RC 21175
Computer Sciences/Mathematics

IBM Research Report

Techniques for Implementing Fast Processor Simulators

Mayan Moudgill
mayan@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden | Austin | China | Haifa | Tokyo | T.J. Watson | Zurich

Techniques for Implementing Fast Processor Simulators

Mayan Moudgill
mayan@watson.ibm.com

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

In this paper we describe techniques that enable the implementation of a fast processor simulator. These techniques have been used to implement a detailed out-of-order processor simulator called Turandot that executes over 350 million instructions per hour.

1. Introduction

Designing a processor involves making design choices between a number of possible microarchitectural and implementation features. In the case of an out-of-order wide-issue superscalar processor, the trade-offs become difficult, both because there are many more design choices to be made, and because many of these choices are sometimes counter-intuitive. It becomes necessary to model the various options and then measure the performance actually obtained before a particular option is chosen.

This approach to processor design requires a simulator that is detailed, flexible, and fast. The simulator must be detailed enough that the design choices made using the simulator would be unaffected by the details not modeled. The simulator must be flexible so that a large part of the design space can be searched. Lastly, the simulator must be fast, so that the numbers required to make a decision can be turned around quickly, a large part of the design space can be searched, and a large number of inputs can be used.

In this paper we describe the techniques used to implement a detailed out-of-order wide-issue superscalar simulator called Turandot. In particular, we shall focus on those techniques that allow Turandot to execute well over 350 million instructions per hour[*] - in many cases, Turandot executes over 400 million instructions per hour.

It is difficult to compare the performance of processor simulators, since performance is directly related to the complexity of the processor being modeled, and the detail at which it is modeled. For instance, the next fastest commercial processor simulator for which simulator performance numbers have been reported that we are aware of is ADAPT [2]. ADAPT has less than half the performance of Turandot, though it is modeling a simpler processor. The fastest academic simulator that we are aware of is the out-of-order processor simulator of the SimpleScalar toolset, sim-outorder [1]. sim-outorder has about twice the performance of Turandot, but is simulating a considerably simpler processor model.

The paper is organized as follows: we first give an overview of the capabilities of Turandot, so as to give an appreciation of the complexities of the processor model. Then we describe the techniques used to obtain the performance of Turandot. This is done in several parts:

- Section 3 details the performance improvements made possible by the shift in processor design philosophy,
- Section 4 describes the benefits obtained from recompiling the simulation executable for each processor model,
- Section 5 concentrates on high level optimizations that are specific to processor simulators and, finally,
- Section 6 examines somewhat more generally applicable performance coding techniques.

We briefly show the time required to simulate various benchmarks using Turandot in Section 7 before concluding in Section 8.

[*] on a RS/6000 43P-140 workstation, with a 200MHz 604e.

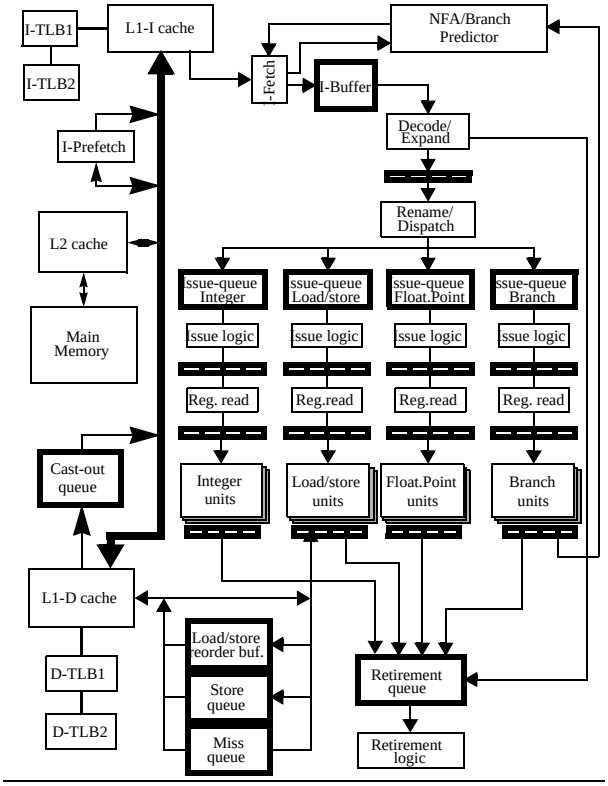

Figure 1: Typical processor model

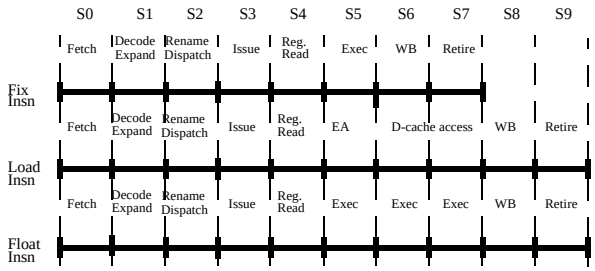

Figure 2: Typical processor pipeline

2. Turandot

Turandot is the simulator part of MET [3], a tool set designed to support the exploration of a large design space for out-of-order wide-issue PowerPC based superscalar processors.

Turandot is a cycle-driven simulator, coded using C. It is either trace-driven or execution-driven using Aria [6]. In the execution-driven mode, Turandot will also simulate instructions from the not-taken path.

A detailed description of Turandot can be found in [5]. In this section, we only outline both its flexibility and its complexity, so that one can appreciate how successful the techniques described in this paper have been in increasing the speed of the simulator.

To give an idea of its flexibility, here is a list of some of the features it supports:

- two levels of caches with 1, 2, or 4 way associativity, either separate I&D or unified.
- two levels of TLBs, with 1, 2, or 4 way associativity, either separate I&D or unified.
- I-cache prefetch.
- multi-ported and interleaved D-caches.
- L1/L2 bus features, including multi-cycle transfers, critical word first transfers, and trailing edge effects.
- dynamic load-over-store speculation, with alias detection and recovery.
- register renaming for 4 register files: condition register fields, general purpose registers, floating point registers, and special purpose registers.
- upto four kinds of units and issue queues: fixed-point, floating point, memory, and condition/branch.
- clustered fix and/or memory queues/units.
- next-fetch-address (NFA) logic for predicting next I-fetch address.
- various branch prediction algorithms for predicting branch direction/target.
- dynamic decomposition of complex instructions into simpler internal micro-instructions. For instance, a **lswx** (load-string word indexed) instruction is broken into several instructions, each of which only load part of the string. Even instructions as simple as **stwu** (store-word and update) can be broken up into two micro-instructions: a store-word and an update.
- varying number of pipeline stages.
- varying sizes of the various queues/buffers.

Fig. 1 shows the micro-architectural features of a typical design-point, while Fig. 2 shows the pipeline stages implemented. This is the design point for which we quote our simulator performance numbers. As can be seen, it is fairly complex. Some of the choices made in the model are:

- 2-way set associative separate I&D caches backed up by a unified 4-way set associative L2 cache.
- two levels of separate I&D side TLBs, each side consisting of a direct mapped I-side TLB backed up by a

unified 2-way associative second level TLB.

- branch prediction using a link stack for link-register branches, a branch target address cache for counter branches and a 2-level branch history table.
- expand logic to break up all instructions that write more than 1 general purpose register and/or read more than 2 general purpose registers.
- register renaming that separately renames general purpose, floating-point, condition register field and special purpose registers.
- two each of fix, memory, float and branch/conditional units, each with its own issue queue.
- load/store reorder logic that allows loads to be executed before stores, and later re-executed if some load and store overlapped.
- castout queues & miss queues for dealing with cache replacement and misses.
- an L1/L2 bus with a 3 cycle transfer time, which returns data in 2 blocks, critical block first.

3. Design-based optimizations

We have found that high frequency processors (500MHz+) lead to simpler and faster simulators. This is a consequence of the design style required to achieve high frequencies; in particular, the processor designers have to develop simpler pipelines, with less complexity per stage. This necessitates designs with low control complexity, fewer places where the processor can stall, as little arbitration as possible, etc. Such design practices, in turn, result in a simulator that is both simpler and faster.

3.1 Simple simulation loop

In the hardware we are modeling, computation occurs in parallel; various latches are simultaneously read at the beginning of a processor cycle, and then, in parallel, written to at the end of the cycle. To properly model the processor, the simulator must ensure that the variables representing such latches are all read before being written during a cycle. Ensuring that this happens becomes difficult, if not impossible, when dealing with complex control flow. The usual, and computationally expensive solution, is to model such registers using a *two-list algorithm* (also known as master/slave latches). This approach uses two variables per register - one of the variables is read from, and the other written to in the course of a cycle. At the end of the cycle, the value in the written-to variable is copied to the read-from variable.

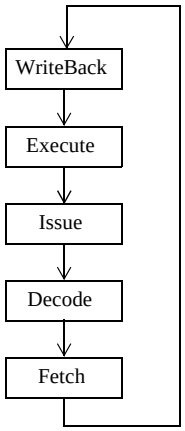

Figure 3: Simulation loop

The reduced control complexity of high-frequency processors makes it feasible to dispense with master/slave latches. Instead, we can in general ensure that all values from the previous cycle are read-from before being written-to by evaluating all stages of the pipeline in reverse order of data flow. For the case of a simple five stage pipeline, this would result in an evaluation loop as shown in Fig. 3. The resulting code is considerably faster since it avoids the over-heads introduced by the management of two variables in the two-list algorithm.

3.2 Collapsed stages

In high-frequency processors, the number of stages at which an instruction can stall has been reduced. Once an instruction passes a stage where the pipeline can be stalled, it can typically proceed for several pipeline stages before encountering the next pipeline stage at which it can be stalled. This allows the simulator to model the effect of executing several stages in one place, thereby decreasing the amount of work the simulator has to do.

As an example, in the processors we model, once a fixed-point instruction is issued from the issue-queue it is guaranteed not to stall till it enters the retirement stage. Thus, in the simulation, once it is known that an instruction is going to be issued, all the “future” actions associated with that instruction can be evaluated immediately, and the relevant structures can be updated. These actions include:

- Marking the instruction as being able to retire after **R** + **E** + **W** cycles, where **R** is the number of register access stages, **E** is the number of execution stages, and **W** the number of write-back stages.

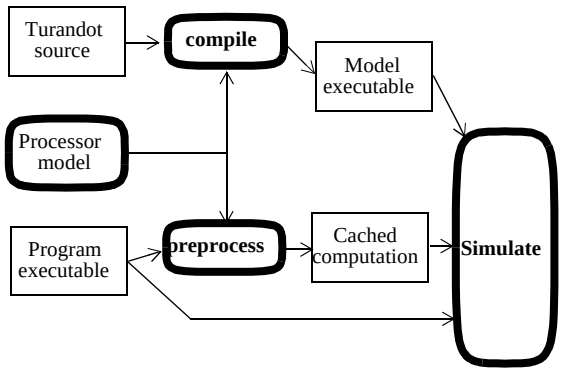

Figure 4: Simulation process in Turandot

- Marking its output as becoming available to instructions that need it after **E** cycles.

This allows the simulator to fold together the modeling of register fetch, execute and write-back into the issue stage.

4. Tailored models

Simulating a processor model under Turandot involves the following steps, as shown in Fig. 4:

- define the processor model (usually involves picking values for various constants and selecting between microarchitectural features).
- compile Turandot for that model.
- preprocess the executable program to be simulated against the model; this step precomputes and caches much of the information required by the simulator. See Section 4.2 for details.
- simulate the program using the compiled model executable and the preprocessed information.

At first glance, this approach appears to add a significant amount of overhead to the simulation process: about 10 minutes for the compile step[*]. However, we typically simulate billions of instructions against each model. The 10 minutes of compilation time is insignificant compared to the hours spent actually using the model. And, as we shall show later in this section, the speedup in the simulation time more than makes up for the extra compilation overhead.

4.1 Compile-time flags

Many simulators with a wide range of features use run-time parameters to specify the exact model being simulated. These parameters are usually used for things such as:

- Feature selection: which microarchitectural features will be modeled. This would include choosing to use/ not use instruction prefetching, or having a unified vs. separate second-level TLBs, and so on.
- Sizing: how big the various architectural features will be. For instance, the number of rename registers or cache line sizes would be controlled by sizing parameters.
- Output control: which statistics will be printed, how often, etc.

We replace each of these with compile-time parameters with a **#define** constant[†]. This has the drawback of forcing us to create a different executable for each set of processor model parameters. However this approach yields much higher performance for several reasons:

- **if(...) ...** statements that are used to control the simulation of a particular feature are replaced by **#if(...) ... #endif** preprocessor directives. This avoids the cost of the extra branches.
- It allows us to statically allocate various data structures, based on the value of #define constants.
- We can pick algorithms and structures based on the various **#define** constants; for instance, different cache simulation algorithms are used for 1, 2 and 4 way associativity.
- The use of **#define** constants allow expressions involving parameters to be evaluated at compile time.
- The compiler can use the immediate form of various machine instructions, instead of first loading the parameter and then computing the instruction.
- Only those statistics that are actually going to be printed are collected; since statistics gathering is a large component of the simulation cost, this can be quite important

[*] The overhead of preprocessing is negligible.

[†] More often, through -D<parameter>=<value> options to the compiler.

4.2 Caching

For each instruction in the program, any datum that is needed by the simulation and can be deduced from either the instruction word or its address is computed exactly once, and then cached. This includes information such as:

- number of registers used
- execution latency
- micro-instructions the instruction expands into
- the maximum number of instructions fetched in a block beginning at this address

As far as possible this computation is performed statically before the simulation. For AIX program executables, this includes all code other than the shared libraries. The shared libraries are translated dynamically. In both cases, the computed information is cached in a table that is indexed by the instruction address. Then, instead of recomputing any information for an instruction, it is obtained by reading from the table.

We make a further optimization - when an instruction is fetched for execution, much of the cached information for that instruction is copied into the in-flight instruction data structure. This avoids the need for repeated indexing into the cache table.

5. High-level optimizations

In this section, we shall describe those performance improvements obtained from techniques that apply predominantly to processor simulators.

5.1 Crystal ball

In a trace and execution-based simulation, we have a “crystal ball”; i.e., we know information about what will happen to the instruction at the time the instruction is fetched. For instance

- For branches, we can tell whether the branch is mispredicted or not.
- In the case of an instruction such as **lswx** (load string indexed), which moves a variable amount of data, we know at fetch time how much data it will move.

This permits several optimizations. For instance, in the case of a branch, we need to back up state to recover the in-order state in the case the branch is mispredicted. From the “crystal ball” we can determine at fetch time whether a branch is mispredicted or not, and whether the branch will need recovery or not. We can use this information to back up state only when necessary, instead of backing up the state on every branch for mispredicted branch recovery purposes.

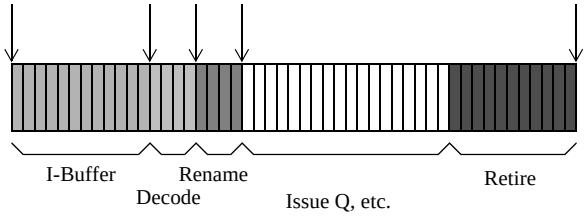

Figure 5: Sub-queues in in-flight queue

5.2 Single queue

All in-flight instructions, and the data pertinent to them, are maintained in a single queue. There is no copying of data (or pointers to data) as an instruction proceeds through the pipeline.

Instead, in-order queues such as the instruction-fetch buffer, the retirement queue or the various decode latches are maintained as sub-queues of this queue by “pointers” to the beginning and end of each sub-queue in the queue, as illustrated in Fig. 5.

For those queues that need to be maintained separately (such as the various issue-queues), space is allocated in the queue entry so that the instructions can be chained together

5.3 .Pseudo-event scheduling

Turandot is a cycle-driven timer. We do not have any event queues. However, in certain situations where events occur infrequently or unpredictably an event queues-like structure make sense. For instance, interactions between the various levels of the memory hierarchy fit into this category.

We optimize the execution time of such queues by maintaining the earliest cycle on which the next event will occur. This allows us to ignore the queue most of the time by simply checking if the current cycle equals the current time, and skipping if it does not. On the rare occasions where we do not skip, we then perform the extra work to see if an event actually needs processing.

5.4 Set-associative cache algorithm

We use different algorithms for simulating 1, 2 and 4 way set-associative caches. The details of the 2 and 4 way algorithms are described in [4]. Briefly, by changing the way the cache directories are probed, and the way LRU information is maintained, we can simulate set-associative caches with performance approaching that of simulating a direct mapped cache.

6. Low-level optimization

In this section, we describe those performance improvements obtained from techniques that can be applied to speed up programs in general.

6.1 Memory allocation

We do not use dynamic memory allocation at all. Instead, we use statically declared arrays of structures. This is made possible by the fact that the code is written such that, given the **#define** values, it is possible to compute the maximum number of each data-type used.

Using arrays of structures also has another benefit; instead of using pointers, we can now use offsets into the array of the relevant type. This has several benefits, including better anti-aliasing and lowered memory usage.

6.2 Optimize for cache locality

We work fairly aggressively on reducing the number of data cache misses. We achieve this principally by reducing the size of the working set.

We read (or produce) the instruction trace in small chunks into a buffer. We have found that, while a large buffer would permit us to amortize the cost of the trace read over a much larger number of instructions read per call to the trace reader, reading the large number of instructions wipes out the cache. It is more efficient to use a small buffer (say, around 16 instructions) and decrease the number of cache misses than to use a large buffer and decrease the number of calls to the trace reader.

Additionally, the various data-structures are optimized for space. This includes packing data as densely as possible by using the minimum number of bits wherever possible. For instance, if the maximum number of instructions in flight is less than 256, we use a single byte; otherwise we use 2 bytes.

Also, as mentioned above, instead of storing a pointer to a structure, we store the offset within the statically declared array of those structures. This usually allows us to use 1 or 2 byte offsets, as opposed to the 4 bytes required for storing a full pointer.

The use of **#define** constants allow us to pick data sizes that support the current options, not the worst case/ For instance, assume that we wanted to support upto 256 rename registers, but for a particular model we need only 128. In that case, we would allocate space only for 128 rename registers, not 256.

6.3 Alias avoidance

Since the code is written in a high-level language, it must be compiled; for best performance, we must make our source code such that it is as easy as possible for the compiler to generate good object code. One of the biggest hindrances to the ability of the compiler to generate good code is the presence of aliases. Consequently, as far as possible, we try to avoid any feature that could interfere with the compilers ability to distinguish between two addresses.

In particular, we have no pointers. Instead, as we have mentioned, all memory is statically allocated. As a matter of fact, we go even further. All the simulation code resides in a single procedure, and all variables are auto (i.e. stack-allocated local) variables. This, coupled with the fact that we are using strict ANSI C, allows the compiler to do an almost perfect job of anti-aliasing.

6.4 Load over branch movement

In general, a compiler cannot reschedule a load operation in the then or else sides of an **if(...)** to execute before the **if(...)** is evaluated. This constraint arises from the fact that the compiler cannot in general determine whether the address being loaded from was in fact a valid address; moving the load before the **if** might result in a segment fault in certain situations. However, it is often beneficial to do this.

In general, we should explicitly code so that the load is performed before the if when the following conditions hold:

- the load is of an array element*,
- it is safe to perform the load before the **if**,
- there is not much code above the **if**, or not much code in the block containing the load.

* or a pointer dereference

Benchmark	Instructions	Cycles/hour	Instructions/hour	Cycles/sec.	Instructions/sec.
compress	34.4 M	259 M	489 M	72 K	136 K
gcc	1212.1 M	259 M	371 M	72 K	103 K
go	42.9 M	275 M	372 M	76 K	103 K
ijpeg	1132.4 M	285 M	590 M	79 K	163 K
li	191.4 M	243 M	460 M	67 K	127 K
m88ksim	117.4 M	260 M	444 M	72 K	123 K
perl	1.2 M	243 M	345 M	67 K	95 K
vortex	2527.1 M	211 M	482 M	58 K	134 K

Figure 6: Performance of Turandot

6.5 Branch removal

Branches hurt processor performance in many ways. A general rule of thumb is that it is always good to reduce the number of branches wherever possible. There are several ways of doing this of which we shall describe two.

First, we can avoid branches by converting the control dependence into an expression evaluation. For instance,

```
if( a && b ) {
  x ++;
}
```

can be converted into the semantically equivalent

```
x += a && b;
```

thereby avoiding a branch.*

The other approach to avoiding branches is to guard several unlikely branches by a single branch which is true if and only if one of the other branches is true. For example, taking a case that occurs in branch prediction simulation:

```
if( branch_cond ) {
  /*** use branch history table ***/
}
else if( branch_through_lr ) {
  /*** use link stack for prediction ***/
}
else if( branch_through_ctr ) {
  /*** use branch-target address cache ***/
}
```

This can be optimized by re-writing as follows:

```
if( branch_of_some_kind ) {
  if( branch_cond ) {
    /*** use branch history table ***/
  }
  else if( branch_through_lr ) {
    /*** use link stack for prediction ***/
  }
  else if( branch_through_ctr ) {
    /*** use branch-target address cache ***/
  }
}
```

Now, assume branches are about 20% of all instructions. Thus, in the original code, 80% of the time we would be executing 3 branches, all of which would be not taken. In the re-written code, we execute exactly one branch for non-branch instructions, and we execute one extra branch for the remaining 20% of instructions that are branches. This results in a considerable reduction in the total number of branches taken (about 1.4 branches per iteration).

7. Performance

The run-time performance of Turandot simulating various programs in execution-driven mode is shown in Fig. 6. The processor model used was the typical model described in Section 2. The programs picked are from the SPECint95 suite, compiled using **xlc** with optimization level **-O2**. The input used for each program was one of the provided training inputs. The programs were run to completion. The numbers were collected on a RS/6000 43P-140 workstation, with a 200MHz PowerPC 604e processor.

As can be seen, Turandot can simulate the execution of between 345 to 590 million instructions per hour. Alternatively, Turandot simulates between 211 and 286 million cycles per hour. By either measure, the performance is quite respectable.

* There are compilers which will automatically do this transformation; however, they will not capture all opportunities.

8. Conclusion

In this paper, we show that it is possible to implement a detailed and flexible simulator for high-frequency out-of-order wide-issue superscalar processors with fairly high run-time performance.

We have outlined many of the techniques we have used to achieve high performance. These range from techniques that take advantage of the increasing simplicity of processor control logic to various coding techniques that are more generally applicable. Perhaps the most novel of these techniques is the one described in Section 4.2 - precomputing and caching information associated with an instruction word and address.

We have used these techniques to implement a detailed flexible simulator, Turandot, that can simulate over 350 million instructions per hour. Turandot has made it possible for us to perform in a reasonable amount of time studies that require the simulation hundreds of different processor configurations for billions of cycles.

9. Acknowledgments

We would like to thank the other members of the MET team - Jaime Moreno, Pradip Bose, J.D. Wellman, and Louise Trevillyan for their assistance in developing, debugging and validating Turandot.

10. References

[1] T. Austin. and D. Burger, “MICRO-30 Simple Scalar Tutorial”, held in conjunction with 30th Annual Symposium on Microarchitecture,1997.

[2] B.G. Burgess and S.P. Litch, “Advanced Alterable Pipeline Timer (ADAPT): A Tool to Design a High-Performance PowerPC Microprocessor,” IEEE International Performance, Computing and Communications Conference, 1997, pp. 271-276.

[3] J. Moreno, M. Moudgill, J.D. Wellman and P. Bose, “The MET: A Microarchitecture Exploration Toolset,” Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[4] M. Moudgill, “Techniques for fast simulation of associative cache directories,” Research Report RC21038,IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1997.

[5] M. Moudgill, J. Moreno, P. Bose and J.D. Wellman, “Turandot: A PowerPC-based wide-issue superscalar processor model for microarchitecture exploration,” Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[6] J.D. Wellman and M. Moudgill, “Aria: an execution simulation environment for microarchitectural analyses,” Research Report (in preparation), IBM Thomas J. Watson Research Center, Yorktown Heights, NY.