Integer operations executed (single or first cluster)
fix1_exd	Integer operations executed (second cluster, if present)
fpu_exd	Floating-point operations executed (single or first cluster)
fp1_exd	Floating-point operations executed (second cluster, if present)
mem_exd	Memory operations executed (single or first cluster)
mem1_exd	Memory operations executed (second cluster, if present)
br_exd	Branch operations executed
log_exd	Logical operations executed
cmplx_exd	Complex operations executed
tot_exd	Total operations executed (across all units)
dispchd	Total operations dispatched (added to all issue queues)
fix_dsp	Integer operations dispatched (added to integer issue queue); single or first cluster
fix1_dsp	Integer operations dispatched (added to integer issue queue); second cluster, if present
fpu_dsp	Integer operations dispatched (added to integer issue queue); single or first cluster
fpu1_dsp	Integer operations dispatched (added to integer issue queue); second cluster, if present

Table 2: Histograms of resource utilization (reported at end of simulation)

Name	Description
mem_dsp	Memory operations dispatched (added to memory issue queue); single or first cluster
mem1_dsp	Memory operations dispatched (added to memory issue queue); second cluster, if present
br_dsp	Branch operations dispatched (added to branch unit issue queue)
renamed	Operations renamed
fetched	Instructions fetched from I-cache
inflight	Operations in-flight
groups	Size of groups

Table 3: Histograms of traumas (reported at end of simulation)

Name	Description
normal	No specific reason (normal execution)
if_nfa	Instruction fetch, next fetch address misspredict
if_tlb1	Instruction fetch, first level TLB miss
if_tlb2	Instruction fetch, second level TLB miss
if_l2	Instruction fetch, L2 instruction cache miss
if_l15	Instruction fetch, L1.5 instruction cache miss
if_l1	Instruction fetch, L1 instruction cache miss
if_pref	Instruction fetch, prefetch buffer miss
if_pred	Instruction fetch, branch misprediction
if_full	Instruction fetch, instruction buffer full
if_flit	Instruction fetch, number of instructions in flight exceeded
if_brch	Instruction fetch,
if_ldst	Instruction fetch, load/store
decode	Decode reason
rename	Rename reason
diq_fix	Integer unit issue queue empty
diq_fpu	Floating-point unit issue queue empty
diq_mem	Memory unit issue queue empty
dir_br	Branch unit issue queue empty
diq_log	Logical unit issue queue empty
diq_cmplx	Complex unit issue queue empty
ful_fix	Integer unit issue queue full
ful_fpu	Floating-point unit issue queue full
ful_mem	Memory unit issue queue full
ful_br	Branch unit issue queue full
ful_log	Logical unit issue queue full
ful_cmplx	Complex unit issue queue full
mm_stqf	Store queue full
mm_stqc	Store queue conflict
mm_roqf	Reorder queue full
mm_dmqf	Miss queue full
mm_dcqf	Castout queue full
mm_stnd	Store whose data is not available
mm_tlb1	First level D-TLB miss
mm_tlb2	Second level D-TLB miss
mm_dl2	L2 D-cache miss
mm_dl1	L1 D-cache miss
rg_fix	Register dependency in integer instruction
rg_fpu	Register dependency in floating-point instruction
rg_mem	Register dependency in memory instruction
rg_br	Register dependency in branch instruction
st_data	Load dependency on store queue, data not available
ret_st	Store retirement
cpi	
totals	
insns	

Table 4: Detailed results reported every cycle (for “tr” models)

Line	Line label	Description
1	CYCLE	Current cycle number
		Current i-fetch trauma (if any). Instruction fetch waiting on:
		if_nfa: next fetch address incorrect
		if_tlb1: i-side level 1 tlb miss
		if_tlb2: i-side level 2 tlb miss
		if_l2: i-side level 2 cache miss
		if_l15: i-side level 1.5 cache miss
		if_l1: i-side level 1 cache miss
		if_pref:
		if_pred:
		if_full:
		if_flit:
		if_brch:
		if_ldst:
		normal:
	[NONE]	Memory status. No memory operations being executed because
		MemDmiss: miss queue full
		MemCast: cast-out queue full
		DTLB: d-side level-2 tlb miss
		D2TLB: d-side level-1 tlb miss
	i=	I-fetch stalled until reported cycle. If reported cycle is close to 2^32-1 (=4294967295), then i-fetch is stalled waiting for some event, as follows:
		...295: trace/program end reached
		...294: unresolved mispredicted branch
		...293: cache mispredict resolution
		...292: mispredicted branch limit exceeded
		...291: synchronizing instruction
	m=	memory units busy till cycle
	d=	earliest cycle d-miss can return
	p=	number of d-cache ports used
	b=	number of unresolved branch predictions
	d=	number of unresolved branches in mispredicted path
	it=	cycle i-tlb miss will be resolved
	dt=	cycle d-tlb miss will be resolved
2	ib=	number of instructions in i-buffer
	fb=	number of fetch blocks in i-buffer
	gr=	number of instruction groups in flight
3	fx=	number of operations in fix-point issue queue
	fp=	number of operations in floating-point issue queue
	mm=	number of operations in memory issue queue
	br=	number of operations in branch issue queue
	lg=	number of operations in logical issue queue
	cx=	number of operations in complex issue queue
	dm=	number of d-cache misses active
	l2=	number of L2 d-cache misses active
	st=	number of operations in store queue
	ro=	number of operations in load reorder queue
	co=	number of lines being cast out
4	n=	number of d-cache misses active
	l2n=	number of L2 d-cache misses acvtive
	l=	number of d-cache missed lines active
	l2l=	number of L2 d-cache missed lines active

Table 4: Detailed results reported every cycle (for “tr” models)

Line	Line label	Description
5	rq=	number of retire queue slots left
	hd=	instruction queue head
	tl=	instruction queue tail
	ren=	instruction queue rename pointer
	dis=	instruction queue dispatch pointer
	gp=	number of general purpose registers in use
	fp=	number of floating point registers in use
	cc=	number of condition code registers in use
	sp=	number of special purpose registers in use
6	hd=	iq head
	dec=	iq decode index
	ren=	iq rename index
	dis=	iq dispatch index
	retire=	iq retire index
	tl=	iq tail
	ro=	reorder queue low position
	[none]	reorder queue high position
	#	reorder queue slots left
	sq	store queue low
	[none]	store queue high
	#	store queue left
7		the architected general purpose registers
8		the physical registers they are mapped to at retire
9		the physical registers they are mapped to at rename
10		the free physical general purpose registers
Instruction queue entries	8-hex:	instruction number (replicated for the case of decomposed instructions)
	[2-hex]	iq index
	2-hex	group id
	8-hex	instruction word
	@8-hex	instruction address
	<8-hex>	data address
	8-hex:	(internal) address of entry
	string	status of instruction
	[8-hex]	cycle at which execution will complete
	chars	various status markers
		! busy
		? mispredicted path
		- mispredicted branch
		+ second or later operation of cracked instruction
		B branch
		C conditional branch
		_ end of group
		| end of fetch block
		? predicated operation
	string	trauma name
	2-dec	trauma id
	2-hex	slot in reorder/store queue
	string	the disassembled instruction
	string	renamed destination registers
	<- string	renamed source registers

Appendix A: Parameters in Turandot (in alphabetical order)

Parameter	Resource	Description	Default	Min.	Max.
ARCH_MAX	Rename	Maximum number of architected registers per class of registers	32		127
BLOCK_ATHEAD_OPS	Load/Store	If **TRUE**, allow certain operations to issue only if they are next to retire	TRUE		
BP_COND_BIMODAL	Branch prediction	If **TRUE**, use bimodal branch prediction algorithm	FALSE		
BP_COND_BITS	Branch prediction	Number of bits per entry in branch prediction table(s)	2	1	2
BP_COND_GSHARE	Branch prediction	If **TRUE**, use G-share branch prediction algorithm	FALSE		
BP_COND_LG2SIZE	Branch prediction	Number of entries in branch prediction table(s)	13		
BP_COND_SELECT	Branch prediction	If **TRUE**, use Alpha-style branch prediction algorithm	TRUE		
BP_COUNTER_ASSOC	Branch prediction	Associativity of counter branch target table	4	4	4
BP_COUNTER_LG2SIZE	Branch prediction	Size of counter branch target table	5		
BP_HISTORY_SHIFT	Branch prediction	Number of bits history is shifted left before being XOR'ed with address in G-share	3		
BP_LINK_SIZE	Branch prediction	Size of return-addresses stack	32		
BRANCH_MAX_PRED	Branch prediction	Maximum number of predicted conditional branches	12		
BR_NUM	Execution units	Number of branch units	2		
BR_RESV	Dispatch	Length of branch issue queue	12		
BR_UNIT	Execution units	Identification of unit for branch operations	3		
CCR_ARCH	Rename	Number of architected CR fields	8		
CCR_PHYS	Rename	Number of physical CR fields	32	16	128
CLUSTER	Execution units	If **TRUE**, the processor has two clusters of integer/load-store units	TRUE		
CLUSTER_FPU	Execution units	If **TRUE**, the processor has two clusters of floating-point units	TRUE		
CMPLX_NUM	Execution units	Number of complex integer execution units	0		
CMPLX_RESV	Execution units	Length of complex integer issue queue	16		
CMPLX_UNIT	Execution units	Identification of unit for complex integer operations	5		
COMBINE_FIX_MEM	Execution units	If **TRUE**, integer operations are executed in load/store execution unit	FALSE		
COMBINE_FIX_NUM	Execution units	Maximum number of integer operations that can be issued per cycle from combined integer-load/store issue queue	0		
COMBINE_MEM_NUM	Execution units	Maximum number of memory operations that can be issued per cycle from combined integer-load/store issue queue	0		

Parameter	Resource	Description	Default	Min.	Max.
COMMIT_STORES_DELAY	Store queue	Delay between writing store operation into the cache (after retirement) and removing the corresponding entry from the store queue	4		
COMMIT_STORES_LATE	Store queue	If **TRUE**, store operations are removed from the retirement queue before the corresponding entry in the store queue is removed	FALSE		
DCACHE_ASSOC	L1-D cache	Associativity of L1 data cache	1	1	4
DCACHE_INTERLEAVE_BANKS	L1-D cache	Number of banks in interleaved L1 data cache	8		
DCACHE_INTERLEAVE_LG2SIZE	L1-D cache	Log2 of number of entries per bank in interleaved L1 data cache	6		
DCACHE_INTERLEAVE_PENALTY	L1-D cache	If bank conflict detected, cycles before operation can be retried in interleaved L1 data cache	5		
DCACHE_IS_INTERLEAVED	L1-D cache	If **TRUE**, L1 data cache is interleaved	FALSE		
DCACHE_LG2ENTRIES	L1-D cache	Log2 of number of lines in L1 data cache	9		
DCACHE_LG2SECTOR	L1-D cache	Log2 of sector size in L1 data cache	6		
DCACHE_LG2SIZE	L1-D cache	Log2 of line size in L1 data cache	7		
DCACHE_PORTS	L1-D cache	Number of L1 data cache ports	2		
DCACHE_SECTORS	L1-D cache	Number of sectors in L1 data cache line	2		
DCACHE_WRITE_PORTS	L1-D cache	Number of write ports in L1 data cache; e.g., maximum number of stores that can be completed per cycle	1		
DCASTOUT_MAX	Miss queue	Maximum number of dirty lines waiting to be written back (total number of dirty lines is actually **DCASTOUT_MAX** + number of memory units - 1)	7		
DCASTOUT_OVERHEAD	Miss queue	Number of cycles the L1-L2 bus is occupied by a transaction	5		
DEBUG	Debugging support	If **TRUE**, generate debugging information	FALSE		
DECODE_ALL_CRACK	Decode	If **TRUE**, all operations from a decomposed (cracked) instruction are placed in same decode group	TRUE		
DECODE_EXTRA_BRANCH	Decode	If **TRUE**, allow a branch operation into a decode group, even if it exceeds DECODE0_MAX	FALSE		
DECODE_MAX	Decode	Maximum number of operations that can be decoded per cycle	4		
DECODE_MAX_INSNS	Decode	Maximum number of instructions that can be decoded per cycle	4		
DECODE_MODEL_REISSUE	Decode	If **TRUE**, decode into groups of a single operation on restart after certain pipeline flushes	TRUE		
DECODE_MULTI_PENALTY	Decode	Number of extra cycles for decoding complex instructions	0		
DECODE_MULTI_SPECIAL	Decode	If **TRUE**, decode complex instructions with DECODE_MULTI_PENALTY	FALSE		
DECODE_ONE_BRANCH	Decode	If **TRUE**, allow at most one branch per decode group	FALSE		

Parameter	Resource	Description	Default	Min.	Max.
DECODE_SLOT0	Decode	If **TRUE**, force certain operations to be the first operation in a decode group	FALSE		
DECODE0_MAX	Decode	Maximum number of non-branch operations that can be decoded per cycle	4		
DISPATCH_GROUPS	Retire	If **TRUE**, dispatch and retire together all the operations of a decode group	TRUE		
DISPATCH_MAX	Dispatch	Maximum number of instructions dispatched	4		
DMEM_LATENCY	Memory	Load-use latency for data cache miss resolved in main memory	40		
DMISS_COUNT_LINES	Results	If **TRUE**, keep track of the number of lines in the load miss queue	TRUE		
DMISS_MAX	Miss queue	Maximum number of *outstanding* misses (total number of misses is actually **DMISS_MAX** + #memory units - 1)	7		
DRAIN_ON_SYNC	I-fetch	Stop fetching instructions after sync instruction is fetched and until sync instruction is retired	FALSE		
DTLB_ASSOC	D-TLB1	Associativity of D-TLB 1	1	1	2
DTLB_LATENCY	D-TLB1	Latency of D-TLB1 miss (miss latency)	4		
DTLB_LG2ENTRIES	D-TLB1	Log2 of the number of entries in D-TLB1	7		
DTLB_LG2SIZE	D-TLB1	Log2 of the page size in D-TLB1	12		
D2CACHE_LATENCY	L2-D cache	Load-use latency of L1 data cache miss that hits in the L2 cache	7		
D2TLB_ASSOC	D-TLB2	Associativity of D-TLB2	4	1	4
D2TLB_LATENCY	D-TLB2	Latency of D-TLB2 miss (miss latency)	40		
D2TLB_LG2ENTRIES	D-TLB2	Log2 of the number of entries in D-TLB2	10		
D2TLB_LG2SIZE	D-TLB2	Log2 of the page size in D-TLB2	12		
EVALUATE_MISPREDICTED_BRANCHES	Branch prediction	If **TRUE**, branches are checked according to parameter EVALUATE_MISPREDICTED_RENAME	TRUE		
EVALUATE_MISPREDICTED_RENAME	Branch prediction	If **TRUE**, branches are checked for misprediction at rename/dispatch stage	TRUE		
EXEC_INORDER	In-order execution	If **TRUE**, do not reorder instructions for execution	FALSE		
EXPAND_INSNS	Decode	If **TRUE**, complex instructions are decomposed into simpler ones (e.g., cracking)	TRUE		
EXPAND_STRINGS_DYNAMIC	Decode	If **TRUE**, expand string instructions based on the number of bytes actually transferred	TRUE		
EXPAND_STRINGS_PENALTY (for load/store string indexed instructions)		Latency from setting the length field in XER to using the value to determine the expansion of string instructions	2		
FIX_NUM	Execution units	Number of fixed-point units	3		
FIX_RESV	Dispatch	Length of fixed point issue queue	20		
FIX_UNIT	Execution units	Identification of unit for integer operations	0		
FPU_NUM	Execution units	Number of floating-point units	2		
FPU_RESV	Dispatch	Length of floating point issue queue	20		
FPU_UNIT	Execution units	Identification of unit for floating-point operations	1		

Parameter	Resource	Description	Default	Min.	Max.
FPR_ARCH	Rename	Number of architected floating-point registers	32		
FPR_PHYS	Rename	Number of physical floating-point registers	64	36	128
GETCPI	Results	If **TRUE**, collect traumas information and print related histograms	TRUE		
GETCPI_REMAINING	Results	If **TRUE**, collect traumas information about operations behind the operation that could not be retired	TRUE		
GPR_ARCH	Rename	Number of architected general-purpose registers	32		
GPR_PHYS	Rename	Number of physical general-purpose registers	64	36	128
GROUPS_MAX	Dispatch	Maximum number of decode groups that can be in flight	16		32
HANDLE_NOPS	Dispatch	If **TRUE**, nop operations are not placed in any issue queue	FALSE		
IBUF_MAX	I-Buffer	Maximum number of fetched instructions which have not yet been decoded (RETIREQ_MAX+IBUF_MAX must be less than 257)	24		255
ICACHE_ASSOC	L1-I cache	Associativity of L1 instruction cache	1	1	4
ICACHE_LG2ENTRIES	L1-I cache	Log2 of number of lines in L1 instruction cache	9		
ICACHE_LG2SIZE	L1-I cache	Log2 of the size of each line in L1 instruction cache	7		
IFETCH_MAX	I-Fetch	Maximum number of instructions to fetch	4		
IFETCH_MULTIPLE_FALLTHRU	I-Fetch	If **FALSE**, fetching stops after the first branch is fetched	TRUE		
IGNORE_SPEC_DMISSES	L1-D cache	If **TRUE**, ignore speculative L1 data cache misses	FALSE		
IMEM_LATENCY	L1-I cache	Latency for instruction cache miss resolved in main memory	40		
INFINITE_CACHE	Cache	If **TRUE,** always hit in all components of the cache hierarchy (caches and TLBs)	FALSE		
INFINITE_DCACHE	L1-D cache	If **TRUE**, always hit in L1 data cache	FALSE		
INFINITE_DTLB	D-TLB1	If **TRUE**, always hit in D-TLB1	FALSE		
INFINITE_D2TLB	D-TLB2	If **TRUE**, always hit in D-TLB2	FALSE		
INFINITE_ICACHE	L1-I cache	If **TRUE**, always hits in L1 instruction cache	FALSE		
INFINITE_I15CACHE	L1.5-I cache	If **TRUE**, always hit in L1.5 instruction cache	FALSE		
INFINITE_ITLB	I-TLB1	If **TRUE,** always hit in I-TLB1	FALSE		
INFINITE_I2TLB	I-TLB2	If **TRUE**, always hit in I-TLB2	FALSE		
INFINITE_L2CACHE	L2 cache	If **TRUE**, always hit in L2 cache	FALSE		
INFLIGHT_MAX	In-flight	Maximum number of instructions in flight	160		255
INORDER_BRANCHES	Execution units	If **TRUE**, no instruction can be issued out-of-order with respect to branch instructions, and branch instructions cannot be issued out-of-order with respect to other instructions	FALSE		

Parameter	Resource	Description	Default	Min.	Max.
IPREFETCH_ENTRIES	I-Prefetch	Number of entries in instruction prefetch buffer	4		
IPREFETCH_LATENCY	I-Prefetch	Number of cycles to transfer from instruction prefetch buffer to processor and L1 instruction cache	1		
IQ_SIZE	Internal data structure	Maximum number of inflight operations	256		256
ITLB_ASSOC	I-TLB1	Associativity of I-TLB1	1	1	2
ITLB_LG2ENTRIES	I-TLB1	Log2 of number of entries in I-TLB1	7		
ITLB_LG2SIZE	I-TLB1	Log2 of the page size in I-TLB1	12		
ITLB_LATENCY	I-TLB1	Latency of I-TLB1 miss (miss latency)	4		
I15CACHE_ASSOC	L1.5-I cache	L1.5 I-cache associativity	1	1	4
I15CACHE_LATENCY	L1.5-I cache	Latency for instruction cache miss resolved in L1.5 instruction cache	4		
I15CACHE_LG2ENTRIES	L1.5-I cache	Log2 of number of entries in L1.5 instruction cache	12		
I15CACHE_LG2SIZE	L1.5-I cache	Log2 of line size in L1.5 instruction cache	7		
I15PREFETCH_HIT_LATENCY	L1.5-I cache	Number of cycles to transfer first line from L1.5 to instruction prefetch butter	5		
I15PREFETCH_NEXT_LATENCY	L1.5-I cache	Number of cycles to transfer next line (after hit) from L1.5 to instruction prefetch buffer	3		
I2CACHE_LATENCY	L2-I cache	Latency for instruction cache miss resolved in L2 cache	7		
I2PREFETCH_HIT_LATENCY	L2-I cache	Number of cycles to transfer first line from L2 to instruction prefetch butter	8		
I2PREFETCH_NEXT_LATENCY	L2-I cache	Number of cycles to transfer next line (after hit) from L2 to instruction prefetch buffer	4		
I2TLB_ASSOC	I-TLB2	Associativity of I-TLB2	4	1	4
I2TLB_LATENCY	I-TLB2	Latency of I-TLB2 miss (miss latency)	40		
I2TLB_LG2ENTRIES	I-TLB2	Log2 of the number of entries in I-TLB2	10		
I2TLB_LG2SIZE	I-TLB2	Log2 of the page size in I-TLB2	12		
L2CACHE_ASSOC	L2 cache	Associativity of L2 cache	4	1	4
L2CACHE_LG2ENTRIES	L2 cache	Log2 of the number of lines in L2	14		
L2CACHE_LG2SIZE	L2 cache	Log2 of the size of each line in L2	7		
LOG_NUM	Execution units	Number of CR logical units	1		
LOG_RESV	Execution units	Length of CR logical issue queue	12		
LOG_UNIT	Execution units	Identification of unit for CR logical operations	4		
MEM_NUM	Execution units	Number of memory units	2		
MEM_RESV	Dispatch	Length of memory issue queue	20		
MEM_UNIT	Execution units	Identification of unit for memory operations	2		
MEMQ_FLUSHES_IMMEDIATELY	Execution units	If **TRUE**, flush the instruction queue on certain kinds of load/store conflicts	TRUE		
MISPREDICT_RECOVERY_CYCLES	Branch prediction	Number of cycles required to recover from a mispredicted branch	0		
MTSR_PURGE_TLB	Execution units	If **TRUE**, MTSR instructions invalidate all entries in DTLB-1	TRUE		

Parameter	Resource	Description	Default	Min.	Max.
MULTI_DECODE_STAGE	Decode	Number of additional stages in decode	2	0	2
NDEBUG	Debugging support	If **TRUE**, do not generate debugging information	TRUE		
NFA_ASSOC	Next fetch address	Associativity of NFA table	4	1	4
NFA_LATENCY	Next fetch address	Number of cycles to recover from NFA misprediction. NFA prediction different from branch prediction (misprediction latency).	2		
NFA_LG2ENTRIES	Next fetch address	Log2 of the number of entries in BTAC	12		
NFA_NEXT_SEQ	Next fetch address	If **TRUE**, fetch next sequential address; if **FALSE,** use NFA table	FALSE		
NO_DECODE_STAGE	Decode	If **TRUE**, decode stage is merged with other stage	FALSE		
NO_PARTIAL_IFETCH	I-fetch	If **TRUE**, do not fetch instruction if all corresponding decomposed operations cannot be placed in instruction queue or instruction fetch buffer	TRUE		
PERFECT_BRANCH_PREDICTION	Branch prediction	If **TRUE**, assume perfect prediction of all branches	FALSE		
PERFECT_COND_BRANCHES	Branch prediction	If **TRUE**, assumes perfect prediction of Branch Conditional (bc) instructions	FALSE		
PERFECT_CTR_BRANCHES	Branch prediction	If **TRUE**, assumes perfect prediction of Branch Conditional to Count Register instructions	FALSE		
PERFECT_LINK_BRANCHES	Branch prediction	If **TRUE**, assumes perfect prediction of Branch Conditional to Link Register instructions	FALSE		
PERFECT_LOAD_REORDER	Load/store reorder buffer	If **TRUE**, reorder only load operations which are guaranteed not to conflict with any store operation	FALSE		
PERFECT_NFA_PREDICTION	Next fetch address	If **TRUE**, always fetch from predicted address	FALSE		
PHYS_MAX	Rename	Maximum number of physical registers for any class	128		
PREFETCH_NEXT_SEQ	Prefetch	If **TRUE**, prefetch next line after miss	TRUE		
PREFETCH_USING_NFA	Prefetch	If **TRUE**, prefetch next line based on NFA table	FALSE		
RECOVER_AT_EVALUATION	Branch prediction	If **TRUE**, branch misprediction forces pipeline flush when branch is resolved (instead of at branch retirement)	TRUE		
RENAME_MAX	Rename	Maximum number of instructions renamed	4		
RENAME_MULTIPLE_BRANCHES	Rename	If **TRUE**, more than one branch instruction can be renamed per cycle	TRUE		
REORDER_ALL_LOADS	Load/store reorder buffer	If **TRUE**, place all load operations in reorder buffer regardless of whether they have been reordered or not	TRUE		
REORDERQ_AT_DISPATCH	Load/store reorder buffer	If **TRUE**, allocate reorder buffer entry at dispatch time; if **FALSE**, allocate reorder buffer entry at issue time	TRUE		

Parameter	Resource	Description	Default	Min.	Max.
REORDERQ_HIWATER	Load/store reorder buffer	Number of entries in the reorder buffer for the case of entries allocated at issue time	30		
REORDERQ_IGNORE_CONFLICT	Load/store reorder buffer	If **TRUE**, do not generate exceptions for conflicts encountered in reorder buffer	FALSE		
REORDERQ_IGNORE_ID	Load/store reorder buffer	If **TRUE**, ignore age in reorder buffer entries (that is, generate exception even for conflicts with more recent -younger- operations)	FALSE		
REORDERQ_LENGTH	Load/store reorder buffer	Number of entries in the load/store reorder buffer	31		31
RETIRE_DELAY	Retire	Number of cycles between write-back and retirement	1		
RETIRE_GROUPS	Retire	If **TRUE**, retire only whole decode groups of instructions	TRUE		
RETIRE_MAX	Retire	Maximum number of instructions retired	8		
RETIREQ_MAX	Retire	Length of the retirement queue (RETIREQ_MAX+IBUF_MAX must be less than 257)	128		255
RF_DELAY	Read	Number of cycles for register read stage	1		
SPR_ARCH	Rename	Number of architected special-purpose registers (SPRs)	32		
SPR_PHYS	Rename	Number of physical special-purpose registers (SPRs)	64	36	128
STATS	Results	If **TRUE**, collect data during simulation and print related histograms	TRUE		
STORE_RETIRE_NOLOAD	Retire	If **TRUE**, write store to cache only if no load operation was executed in that cycle	TRUE		
STORE_RETIRE_PORTS	Retire	If **TRUE**, write store to cache if all ports were not used by load operations	FALSE		
STORE_RETIRE_SEPARATE	Retire	If **TRUE**, write store to cache using separate store ports	FALSE		
STOREQ_AT_DISPATCH	Store queue	If **TRUE**, allocate store queue entry at dispatch time; if **FALSE**, allocate store queue entry at issue time	TRUE		
STOREQ_FWD_AVAIL	Store queue	Number of extra cycles to forward data to pending load operation after data becomes available in store queue	0		
STOREQ_FWD_DELAY	Store queue	Number of extra cycles to forward data to load operation from store queue on store queue hit	2		
STOREQ_HIWATER	Store queue	Number of entries in the store queue for the case of entries allocated at issue time	30		
STOREQ_IGNORE_CONFLICT	Store queue	If **TRUE**, ignore load hitting store entry in store queue	FALSE		
STOREQ_IGNORE_ID	Store queue	If **TRUE**, ignore age in store queue entries (that is, generate exception even for conflicts with more recent -younger- operations)	FALSE		
STOREQ_IGNORE_PARTIAL	Store queue	If **TRUE**, ignore partial overlaps in store queue	TRUE		

Parameter	Resource	Description	Default	Min.	Max.
STOREQ_IGNORE_RETIRED	Store queue	If **TRUE**, do not try to match with retired stores that are in store queue	FALSE		
STOREQ_INORDER_CUTTOFF	Store queue	Number of entries reserved for in-order store operations	4		
STOREQ_LENGTH	Store queue	Number of entries in the store queue	31		
STOREQ_SAME_CYCLE_ABORT	Store queue	If **TRUE**, flush pipe if overlapping load and store operations issue in the same cycle	FALSE		
TIMELINE	Results	If **TRUE**, generate pipeline timeline	FALSE		
UNIFIED_2TLB	I-TLB2	If **TRUE**, use second level D-TLB for instructions	TRUE		
USE_BITS_MEMQ	Store queue	If 0, use all address bits to detect whether load overlaps with store. Otherwise, use lower USE_BITS_MEMQ number of bits to determine overlap.	0		
USE_BLOCK_NFA	Next fetch address	If **TRUE**, use fetch block addresses in NFA table, instead of individual instruction addresses	FALSE		
USE_IPREFETCH	I-Prefetch	If **TRUE**, perform instruction prefetch	TRUE		
USE_I15CACHE	L1.5 I-cache	If **TRUE**, use L1.5 instruction cache	FALSE		
USE_NONPIPE_FIX_OPS	Execution units	If **TRUE**, treat certain fix-point operations as non-pipelineable	FALSE		
USE_NONPIPE_FPU_OPS	Execution units	If **TRUE**, treat certain floating-point operations as non-pipelineable	FALSE		
USE_PREDICATION	I-fetch	If **TRUE**, support predicated operations	FALSE		
USE_PREDICATION_MODE1	I-fetch	If **TRUE**, support predicated operations where the predicated ops are cracked differently based on whether the predicate is true or false	TRUE		
USE_STOREQ_NODATA_CIRCULATE	Execution units	If **TRUE**, recirculate a load operation when the load hits in the store queue but data is not available	FALSE		
USING_ARIA	Simulation	If **TRUE**, use instruction trace generated dynamically by Aria			
USING_CMPLX_UNIT	Execution units	If **TRUE**, support complex instruction unit	FALSE		
USING_FF	Simulation	If **TRUE**, use fF52 trace			
USING_LOG_UNIT	Execution units	If **TRUE**, support CR logic operation unit	FALSE		
USING_SPECULATION	Simulation	If **TRUE**, simulate effects of mispredicted instructions			
VERIFY	Debug		FALSE		
WIDE_ARCH_REG	Rename	If **TRUE**, support more than 64 architected registers per class	FALSE		

Appendix B: Arguments to the predecoding step

The arguments to the predecoding step implemented by the tools `dep_prep` and `ffdep_prep` address the following features of the processor model:

Latency: Number of cycles required by the execution of an operation. For the case of non-pipelined operations, such as integer divide, this is also the occupancy of the corresponding functional unit.

Availability: Number of cycles required before being able to use as input the result from an operation. This argument is intended to capture the behavior of long bypasses, wherein the result from an operation cannot be fed as input to another operation in the same cycle when the result is produced.

Extra availability: Number of additional cycles required by some operations to allow using its results as input to another operation. This argument is intended to capture the behavior of some operations that deviate from the normal operation of a base instruction (such as a sign-extended load with respect to a zero-extended load instruction). The value specified corresponds to the extra cycles beyond the normal availability.

Functional unit: Assignment of operations to specific functional units. This is intended to capture the ability to execute the various classes of operations either in dedicated or general-purpose functional units.

I-cache: Organization of instruction cache: line size, alignment and size of fetch block.

I-expansion: Expansion (decomposition) of specific instructions into simpler instructions (also known as “cracking” in the GP processor).

Predication: Features for evaluating predicated execution.

Table 5 lists all the arguments to dep_prep and ffdep_prep, which are passed as command-line flags (-x arguments) using a keyword followed by a number when applicable. The value for each parameter used in a simulation session is reported as part of the simulation output; each parameter is reported with an explicit name instead of the keyword used as input to dep_prep. For each parameter, Table 5 gives the keyword (shortname) and the longname. Note that there are some parameters reported at simulation time which do not have an equivalent shortname; these parameters are inferred from the value of other parameters specified as input to dep_prep.

Table 5: Arguments to predecoding step (in long-name alphabetical order)

Shortname (command-line keyword)	Longname (reported in simulation output)	Description	Default value
-h		display summary of options	
-aa N	arithmetic_load_avail	additional cycles for data from arithmetic load operation to be available, beyond those of a regular (zero-extended) load operation	0
-la N	arithmetic_load_latency	additional cycles for latency of arithmetic load operation, beyond those of a regular (zero-extended) load operation	0
(internal use)	arithmetic_subword_load_avail	additional cycles for data from arithmetic subword load operation to be available, beyond those of a regular (word-long) arithmetic load operation (the value of this parameter is the maximum among arithmetic_load_avail and subword_load_avail)	0
(internal use)	arithmetic_subword_load_latency	additional cycles for latency of arithmetic subword integer load operation, beyond those of a regular (word-long) arithmetic load operation (the value of this parameter is the maximum among arithmetic_load_latency and subword_load_latency)	0
-ab N	branch_avail	number of cycles for result from branch operation to be available	1
-lb N	branch_latency	latency of branch operation	1
-ub k	branch_unit	unit where branch operations are performed	3
-dc N	cache_size	size of cache line (in words) (reported as cache_align in dep-prep output)	32
(internal use)	cache_mask	(internal use only)	31
-ac N	cmp_avail	number of cycles for result from compare operation to be available	1
-lc N	cmp_latency	latency of compare operation	1
-aL N	crl_avail	number of cycles for result from CR logical operation to be available	1
-lL N	crl_latency	latency of CR logical operations	1
-uL k	crl_unit	unit where CR logical operations are performed	3
-aD N	fdiv_avail	number of cycles for result from floating-point divide operation to be available	21
-lD N	fdiv_latency	latency of floating-point divide operation	21
-da N	fetch_align	alignment at which start fetching blocks from a cache line (in words)	4
(internal use)	fetch_align_mask	(internal use only)	3
-df N	fetch_max	maximum size of fetch block (in words)	4
-ax N	fix_avail	number of cycles for result from integer (fixed-point) operation to be available (other than multiply and divide)	1
-lx N	fix_latency	latency of fixed-point instruction	1
-ux k	fix_unit	unit where fixed-point (integer) operations are performed	0
-sall	flag_all_split	do not expand any operations	expand
-scrl	flag_crl_split	do not expand CR logical operations	expand
-sfpldu	flag_fpldu_split	do not expand floating-point load with update instructions	expand
-sfpst	flag_fpst_split	do not expand floating-point store instructions	expand
-sfpstu	flag_fpstu_split	do not expand floating-point store with update instructions	expand
(internal use)	flag_fxlda_split	do not expand arithmetic load instructions	expand (fixed)
(internal use)	flag_fxldau_split	do not expand arithmetic load with update instructions	expand (fixed)
-sfxldu	flag_fxldu_split	do not expand fixed-point load with update instructions	expand

Table 5: Arguments to predecoding step (in long-name alphabetical order)

Shortname (command-line keyword)	Longname (reported in simulation output)	Description	Default value
-sfxst	flag_fxst_split	do not expand fixed-point store instructions	expand
-sfxstu	flag_fxstu_split	do not expand fixed-point store with update instructions	expand
(internal use)	flag_fxstux_split	do not expand store with update instructions	expand (fixed)
-a N	flag_lg2arch_max	log2 of the number of architected registers	5
-smult	flag_mult_split	do not expand any multi-source/target operations	expand
-sshow	flag_show_split	show the result of all expansions	no show
-pshow	flag_show_pred	enable display of predication	disabled
-sspr	flag_spr_split	do not expand SPR operations	expand
-sstring	flag_string_split	do not expand string operations	expand
-sstu2	flag_stu3_split	when expanding store with update operations, expand into 2 instead of 3 operations	expand into 3
-g [2|3]	flag_use_group	enable alternative mechanisms for grouping instructions	1
(internal use)	flag_wide_arch_reg	(internal use)	disabled
-aF N	float_load_avail	additional cycles for data from floating-point load operation to be available, beyond those of an integer load operation	0
-lF N	float_load_latency	additional cycles for latency of floating-point load operations	0
-af N	fpu_avail	number of cycles for result from floating-point operation to be available (other than divide, square-root)	3
-lf N	fpu_latency	latency of floating-point instructions (other than divide, square-root,...)	3
-uf k	fpu_unit	unit where floating-point operations are performed	1
-ad N	idiv_avail	number of cycles for result from integer divide operation to be available	8
-ld N	idiv_latency	latency of integer divide instructions	8
-uX k	cmplx_unit	unit where complex operations are performed	0
(internal use)	idiv_unit	unit where integer divide operations are performed; the value of this parameter is inferred from icomplex_unit	0
-am N	imul_avail	number of cycles for result from integer multiply operation to be available	5
-lm N	imul_latency	latency of integer multiply instruction	5
(internal use)	imul_unit	unit where integer multiply operations are performed; the value of this parameter is inferred from icomplex_unit	0
-al N	load_avail	number of cycles for result from integer load operation to be available	3
-ll N	load_latency	latency of load instruction	3
-ul k	load_unit	unit where load operations are performed	2
-o foo	output_file	name of output file	
-p	pred_do	enable predicated execution	disabled
-pdist k	pred_dist	predication distance	1
-pmode		predication mode	0
-as N	store_avail	number of cycles for result from store operation to be available	3
-ls N	store_latency	latency of store instruction	3
-ae N	subword_load_avail	additional cycles for data from subword load operation to be available, beyond those of a regular (word-long) load operation	0
-le N	subword_load_latency	additional cycles for latency of subword integer load operations	0
-us k	store_unit	unit where store operations are performed	2