# PTLsim User's Guide and Reference

*The Anatomy of an x86-64 Out of Order Superscalar Microprocessor*

Matt T. Yourst
`<yourst@yourst.com>`

Revision 20070923
***Second Edition***

The latest version of PTLsim and this document are always available at:

**www.ptlsim.org**

# Contents

# Part I

# PTLsim User’s Guide

# Chapter 1

# Introducing PTLsim

## 1.1 Introducing PTLsim

**PTLsim** is a state of the art cycle accurate microprocessor simulator and virtual machine for the x86 and x86-64 instruction sets. PTLsim models a modern superscalar out of order x86-64 compatible processor core at a configurable level of detail ranging from full-speed native execution on the host CPU all the way down to RTL level models of all key pipeline structures. In addition, the complete cache hierarchy, memory subsystem and supporting hardware devices are modeled with true cycle accuracy. PTLsim supports the full x86-64 instruction set of the Pentium 4+, Athlon 64 and similar machines with all extensions (x86-64, SSE/SSE2/SSE3, MMX, x87). It is currently the only tool available to the public to support true cycle accurate modeling of real x86 microarchitectures.

PTLsim is very different from most cycle accurate simulators. Because it runs directly on the same platform it is simulating (an x86 or x86-64 machine, typically running Linux), it is able to switch in and out of full out of order simulation mode and native x86 or x86-64 mode at any time completely transparent to the running user code. This lets users quickly profile a small section of the user code without the overhead of emulating the uninteresting parts, and enables automatic debugging by finding the divergence point between a real reference machine and the simulation.

PTLsim comes in two flavors. The classic version runs any 32-bit or 64-bit single threaded userspace Linux application. We have successfully run a wide array of programs under PTLsim, from typical benchmarks to graphical applications and network servers.

PTLsim/X runs on the bare hardware and integrates with Xen hypervisor, allowing it to provide full system x86-64 simulation, multi-processor and multi-threading support (SMT and multi-core models), checkpoints, cycle accurate virtual device timing models, deterministic time dilation, and much more, all without sacrificing the speed and accuracy inherent in PTLsim's design. PTLsim/X makes it possible to run any Xen-compatible operating system under simulation; we have successfully booted arbitrary Linux distributions and industry standard applications and benchmarks under PTLsim/X.

Compared to competing simulators, PTLsim provides extremely high performance even when running in full cycle accurate out of order simulation mode. Through extensive tuning, cache profiling and the use of x86 specific accelerated vector operations and instructions, PTLsim significantly cuts simulation time compared to traditional research simulators. Even with its optimized core, PTLsim still allows a significant amount of flexibility for easy experimentation through the use of optimized C++ template classes and libraries suited to synchronous logic design.

## 1.2 History

PTLsim was designed and developed by Matt T. Yourst `<yourst@yourst.com>` with its beginnings dating back to 2001. The main PTLsim code base, including the out of order processor model, has been in active development since 2003 and has been used extensively by our processor design research group at the State University of New York at Binghamton in addition to hundreds of major universities, industry research labs and several well known microprocessor vendors.

PTLsim is not related to other legacy simulators. It is our hope that PTLsim will help microprocessor researchers move to a contemporary and widely used instruction set (x86 and x86-64) with readily available hardware implementations. This will provide a new option for researchers stuck with simulation tools supporting only the Alpha or MIPS based instruction sets, both of which have since been discontinued on real commercially available hardware (making co-simulation impossible) with an uncertain future in up to date compiler toolchains.

The PTLsim software and this manual are free software, licensed under the GNU General Public License version 2.

# Chapter 2

# Getting Started

## 2.1 Documentation Map

This manual has been divided into several parts:

- Part I introduces PTLsim, reviews the x86 architecture, and describes PTLsim's implementation of x86 in terms of uops, microcode and internal structures.
- Part II describes the use and implementation of userspace PTLsim.
  - If you simply want to *use* PTLsim, this part starts with an easy to follow **tutorial**
- Part III describes the use and implementation of full system PTLsim/X.
  - If you simply want to *use* full system PTLsim/X, this part starts with an easy to follow **tutorial**
- Part IV details the design and implementation of the PTLsim out of order superscalar core model
  - Read this part if you want to understand and modify PTLsim's out of order core.
- Part V is a reference manual for the PTLsim internal uop instruction set, the performance monitoring events the simulator supports and a variety of other technical information.

## 2.2 Additional Resources

The latest version of PTLsim and this document are always available at the PTLsim web site:

**http://www.ptlsim.org**

# Chapter 3

# PTLsim Architecture

# Chapter 4

# PTLsim Code Base

## 4.1 Code Base Overview

PTLsim is written in C++ with extensive use of x86 and x86-64 inline assembly code. It must be compiled with gcc on a Linux 2.6 based x86 or x86-64 machine. The C++ variant used by PTLsim is known as Embedded C++. Essentially, we only use the features found in C, but add templates, classes and operator overloading. Other C++ features such as hidden side effects in constructors, exception handling, RTTI, multiple inheritance, virtual methods (in most cases), thread local storage and so on are forbidden since they cannot be adequately controlled in the embedded "bare hardware" environment in which PTLsim runs, and can result in poor performance. We have our own standard template library, SuperSTL, that must be used in place of the C++ STL.

Even though the PTLsim code base is very large, it is well organized and structured for extensibility. The following section is an overview of the source files and subsystems in PTLsim:

- **PTLsim Core Subsystems:**
  - `ptlsim.cpp` and `ptlsim.h` are responsible for general top-level PTLsim tasks and starting the appropriate simulation core code.
  - `uopimpl.cpp` contains implementations of all uops and their variations. PTLsim implements most ALU and floating point uops in assembly language so as to leverage the exact semantics and flags generated by real x86 instructions, since most PTLsim uops are so similar to the equivalent x86 instructions. When compiled on a 32-bit system, some of the 64-bit uops must be emulated using slower C++ code.
  - `ptlhwdef.cpp` and `ptlhwdef.h` define the basic uop encodings, flags and registers. The tables of uops might be interesting to see how a modern x86 processor is designed at the microcode level. The basic format is discussed in Section 5.1; all uops are documented in Section 27.
  - `seqcore.cpp` implements the sequential in-order core. This is a strictly functional core, without data caches, branch prediction and so forth. Its purpose is to provide fast execution of the raw uop stream and debugging of issues with the decoder, microcode or virtual hardware rather than a specific core model.

- **Decoder, Microcode and Basic Block Cache:**
  - `decode-core.cpp` coordinates the translation from x86 and x86-64 into uops, maintains the basic block cache and handles self modifying code, invalidation and other x86 specific complexities.
  - `decode-fast.cpp` decodes the subset of the x86 instruction set used by 95% of all instructions with four or fewer uops. It should be considered the "fast path" decoder in a hardware microprocessor.
  - `decode-complex.cpp` decodes complex instructions into microcode, and provides most of the assists (microcode subroutines) required by x86 machines.
  - `decode-sse.cpp` decodes all SSE, SSE2, SSE3 and MMX instructions
  - `decode-x87.cpp` decodes x87 floating point instructions and provides the associated microcode
  - `decode.h` contains definitions of the above functions and classes.
- **Out Of Order Core:**
  - `ooocore.cpp` is the out of order simulator control logic. The microarchitectural model implemented by this simulator is the subject of Part IV.
  - `ooopipe.cpp` implements the discrete pipeline stages (frontend and backend) of the out of order model.
  - `oooexec.cpp` implements all functional units, load/store units and issue queue and replay logic
  - `ooocore.h` defines most of the configurable parameters for the out of order core not intrinsic to the PTLsim uop instruction set itself.
  - `dcache.cpp` and `dcache.h` contain the data cache model. At present the full L1/L2/L3/mem hierarchy is modeled, along with miss buffers, load fill request queues, ITLB/DTLB and bus interfaces. The cache hierarchy is very flexible configuration wise; it is described further in Section 25.
  - `branchpred.cpp` and `branchpred.h` is the branch predictor. By default, this is set up as a hybrid bimodal and history based predictor with various customizable parameters.
- **Linux Hosted Kernel Interface:**
  - `kernel.cpp` and `kernel.h` is where all the virtual machine "black magic" takes place to let PTLsim transparently switch between simulation and native mode and 32-bit/64-bit mode (or only 32-bit mode on a 32-bit x86 machine). In general you should not need to touch this since it is very Linux kernel specific and works at a level below the standard C/C++ libraries.
  - `lowlevel-64bit.S` contains 64-bit startup and context switching code. PTLsim execution starts here if run on an x86-64 system.

- `lowlevel-32bit.S` contains 32-bit startup and context switching code. PTLsim execution starts here if run on a 32-bit x86 system.
- `injectcode.cpp` is compiled into the 32-bit and 64-bit code injected into the target process to map the `ptlsim` binary and pass control to it.
- `loader.h` is used to pass information to the injected boot code.

• **PTLsim/X Bare Hardware and Xen Interface:**

  - `ptlxen.cpp` brings up PTLsim on the bare hardware, dispatches traps and interrupts, virtualizes Xen hypercalls, communicates via DMA with the PTLsim monitor process running in the host domain 0 and otherwise serves as the kernel of PTLsim's own mini operating system.
  - `ptlxen-memory.cpp` is responsible for all page based memory operations within PTLsim. It manages PTLsim's own internal page tables and its physical memory map, and services page table walks, parts of the x86 microcode and memory-related Xen hypercalls.
  - `ptlxen-events.cpp` provides all interrupt (VIRQ) and event handling, manages PTLsim's time dilation technology, and provides all time and event related hypercalls.
  - `ptlxen-common.cpp` provides common functions used by both PTLsim itself and PTLmon.
  - `ptlxen.h` provides inline functions and defines related to full system PTLsim/X.
  - `ptlmon.cpp` provides the PTLsim monitor process, which runs in domain 0 and interfaces with the PTLsim hypervisor code inside the target domain to allow it to communicate with the outside world. It uses a client/server architecture to forward control commands to PTLsim using DMA and Xen hypercalls.
  - `xen-types.h` contains Xen-specific type definitions
  - `ptlsim-xen-hypervisor.diff` and `ptlsim-xen-tools.diff` are patches that must be applied to the Xen hypervisor source tree and the Xen userspace tools, respectively, to allow PTLsim to be injected into domains.
  - `ptlxen.lds` and `ptlmon.lds` are linker scripts used to lay out the memory image of PTLsim and PTLmon.
  - `lowlevel-64bit-xen.S` contains the PTLsim/X boot code, interrupt handling and exception handling
  - `ptlctl.cpp` is a utility used within a domain under simulation to control PTLsim
  - `ptlcalls.h` provides a library of functions used by code within the target domain to control PTLsim.

• **Support Subsystems:**

  - `superstl.h`, `superstl.cpp` and `globals.h` implement various standard library functions and classes as an alternative to C++ STL. These libraries also contain a number of features very useful for bit manipulation.

  - `logic.h` is a library of C++ templates for implementing synchronous logic structures like associative arrays, queues, register files, etc. It has some very clever features like `FullyAssociativeArray8bit`, which uses x86 SSE vector instructions to associatively match and process ~16 byte-sized tags every cycle. These classes are fully parameterized and useful for all kinds of simulations.
  - `mm.cpp` is the PTLsim custom memory manager. It provides extremely fast memory allocation functions based on multi-threaded slab caching (the same technique used inside Linux itself) and extent allocation, along with a traditional physical page allocator. The memory manager also provides PTLsim's garbage collection system, used to discard unused or least recently used objects when allocations fail.
  - `mathlib.cpp` and `mathlib.h` provide standard floating point functions suitable for embedded systems use. These are used heavily as part of the x87 microcode.
  - `klibc.cpp` and `klibc.h` provide standard libc-like library functions suitable for use on the bare hardware
  - `syscalls.cpp` and `syscalls.h` declare all Linux system call stubs. This is also used by PTLsim/X, which emulates some Linux system calls to make porting easier.
  - `config.cpp` and `config.h` manage the parsing of configuration options for each user program. This is a general purpose library used by both PTLsim itself and the userspace tools (PTLstats, etc)
  - `datastore.cpp` and `datastore.h` manage the PTLsim statistics data store file structure.

- **Userspace Tools:**
  - `ptlstats.cpp` is a utility for printing and analyzing the statistics data store files in various human readable ways.
  - `dstbuild` is a Perl script used to parse stats.h and generate the datastore template (Section 8)
  - `makeusage.cpp` is used to capture the usage text (help screen) for linking into PTLsim
  - `cpuid.cpp` is a utility program to show various data returned by the x86 `cpuid` instruction. Run it under PTLsim for a surprise.
  - `glibc.cpp` contains miscellaneous userspace functions
  - `ptlcalls.c` and `ptlcalls.h` are optionally compiled into user programs to let them switch into and out of simulation mode on their own. The `ptlcalls.o` file is typically linked with Fortran programs that can't use regular C header files.

## 4.2 Common Libraries and Logic Design APIs

PTLsim includes a number of powerful C++ templates, macros and functions not found anywhere else. This section attempts to provide an overview of these structures so that users of PTLsim will use them instead of trying to duplicate work we've already done.

### 4.2.1 General Purpose Macros

The file `globals.h` contains a wide range of very useful definitions, functions and macros we have accumulated over the years, including:

- Basic data types used throughout PTLsim (e.g. `W64` for 64-bit words, `Waddr` for words the same size as pointers, and so on)
- Type safe C++ template based functions, including `min`, `max`, `abs`, `mux`, etc.
- Iterator macros (`foreach`)
- Template based metaprogramming functions including `lengthof` (finds the length of any static array), `offsetof` (offset of member in structure), `baseof` (member to base of structure), and `log2` (takes the base-2 log of any constant at compile time)
- Floor, ceiling and masking functions for integers and powers of two (`floor`, `trunc`, `ceil`, `mask`, `floorptr`, `ceilptr`, `maskptr`, `signext`, etc)
- Bit manipulation macros (`bit`, `bitmask`, `bits`, `lowbits`, `setbit`, `clearbit`, `assignbit`). Note that the `bitvec` template (see below) should be used in place of these macros wherever it is more convenient.
- Comparison functions (`aligned`, `strequal`, `inrange`, `clipto`)
- Modulo arithmetic (`add_index_modulo`, `modulo_span`, et al)
- Definitions of basic x86 SSE vector functions (e.g. `x86_cpu_pcmpeqb` et al)
- Definitions of basic x86 assembly language functions (e.g. `x86_bsf64` et al)
- A full suite of bit scanning functions (`lsbindex`, `msbindex`, `popcount` et al)
- Miscellaneous functions (`arraycopy`, `setzero`, etc)

### 4.2.2 Super Standard Template Library (SuperSTL)

The Super Standard Template Library (SuperSTL) is an internal C++ library we use internally in lieu of the normal C++ STL for various technical and preferential reasons. While the full documentation is in the comments of `superstl.h` and `superstl.cpp`, the following is a brief list of its features:

- I/O stream classes familiar from Standard C++, including `istream` and `ostream`. Unique to SuperSTL is how the comma operator (“,”) can be used to separate a list of objects to send to or from a stream, in addition to the usual C++ insertion operator (“<<”).
- To read and write binary data, the `idstream` and `odstream` classes should be used instead.

- String buffer (`stringbuf`) class for composing strings in memory the same way they would be written to or read from an `ostream` or `istream`.
- String formatting classes (`intstring`, `hexstring`, `padstring`, `bitstring`, `bytemaskstring`, `floatstring`) provide a wrapper around objects to exercise greater control of how they are printed.
- Array (`array`) template class represents a fixed size array of objects. It is essentially a simple but very fast wrapper for a C-style array.
- Bit vector (`bitvec`) is a heavily optimized and rewritten version of the Standard C++ `bitset` class. It supports many additional operations well suited to logic design purposes and emphasizes extremely fast branch free code.
- Dynamic Array (`dynarray`) template class provides for dynamically sized arrays, stacks and other such structures, similar to the Standard C++ `valarray` class.
- Linked list node (`listlink`) template class forms the basis of double linked list structures in which a single pointer refers to the head of the list.
- Queue list node (`queuelink`) template class supports more operations than `listlink` and can serve as both a node in a list and a list head/tail header.
- Index reference (`indexref`) is a smart pointer which compresses a full pointer into an index into a specific structure (made unique by the template parameters). This class behaves exactly like a pointer when referenced, but takes up much less space and may be faster. The `indexrefnull` class adds support for storing null pointers, which `indexref` lacks.
- `Hashtable` class is a general purpose chaining based hash table with user configurable key hashing and management via add-on template classes.
- `SelfHashtable` class is an optimized hashtable for cases where objects contain their own keys. Its use is highly recommended instead of `Hashtable`.
- `ChunkList` class maintains a linked list of small data items, but packs many of these items into a chunk, then chains the chunks together. This is the most cache-friendly way of maintaining variable length lists.
- `CRC32` calculation class is useful for hashing
- `CycleTimer` is useful for timing intervals with sub-nanosecond precision using the CPU cycle counter (discussed in Section 11.5).

### 4.2.3 Logic Standard Template Library (LogicSTL)

The Logic Standard Template Library (LogicSTL) is an internally developed add-on to SuperSTL which supports a variety of structures useful for modeling sequential logic. Some of its primitives may look familiar to Verilog or VHDL programmers. While the full documentation is in the comments of `logic.h`, the following is a brief list of its features:

- `latch` template class works like any other assignable variable, but the new value only becomes visible after the `clock()` method is called (potentially from a global clock chain).
- `Queue` template class implements a general purpose fixed size queue. The queue supports various operations from both the head and the tail, and is ideal for modeling queues in microprocessors.
- Iterators for `Queue` objects such as `foreach_forward`, `foreach_forward_from`, `foreach_forward_after`, `foreach_backward`, `foreach_backward_from`, `foreach_backward_before`.
- `HistoryBuffer` maintains a shift register of values, which when combined with a hash function is useful for implementing predictor histories and the like.
- `FullyAssociativeTags` template class is a general purpose array of associative tags in which each tag must be unique. This class uses highly efficient matching logic and supports pseudo-LRU eviction, associative invalidation and direct indexing. It forms the basis for most associative structures in PTLsim.
- `FullyAssociativeArray` pairs a `FullyAssociativeTags` object with actual data values to form the basis of a cache.
- `AssociativeArray` divides a `FullyAssociativeArray` into sets. In effect, this class can provide a complete cache implementation for a processor.
- `LockableFullyAssociativeTags`, `LockableFullyAssociativeArray` and `LockableAssociativeArray` provide the same services as the classes above, but support locking lines into the cache.
- `CommitRollbackCache` leverages the `LockableFullyAssociativeArray` class to provide a cache structure with the ability to roll back all changes made to memory (not just within this object, but everywhere) after a checkpoint is made.
- `FullyAssociativeTags8bit` and `FullyAssociativeTags16bit` work just like `FullyAssociativeTags`, except that these classes are dramatically faster when using small 8-bit and 16-bit tags. This is possible through the clever use of x86 SSE vector instructions to associatively match and process 16 8-bit tags or 8 16-bit tags every cycle. In addition, these classes support features like removing an entry from the middle of the array while compacting entries around it in constant time. These classes should be used in place of `FullyAssociativeTags` whenever the tags are small enough (i.e. almost all tags except for memory addresses).
- `FullyAssociativeTagsNbitOneHot` is similar to `FullyAssociativeTagsNbit`, but the user must guarantee that all tags are unique. This property is used to perform extremely fast matching even with long tags (32+ bits). The tag data is striped across multiple SSE vectors and matched in parallel, then a clever adaptation of the sum-of-absolute-differences SSE instruction is used to extract the single matching element (if any) in O(1) time.

### 4.2.4 Miscellaneous Code

The out of order simulator, ooocore.h, contains several reusable classes, including:

- `IssueQueue` template class can be used to implement all kinds of broadcast based issue queues
- `StateList` and `ListOfStateLists` is useful for collecting various lists that objects can be on into one structure.

# Chapter 5

# x86 Instructions and Micro-Ops (uops)

## 5.1 Micro-Ops (uops) and TransOps

PTLsim presents to the target code a full implementation of the x86 and x86-64 instruction set (both 32-bit and 64-bit modes), including most user and kernel level instructions supported by the Intel Pentium 4 and AMD K8 microprocessors (i.e. all standard instructions, SSE/SSE2, x86-64 and most of x87 FP). At the present stage of development, the vast majority of all userspace and 32-bit/64-bit privileged instructions are supported.

The x86 instruction set is based on the two-operand CISC concept of load-and-compute and load-compute-store. However, all modern x86 processors (including PTLsim) do not directly execute complex x86 instructions. Instead, these processors translate each x86 instruction into a series of micro-operations (*uops*) very similar to classical load-store RISC instructions. Uops can be executed very efficiently on an out of order core, unlike x86 instructions. In PTLsim, uops have three source registers and one destination register. They may generate a 64-bit result and various x86 status flags, or may be loads, stores or branches.

The x86 instruction decoding process initially generates translated uops (*transops*), which have a slightly different structure than the true uops used in the processor core. Specifically, sources and destinations are represented as un-renamed architectural registers (or special temporary register numbers), and a variety of additional information is attached to each uop only needed during the renaming and retirement process. TransOps (represented by the `TransOp` structure) consist of the following:

- `som`: Start of Macro-Op. Since x86 instructions may consist of multiple transops, the first transop in the sequence has its `som` bit set to indicate this.
- `eom`: End of Macro-Op. This bit is set for the last transop in a given x86 instruction (which may also be the first uop for single-uop instructions)
- `bytes`: Number of bytes in the corresponding x86 instruction (1-15). The same `bytes` field value is present in all uops comprising an x86 instruction.

- `opcode`: the uop (not x86) opcode
- `size`: the effective operation size (0-3, for 1/2/4/8 bytes)
- `cond`: the x86 condition code for branches, selects, sets, etc. For loads and stores, this field is reused to specify unaligned access information as described later.
- `setflags`: subset of the x86 flags set by this uop (see Section 5.4)
- `internal`: set for certain microcode operations. For instance, loads and stores marked internal access on-chip registers or buffers invisible to x86 code (e.g. machine state registers, segmentation caches, floating point constant tables, etc).
- `rd`, `ra`, `rb`, `rc`: the architectural source and destination registers (see Section 18.4.1)
- `extshift`: shift amount (0-3 bits) used for shifted adds (x86 memory addressing and LEA). The `rc` operand is shifted left by this amount.
- `cachelevel`: used for prefetching and non-temporal loads and stores
- `rbimm` and `rcimm`: signed 64-bit immediates for the rb and rc operands. These are selected by specifying the special constant `REG_imm` in the `rb` and `rc` fields, respectively.
- `riptaken`: for branches only, the 64-bit target RIP of the branch if it were taken.
- `ripseq`: for branches only, the 64-bit sequential RIP of the branch if it were not taken.

Appendix 27 describes the semantics and encoding of all uops supported by the PTLsim processor model. The following is an overview of the common features of these uops and how they are used to synthesize specific x86 instructions.

## 5.2 Load-Execute-Store Operations

Simple integer and floating point operations are fairly straightforward to decode into loads, stores and ALU operations; a typical load-op-store ALU operation will consist of a load to fetch one operand, the ALU operation itself, and a store to write the result. The instruction set also implements a number of important but complex instructions with bizarre semantics; typically the translator will synthesize and inject into the uop stream up to 8 uops for more complex instructions.

## 5.3 Operation Sizes

Most x86-64 instructions can operate on 8, 16, 32 or 64 bits of a given register. For 8-bit and 16-bit operations, only the low 8 or 16 bits of the destination register are actually updated; 32-bit and 64-bit operations are zero extended as with RISC architectures. As a result, a dependency on the old destination register may be introduced so merging can be performed. Fortunately, since x86 features destructive overwrites of the destination register (i.e. the `rd` and `ra` operands are the same), the `ra` operand is generally already a dependency. Thus, the PTLsim uop encoding reserves 2 bits to specify the operation size; the low bits of the new result are automatically merged with the old destination value (in `ra`) as part of the ALU logic. This applies to the `mov` uop as well, allowing operations like "`mov al,bl`" in one uop. Loads do not support this mode, so loads into 8-bit and 16-bit registers must be followed by a separate `mov` uop to truncate and merge the loaded value into the old destination properly. Fortunately this is not necessary when the load-execute form is used with 8-bit and 16-bit operations.

The x86 ISA defines some bizarre byte operations as a carryover from the ancient 8086 architecture; for instance, it is possible to address the second byte of many integer registers as a separate register (i.e. as `ah`, `bh`, `ch`, `dh`). The `mask` uop is used for handling this rare but important set of operations.

## 5.4 Flags Management and Register Renaming

Many x86 arithmetic instructions modify some or all of the processor's numerous status and condition flag bits, but only 5 are relevant to normal execution: Zero, Parity, Sign, Overflow, Carry. In accordance with the well-known "ZAPS rule", any instruction that updates any of the Z/P/S flags updates all three flags, so in reality only three flag entities need to be tracked: ZPS, O, F ("ZAPS" also includes an Auxiliary flag not accessible by most modern user instructions; it is irrelevant to the discussion below).

The x86 flag update semantics can hamper out of order execution, so we use a simple and well known solution. The 5 flag bits are attached to each result and physical register (along with *invalid* and *waiting* bits used by some cores); these bits are then consumed along with the actual result value by any consumers that also need to access the flags. It should be noted that not all uops generate all the flags as well as a 64-bit result, and some uops only generate flags and no result data.

The register renaming mechanism is aware of these semantics, and tracks the latest x86 instruction in program order to update each set of flags (ZAPS, C, O); this allows branches and other flag consumers to directly access the result with the most recent program-ordered flag updates yet still allows full out of order scheduling. To do this, x86 processors maintain three separate rename table entries for the ZAPS, CF, OF flags in addition to the register rename table entry, any or all of which may be updated when uops are renamed. The `TransOp` structure for each uop has a 3-bit `setflags` field filled out during decoding in accordance with x86 semantics; the `SETFLAG_ZF`, `SETFLAG_CF`, `SETFLAG_OF` bits in this field are used to determine which of the ZPS, O, F flag subsets to rename.

As mentioned above, any consumer of the flags needs to consult at most three distinct sources: the last ZAPS producer, the Carry producer and the Overflow producer. This conveniently fits into PTLsim's three-operand uop semantics. Various special uops access the flags associated with an operand rather than the 64-bit operand data itself. Branches always take two flag sources, since in x86 this is enough to evaluate any possible condition code combination (the `cond_code_to_flag_regs` array provides this mapping).

Various ALU instructions consume only the flags part of a source physical register; these include `addc` (add with carry), `rcl/rcr` (rotate carry), `sel.cc` (select for conditional moves) and so on. Finally, the `collcc` uop takes three operands (the latest producer of the ZAPS, CF and OF flags) and merges the flag components of each operand into a single flag set as its result.

PTLsim also provides compound compare-and-branch uops (`br.sub.cc` and `br.and.cc`); these are currently used mostly in microcode, but a core could dynamically merge `CMP` or `TEST` and `Jcc` instructions into these uops; this is exactly what the Intel Core 2 and a few research processors already do.

## 5.5 x86-64

The 64-bit x86-64 instruction set is a fairly straightforward extension of the 32-bit IA-32 (x86) instruction set. The x86-64 ISA was introduced by AMD in 2000 with its K8 microarchitecture; the same instructions were subsequently plagiarized by Intel under a different name ("EM64T") several years later. In addition to extending all integer registers and ALU datapaths to 64 bits, x86-64 also provides a total of 16 integer general purpose registers and 16 SSE (vector floating and fixed point) registers. It also introduced several 64-bit address space simplifications, including RIP-relative addressing and corresponding new addressing modes, and eliminated a number of legacy features from 64-bit mode, including segmentation, BCD arithmetic, some byte register manipulation, etc. Limited forms of segmentation are still present to allow thread local storage and mark code segments as 64-bit. In general, the encoding of x86-64 and x86 are very similar, with 64-bit mode adding a one byte REX prefix to specify additional bits for source and destination register indexes and effective address size. As a result, both variants can be decoded by similar decoding logic into a common set of uops.

## 5.6 Unaligned Loads and Stores

Compared to RISC architectures, the x86 architecture is infamous for its relatively widespread use of unaligned memory operations; any implementation must efficiently handle this scenario. Fortunately, analysis shows that unaligned accesses are rarely in the performance intensive parts of a modern program (with the exception of certain media processing algorithms). Once a given load or store is known to frequently have an unaligned address, it can be preemptively split into two aligned loads or stores at decode time. PTLsim does this by initially causing all unaligned loads and stores to raise an `UnalignedAccess` internal exception, forcing a pipeline flush. At this point, the special `unaligned` bit is set for the problem load or store uop in its translated basic block representation. The next time the offending uop is encountered, it will be split into two parts very early in the pipeline.

PTLsim includes special uops to handle loads and stores split into two in this manner. The `ld.lo` uop rounds down its effective address $\lfloor A \rfloor$ to the nearest 64-bit boundary and performs the load. The `ld.hi` uop rounds up to $\lceil A+8 \rceil$, performs another load, then takes as its third rc operand the first (`ld.lo`) load's result. The two loads are concatenated into a 128-bit word and the final unaligned data is extracted. Stores are handled in a similar manner, with `st.lo` and `st.hi` rounding down and up to store parts of the unaligned value in adjacent 64-bit blocks. Depending on the core model, these unaligned load or store pairs access separate store buffers for each half as if they were independent.

## 5.7 Repeated String Operations

The x86 architecture allows for repeated string operations, including block moves, stores, compares and scans. The iteration count of these repeated operations depends on a combination of the `rcx` register and the flags set by the repeated operation (e.g. compare). To translate these instructions, PTLsim treats the `rep xxx` instruction as a single basic block; any basic block in progress before the repeat instruction is terminated and the repeat is decoded as a separate basic block. To handle the unusual case where the repeat count is zero, a check uop (see below) is inserted at the top of the loop to protect against this case; PTLsim simply bypasses the offending block if the check fails.

## 5.8 Checks and SkipBlocks

PTLsim includes special uops (`chk.and.cc`, `chk.sub.cc`) that compare two values or condition codes and cause a special internal exception if the result is true. The `SkipBlock` internal exception generated by these uops tells the core to literally annul all uops in this instruction, dynamically turning it into a nop. As described above, this is useful for string operations where a zero count causes all of the instruction's side effects to be annulled. Similarly, the `AssistCheck` internal exception dynamically turns the instruction into an assist, for those cases where certain rare conditions may require microcode intervention more complex than can be inlined into the decoded instruction stream.

## 5.9 Shifts and Rotates

The shift and rotate instructions have some of the most bizarre semantics in the entire x86 instruction set: they may or may not modify a subset of the flags depending on the rotation count operand, which we may not even know until the instruction issues. For fixed shifts and rotates, these semantics can be preserved by the uops generated, however variable rotations are more complex. The `collcc` uop is put to use here to collect all flags; the collected result is then fed into the shift or rotate uop as its `rc` operand; the uop then replicates the precise x86 behavior (including rotates using the carry flag) according to its input operands.

## 5.10 SSE Support

PTLsim provides full support for SSE and SSE2 vector floating point and fixed point, in both scalar and vector mode. As is done in the AMD K8 and Pentium 4, each SSE operation on a 128-

bit vector is split into two 64-bit halves; each half (possibly consisting of a 64-bit load and one or more FPU operations) is scheduled independently. Because SSE instructions do not set flags like x86 integer instructions, architectural state management can be restricted to the 16 128-bit SSE registers (represented as 32 paired 64-bit registers). The `mxcsr` (media extensions control and status register) is represented as an internal register that is only read and written by serializing microcode; since the exception and status bits are "sticky" (i.e. only set, never cleared by hardware), this has no effect on out of order execution. The processor's floating point units can operate in either 64-bit IEEE double precision mode or on two parallel 32-bit single precision values.

PTLsim also includes a variety of vector integer uops used to construct SSE2/MMX operations, including packed arithmetic and shuffles.

## 5.11 x87 Floating Point

The legacy x87 floating point architecture is the bane of all x86 processor vendors' existence, largely because its stack based nature makes out of order processing so difficult. While there are certainly ways of translating stack based instruction sets into flat addressing for scheduling purposes, we do not do this. Fortunately, following the Pentium III and AMD Athlon's introduction, x87 is rapidly headed for planned obsolescence; most major applications released within the last few years now use SSE instructions for their floating point needs either exclusively or in all performance critical parts. To this end, even Intel has relegated x86 support on the Pentium 4 and Core 2 to a separate low performance legacy unit, and AMD has restricted x87 use in 64-bit mode. For this reason, PTLsim translates legacy x87 instructions into a serialized, program ordered and emulated form; the hardware does not contain any x87-style 80-bit floating point registers (all floating point hardware is 32-bit and 64-bit IEEE compliant). We have noticed little to no performance problem from this approach when examining typical binaries, which rarely if ever still use x87 instructions in compute-intensive code.

## 5.12 Floating Point Unavailable Exceptions

The x86 architecture specifies a mode in which all floating point operations (SSE and x87) will trigger a Floating Point Unavailable exception (`EXCEPTION_x86_fpu_not_avail`, vector 0x7) if the `TS` (task switched) bit in control register `CR0` is set. This allows the kernel to defer saving the floating point registers and state of the previously scheduled thread until that state is actually modified, thus speeding up context switches. PTLsim supports this feature by requiring any commits to the floating point state (SSE XMM registers, x87 registers or any floating point related control or status registers) to check the `uop.is_sse` and `uop.is_x87` bits in the uop. If either of these is set, the pipeline must be flushed and redirected into the kernel so it can save the FPU state.

## 5.13 Assists

Some operations are too complex to inline directly into the uop stream. To perform these instructions, a special uop (`brp`: branch private) is executed to branch to an *assist* function implemented in microcode. In PTLsim, some assist functions are implemented as regular C/C++ or assembly language code when they interact with the rest of the virtual machine. Examples of instructions requiring assists include system calls, interrupts, some forms of integer division, handling of rare floating point conditions, CPUID, MSR reads/writes, various x87 operations, any serializing instructions, etc. These are listed in the `ASSIST_xxx` enum found in `decode.h`.

Prior to entering an assist, uops are generated to load the `REG_selfrip` and `REG_nextrip` internal registers with the RIP of the instruction itself and the RIP after its last byte, respectively. This lets the assist microcode correctly update RIP before returning, or signal a fault on the instruction if needed. Several other assist related registers, including `REG_ar1`, `REG_ar2`, `REG_ar3`, are used to store parameters passed to the assist. These registers are not architecturally visible, but must be renamed and separately maintained by the core as if they were part of the user-visible state.

While the exact behavior depends on the core model (out of order, SMT, sequential, etc), generally when the processor fetches an assist (`brp` uop), the frontend pipeline is stalled and execution waits until the `brp` commits, at which point an assist function within PTLsim is called. This is necessary because assists are not subject to the out of order execution mechanism; they directly update the architectural registers on their own. In a real processor there are slightly more efficient ways of doing this without flushing the pipeline, however in PTLsim assists are sufficiently rare that the performance impact is negligible and this approach significantly reduces complexity. For the out of order core, the exact mechanism used is described in Section 24.6.

# Chapter 6

# Decoder Architecture and Basic Block Cache

## 6.1 Basic Block Cache

As described in Section 5.1, x86 instructions are decoded into transops prior to actual execution by the core. To achieve high performance, PTLsim maintains a *basic block cache* (BB cache) containing the program ordered translated uop (*transop*) sequence for previously decoded basic blocks in the program. Each basic block (`BasicBlock` structure) consists of up to 64 uops and is terminated by either a control flow operation (conditional, unconditional, indirect branch) or a barrier operation, i.e. a microcode assist (including system calls and serializing instructions).

## 6.2 Identifying Basic Blocks

In a userspace only simulator, the RIP of a basic block's entry point (plus a few other attributes described below) serves to uniquely identify that basic block, and can be used as a key in accessing the basic block cache. In a full system simulator, the BB cache must be indexed by much more than just the virtual address, because of potential virtual page aliasing and the need to persistently cache translations across context switches. The following fields, in the *RIPVirtPhys* structure, are required to correctly access the BB cache in any full system simulator or binary translation system (128 bits total):

- `rip`: Virtual address of first instruction in BB (48 bits), since embedded RIP-relative constants and branch encodings depend on this. Modern OS's map shared libraries and binaries at the same addresses every time, so translation caching remains effective across runs.
- `mfnlo`: MFN (Machine Frame Number, i.e. physical page frame number) of first byte in BB (28 bits), since we need to handle self modifying code invalidations based on physical addresses (because of possible virtual page aliasing in multiple page tables)

- `mfnhi`: MFN of last byte in BB (28 bits), since a single x86 basic block can span up to two pages. In pathological cases, it is possible to create two page tables that both map the same MFN X at virtual address V, but map different MFNs at virtual address V+4096. If an instruction crosses this page boundary, the meaning of the instruction bytes on the second page will be different; hence we must take into account both physical pages to look up the correct translation.

- Context info (up to 24 bits), since the uops generated depend on the current CPU mode and CS descriptor settings

  – `use64`: 32-bit or 64-bit mode? (encoding differences)

  – `kernel`: Kernel or user mode?

  – `df`: EFLAGS status (direction flag, etc)

  – Other info (e.g. segmentation assumptions, etc.)

The basic block cache is always indexed using an `RIPVirtPhys` structure instead of a simple RIP. To do this, the `RIPVirtPhys.rip` field is set to the desired RIP, then `RIPVirtPhys.update(ctx)` is called to translate the virtual address onto the two physical page MFNs it could potentially span (assuming the basic block crosses two pages).

Notice that the other attribute bits (`use64`, `kernel`, `df`) mean that two distinct basic blocks may be decoded from the exact same RIP on the same physical page(s), yet the uops in each translated basic block will be different because the two basic blocks were translated in a different context (relative to these attribute bits). This is especially important for x86 string move/compare/store/load/scan instructions (`MOVSB`, `CMPSB`, `STOSB`, `LODSB`, `SCASB`), since the correct increment constants depend on the state of the direction flag in the context in which the BB was used. Similarly, if a user program tries to decode a supervisor-only opcode, code to call the general protection fault handler will be produced instead of the real uops produced only in kernel mode.

## 6.3 Invalid Translations

The `BasicBlockCache.translate(ctx, rvp)` function *always* returns a `BasicBlock` object, even if the specified RIP was on an invalid page or some of the instruction bytes were invalid. When decoding cannot continue for some reason, the decoder simply outputs a microcode branch to one of the following assists:

- `ASSIST_INVALID_OPCODE` when the opcode or instruction operands are invalid relative to the current context.

- `ASSIST_EXEC_PAGE_FAULT` when the specified RIP falls on an invalid page. This means a page is marked as not present in the current page table at the time of decoding, or the page is present but has its NX (no execute) bit set in the page table entry. The `EXEC_PAGE_FAULT` assist is also generated when the page containing the RIP itself is valid, but part of an instruction extends beyond that page onto an invalid page. The decoder tries to decode as many instruction bytes as possible, but will insert an `EXEC_PAGE_FAULT` assist whenever it determines, based on the bytes already decoded, that the remainder of the instruction would fall on the invalid page.

- `ASSIST_GP_FAULT` when attempting to decode a restricted kernel-only opcode while running in user mode.

Before redirecting execution to the kernel's exception handler, the `EXEC_PAGE_FAULT` microcode verifies that the page in question is still invalid. This avoids a spurious page fault in the case where an instruction was originally decoded on an invalid page, but the page tables were updated after the translation was first made such that the page is now valid. When this is the case, all bogus basic blocks on the page (which were decoded into a call to `EXEC_PAGE_FAULT`) must be invalidated, allowing a correct translation to be made now that the page is valid. The page at the virtual address after the page in question may also need to be invalidated in the case where some instruction bytes cross the page boundary.

## 6.4 Self Modifying Code

In x86 processors, the translation process is considerably more complex, because of self modifying code (SMC) and its variants. Specifically, the instruction bytes of basic blocks that have already been translated and cached may be overwritten; these old translations must be discarded. The x86 architecture guarantees that all code modifications will be visible immediately after the instruction making the modification; unlike other architectures, no "instruction cache flush" operation is provided. Several kinds of SMC must be handled correctly:

- Classical SMC: stores currently in the pipeline overwrite other instructions that have already been fetched into the pipeline and even speculatively executed out of order;
- Indirect SMC: stores write to a page on which previously translated code used to reside, but that page is now being reused for unrelated data or new code. This case frequently arises in operating system kernels when pages are swapped in and out from disk.
- Cross-modifying SMC: in a multiprocessor system, one processor overwrites instructions that are currently in the pipeline on some other core. The x86 standard is ambiguous here; technically no pipeline flush and invalidate is required; instead, the cache coherence mechanism and software mutexes are expected to prevent this case.
- External SMC: an external device uses direct memory access (DMA) to overwrite the physical DRAM page containing previously translated code. In theory, this can happen while the affected instructions are in the pipeline, but in practice no operating system would ever allow this. However, we still must invalidate any translations on the target page to prevent them from being looked up far in the future.

To deal with all these forms of SMC, PTLsim associates a “dirty” bit with every physical page (this is unrelated to the “dirty” bit in user-visible page table entries). Whenever the first uop in an x86 instruction (i.e. the “SOM”, start-of-macro-op uop) commits, the current context is used to translate its RIP into the physical page MFN on which it resides, as described in Section 6.2. If the instruction’s length in bytes causes it to overlap onto a second page, that high MFN is also looked up (using the virtual address *rip* + 4096). If the dirty bits for either the low or high MFN are set, this means the instruction bytes may have been modified sometime after the time they were last translated and added to the basic block cache. In this case, the pipeline must be flushed, and all basic blocks on the target MFN (and possibly the overlapping high MFN) must be invalidated before clearing the dirty bit. Technically the RIP-to-physical translation would be done in the instruction fetch stage in most core models, then simply stored as an `RIPVirtPhys` structure inside the uop until commit time.

The dirty bit can be set by several events. Obviously any store uops will set the dirty bit (thus handling the classical, indirect and cross-modifying cases), but notice that this bit is not checked again until the first uop in the *next* x86 instruction. This behavior is required because it is perfectly legal for an x86 store to overwrite its own instruction bytes, but this does not become visible until the same instruction executes a second time (otherwise an infinite loop of invalidations would occur). Microcoded x86 instructions implemented by PTLsim itself set dirty bits when their constituent internal stores commit. Finally, DMA transfers and external writes also set the dirty bit of any pages touched by the DMA operation.

The dirty bit is only cleared when all translated basic blocks are invalidated on a given page, and it remains clear until the first write to that page. However, no action is taken when additional basic blocks are decoded from a page already marked as dirty. This may seem counterintuitive, but it is necessary to avoid deadlock: if the page were invalidated and retranslated at fetch time, future stages in a long pipeline could potentially still have references to unrelated basic blocks on the page being invalidated. Hence, all invalidations are checked and processed only at commit time.

Other binary translation based software and hardware [17, 12, 10, 13, 14] have special mechanisms for write protecting physical pages, such that when a page with translations is first written by stores or DMA, the system immediately invalidates all translations on that page. Unfortunately, this scheme has a number of disadvantages. First, patents cover its implementation [19, 18, 17], which we would like to avoid. In addition, our design eliminates forced invalidations when the kernel frees up a page containing code that’s immediately overwritten with normal user data (a very common pattern according to our studies). If that page is never executed again, any translations from it will be discarded in the background by the LRU mechanism, rather than interrupting execution to invalidate translations that will never be used again anyway. Fortunately, true classical SMC is very rare in modern x86 code, in large part because major microprocessors have slapped a huge penalty on its use (particularly in the case of the Pentium 4 and Transmeta processors, both of which store translated uops in a cache similar to PTLsim’s basic block cache).

## 6.5 Memory Management of the Basic Block Cache

The PTLsim memory manager (in `mm.cpp`, see Section 7.3 for details) implements a reclaim mechanism in which other subsystems register functions that get called when an allocation fails. The basic block cache registers a callback, `bbcache_reclaim()` and `BasicBlockCache::reclaim()`, to invalidate and free basic blocks when PTLsim runs out of memory.

The algorithm used to do this is a pseudo-LRU design. Every basic block has a `lastused` field that gets updated with the current cycle number whenever `BasicBlock::use(sim_cycle)` is called (for instance, in the fetch stage of a core model). The reclaim algorithm goes through all basic blocks and calculates the oldest, average and newest `lastused` cycles. The second pass then invalidates any basic blocks that fall below this average cycle; typically around half of all basic blocks fall in the least recently used category. This strategy has proven very effective in freeing up a large amount of space without discarding currently hot basic blocks.

Each basic block also has a reference counter, `refcount`, to record how many pointers or references to that basic block currently exist anywhere inside PTLsim (especially in the pipelines of core models). The `BasicBlock::acquire()` and `release()` methods adjust this counter. Core models should acquire a basic block once for every uop in the pipeline within that basic block; the basic block is released as uops commit or are annulled. Since basic blocks may be speculatively translated in the fetch stage of core models, this guarantees that live basic blocks currently in flight are never freed until they actually leave the pipeline.

# Chapter 7

# PTLsim Support Subsystems

## 7.1 Uop Implementations

PTLsim provides implementations for all uops in the `uopimpl.cpp` file. C++ templates are combined with gcc's smart inline assembler type selection constraints to translate all possible permutations (sizes, condition codes, etc) of each uop into highly optimized code. In many cases, a real x86 instruction is used at the core of each corresponding uop's implementation; code after the instruction just captures the generated x86 condition code flags, rather than having to manually emulate the same condition codes ourselves. The code implementing each uop is then called from elsewhere in the simulator whenever that uop must be executed. Note that loads and stores are implemented elsewhere, since they are too dependent on the specific core model to be expressed in this generic manner.

An additional optimization, called *synthesis*, is also used whenever basic blocks are translated. Each uop in the basic block is mapped to the address of a native PTLsim function in `uopimpl.cpp` implementing the semantics of that uop; this function pointer is stored in the `synthops[]` array of the `BasicBlock` structure. This saves us from having to use a large jump table later on, and can map uops to pre-compiled templates that avoid nearly all further decoding of the uop during execution.

## 7.2 Configuration Parser

PTLsim supports a wide array of command line or scriptable configuration options, described in Section 10.3. The configuration parser engine (used by both PTLsim itself and utilities like PTLstats) is in `config.cpp` and `config.h`. For PTLsim itself, each option is declared in three places:

- `ptlsim.h` declares the `PTLsimConfig` structure, which is available from anywhere as the `config` global variable. The fields in this structure must be of one of the following types: `W64` (64-bit integer), `double` (floating point), `bool` (on/off boolean), or `stringbuf` (for text parameters).

- `ptlsim.cpp` declares the `PTLsimConfig::reset()` function, which sets each option to its default value.

- `ptlsim.cpp` declares the `ConfigurationParser<PTLsimConfig>::setup()` template function, which registers all options with the configuration parser.

## 7.3 Memory Manager

### 7.3.1 Memory Pools

PTLsim uses its own custom memory manager for all allocations, given its specialized constraints (particularly for PTLsim/X, which runs on the bare hardware). The PTLsim memory manager (in `mm.cpp`) uses three key structures.

The *page allocator* allocates spans of one or more virtually contiguous pages. In userspace-only PTLsim, the page allocator doesn't really exist: it simply calls `mmap()` and `munmap()`, letting the host kernel do the actual allocation. In the full system PTLsim/X, the page allocator actually works with physical pages and is based on the extent allocator (see below). The `ptl_alloc_private_pages()` and `ptl_free_private_pages()` functions should be used to directly allocate page-aligned memory (or individual pages) from this pool.

The *general allocator* uses the `ExtentAllocator` template class to allocate large objects (greater than page sized) from a pool of free extents. This allocator automatically merges free extents and can find a matching free block in O(1) time for any allocation size. The general allocator obtains large chunks of memory (typically 64 KB at once) from the page allocator, then sub-divides these extents into individual allocations.

The *slab allocator* maintains a pool of page-sized "slabs" from which fixed size objects are allocated. Each page only contains objects of one size; a separate slab allocator handles each size from 16 bytes up to 1024 bytes, in 16-byte increments. The allocator provides extremely fast allocation performance for object oriented programs in which many objects of a given size are allocated. The slab allocator also allocates one page at a time from the global page allocator. However, it maintains a pool of empty pages to quickly satisfy requests. This is the same architecture used by the Linux kernel to satisfy memory requests.

The `ptl_mm_alloc()` function intelligently decides from which of the two allocators (general or slab) to allocate a given sized object, based on the size in bytes, object type and caller. The standard `new` operator `and malloc()` both use this function. Similarly, the `ptl_mm_free()` function frees memory. PTLsim uses a special bitmap to track which pages are slab allocator pages; if a pointer falls within a slab, the slab deallocator is used; otherwise the general allocator is used to free the extent.

### 7.3.2 Garbage Collection and Reclaim Mechanism

The memory manager implements a garbage collection mechanism with which other subsystems register reclaim functions that get called when an allocation fails. The `ptl_mm_register_reclaim_handler()` function serves this role. Whenever an allocation fails, the reclaim handlers are called in sequence, followed by an extent cleanup pass, before retrying the allocation. This process repeats until the allocation succeeds or an abort threshold is reached.

The reclaim function gets passed two parameters: the size in bytes of the failed allocation, and an *urgency* parameter. If *urgency* is 0, the subsystem registering the callback should do everything in its power to free all memory it owns. Otherwise, the subsystem should progressively trim more and more unused memory with each call (and increasing urgency). *Under no circumstances* is a reclaim handler allowed to allocate *any* additional memory! Doing so will create an infinite loop; the memory manager will detect this and shut down PTLsim if it is attempted.

# Chapter 8

# Statistics Collection and Analysis

## 8.1 PTLsim Statistics Data Store

### 8.1.1 Introduction

PTLsim maintains a huge number of statistical counters and data points during the simulation process, and can optionally save this data to a file by using the "`-stats` *filename*" configuration option. The data store is a binary file format used to efficiently capture large quantities of statistical information for later analysis. This file format supports storing multiple regular or triggered snapshots of all counters. Snapshots can be subtracted, averaged and extensively manipulated, as will be described later on.

PTLsim makes it trivial to add new performance counters to the statistics data tree. All counters are defined in `stats.h` as a tree of nested structures; the top-level `PTLsimStats` structure is mapped to the global variable `stats`, so counters can be directly updated from within the code by simple increments, e.g. `stats.xxx.yyy.zzz.countername++`. Every node in the tree can be either a `struct`, `W64` (64-bit integer), `double` (floating point) or `char` (string) type; arrays of these types are also supported. In addition, various attributes, described below, can be attached to each node or counter to specify more complex semantics, including histograms, labeled arrays, summable nodes and so on.

PTLsim comes with a special script, `dstbuild` ("data store template builder") that parses `stats.h` and constructs a binary representation (a "template") describing the structure; this template data is then compiled into PTLsim. Every time PTLsim creates a statistics file, it first writes this template, followed by the raw `PTLsimStats` records and an index of those records by name. In this way, the complete data store tree can be reconstructed at a later time even if the original `stats.h` or PTLsim version that created the file is unavailable. This scheme is analogous to the separation of XML schemas (the template) from the actual XML data (the stats records), but in our case the template and data is stored in binary format for efficient parsing.

We suggest using the data store mechanism to store *all* statistics generated by your additions to PTLsim, since this system has built-in support for snapshots, checkpointing and structured easy to parse data (unlike simply writing values to a text file). It is further suggested that only raw values be saved, rather than doing computations in the simulator itself - leave the analysis to PTLstats after gathering the raw data. If some limited computations do need to be done before writing each statistics record, PTLsim will call the PTLsimMachine::update_stats() virtual method to allow your model a chance to do so before writing the counters.

### 8.1.2 Node Attributes

After each node or counter is declared, one of several special C++-style "//" comments can be used to specify *attributes* for that node:

- `struct Name { // rootnode:`

  The node is at the root of the statistics tree (typically this only applies to the PTLsimStats structure itself)

- `struct Name { // node: summable`

  All subnodes and counters under this node are assumed to total 100% of whatever quantity is being measured. This attribute tells PTLstats to print percentages next to the raw values in this subtree for easier viewing.

- `W64 name[arraysize]; // histo:` *`min, max, stride`*

  Specifies that the array of counters forms a *histogram*, i.e. each slot in the array represents the number of occurrences of one event out of a mutually exclusive set of events. The *min* parameter specifies the meaning of the first slot (array element 0), while the *max* parameter specifies the meaning of the last slot (array element *arraysize*-1). The *stride* parameter specifies how many events are counted into every slot (typically this is 1).

  For example, let's say you want to measure the frequency distribution of the number of consumers of each instruction's result, where the maximum number of possible consumers is 256. You could specify this as:

  ```
  W64 consumers[64+1]; // histo: 0, 256, 4
  ```

  This histogram has a logical range of 0 to 256, but is divided into 65 slots. Because the *stride* parameter is 4, any consumer counts from 0 to 3 increment slot 0, counts from 4 to 7 increment slot 1, and so on. When you update this counter array from inside the model, you should do so as follows:

  ```
  stats.xxx.yyy.consumers[min(n / 4, 64)]++;
  ```

- `W64 name[arraysize]; // label: namearray`

  Specifies that the array of counters is a histogram of named, mutually exclusive events, rather than simply raw numbers (as with the *histo* attribute). The *namearray* must be the name of an array of *arraysize* strings, with one entry per event.

  For example, let's say you want to measure the frequency distribution of uop types PTLsim is executing. If there are OPCLASS_COUNT, you could declare the following:

```
W64 opclass[OPCLASS_COUNT]; // label:  opclass_names
```

In some header file included by `stats.h`, you need to declare the actual array of slot labels:

```
static const char* opclass_names[OPCLASS_COUNT] = {"logic", "addsub",
"addsubc", ...};
```

### 8.1.3 Configuration Options

PTLsim supports several options related to the statistics data store:

- `-stats` *`filename`*

  Specify the filename to which statistics data is written. In reality, two files are created: *filename* contains the template and snapshot index, while *filename.data* contains the raw data.

- `-snapshot-cycles` *`N`*

  Creates a snapshot every N simulation cycles, numbered consecutively starting from 0. Without this option, only one snapshot, named `final`, is created at the end of the simulation run.

- `-snapshot-now` *`name`*

  Creates a snapshot named *name* at the current point in the simulation. This can be used to asynchronously take a look at a simulation in progress. *This option is only available in PTLsim/X.*

## 8.2 PTLstats: Statistics Analysis and Graphing Tools

The ***PTLstats*** program is used to analyze the statistics data store files produced by PTLsim. PTLstats will first extract the template stored in all data store files, and will then parse the statistics records into a flexible tree format that can be manipulated by the user. The following is an example of one node in the statistics tree, as printed by PTLstats:

```
dcache {
  store {
    issue (total 68161716) {
    [ 29.7% ] replay (total 20218780) {
    [  0.0% ] sfr_addr_not_ready = 0;
    [ 16.8% ] sfr_data_and_data_to_store_not_ready = 3405878;
    [ 11.8% ] sfr_data_not_ready = 2379338;
    [ 23.4% ] sfr_addr_and_data_to_store_not_ready = 4740838;
    [ 24.5% ] sfr_addr_and_data_not_ready = 4951888;
    [ 23.4% ] sfr_addr_and_data_and_data_to_store_not_ready = 4740838;
  }
```

```
    [  0.0% ] exception = 30429;
    [  7.9% ] ordering = 5404592;
    [ 62.4% ] complete = 42507854;
    [  0.0% ] unaligned = 61;
  }
```

Notice how PTLstats will automatically sum up all entries in certain branches of the tree to provide the user with a breakdown by percentages of the total for that subtree in addition to the raw values. This is achieved using the "`// node: summable`" attribute as described in Section 8.1.2.

Here is an example of a labeled histogram, produced using the "`// label: xxx`" attribute described in Section 8.1.2:

```
  size[4] = {
    ValRange: 3209623 90432573
    Total:   107190122
    Thresh:     10720
    [  6.2% ]        0  6686971 1 (byte)
    [  6.4% ]        1  6860955 2 (word)
    [ 84.4% ]        2 90432573 4 (dword)
    [  3.0% ]        3  3209623 8 (qword)
  };
```

## 8.3 Snapshot Selection

The basic syntax of the PTLstats command is "`ptlstats` *-options filename*". If no options are specified, PTLstats prints out the entire statistics tree from its root, relative to the `final` snapshot.

To select a specific snapshot, use the following option:

```
  ptlstats -snapshot name-or-number ...
```

Snapshots may be specified by name or number.

It may be desirable to examine the difference in statistics *between* two snapshots, for instance to subtract out the counters at the starting point of a given run or after a warmup period. The `-subtract` option provides this facility, for example:

```
  ptlstats -snapshot final -subtract startpoint ...
```

## 8.4 Working with Statistics Trees: Collection, Averaging and Summing

To select a specific subtree of interest, use the syntax of the following example:

```
ptlstats -snapshot final -collect /ooocore/dcache/load example1.stats example2.stats ...
```

This will print out the subtree `/ooocore/dcache/load` in the snapshot named `final` (the default snapshot) for each of the named statistics files `example1.stats`, `example2.stats` and so on. Multiple files are generally used to examine a specific subnode across several benchmarks.

Subtrees or individual statistics can also be summed and averaged across many files, using the `-collectsum` or `-collectaverage` commands in place of `-collect`.

## 8.5 Traversal and Printing Options

The `-maxdepth` option is useful for limiting the depth (in nodes) PTLstats will descend into the specified subtree. This is appropriate when you want to summarize certain classes of statistics printed as percentages of the whole, yet don't want a breakdown of every sub-statistic.

The `-percent-of-toplevel` option changes the way percentages are displayed. By default, percentages are calculated by dividing the total value of each node by the total of its immediate parent node. When `-percent-of-toplevel` is enabled, the divisor becomes the total of the entire subtree, possibly going back several levels (i.e. back to the highest level node marked with the *summable* attribute), rather than each node's immediate parent.

## 8.6 Table Generation

PTLstats provides a facility to easily generate R-row by C-column data tables from a set of R benchmarks run with C different sets of parameters. Tables can be output in a variety of formats, including plain text with tab or space delimiters (suitable for import into a spreadsheet), LaTeX (for direct insertion into research reports) or HTML. To generate a table, use the following syntax:

```
ptlstats -table /final/summary/cycles -rows gzip,gcc,perlbmk,mesa -cols small,large,huge -t
```

In this example, the benchmarks ("gzip", "gcc", "perlbmk", "mesa") will form the rows of the table, while three trials done for each benchmark ("small", "large", "huge") will be listed in the columns. The row and column names will be combined using the pattern "`%row/ptlsim.stats.%col`" to generate statistics data store filenames like "`gzip/ptlsim.stats.small`". PTLstats will then load the data store for each benchmark and trial combination to create the table.

Notice that you must create your own scripts, or manually run each benchmark and trial with the desired PTLsim options, plus "`-stats ptlsim.stats.`*`trialname`*". PTLstats will only report these results in table form; it will not actually run any benchmarks.

The `-tabletype` option specifies the data format of the table: "`text`" (plain text with space delimiters, suitable for import into a spreadsheet), "`latex`" (LaTeX format, useful for directly inserting into research reports), or "`html`" (HTML format for web pages).

The "`-scale-relative-to-col N`" option forces PTLstats to compute the percentage of increase or decrease for each cell relative to the corresponding row in some other reference column *N*. This is useful when running a "baseline" case, to be displayed as a raw value (usually the cycle count, `/final/summary/cycles`) in column 0, while all other experimental cases are displayed as a percentage increase (fewer cycles, for a positive percentage) or percentage decrease (negative value) relative to this first column ($N = 0$).

### 8.6.1 Bargraph Generation

In addition to creating tables, PTLstats can directly create colorful graphs (in Scalable Vector Graphics (SVG) format) from a set of benchmarks (specified by the `-rows` option) and trials of each benchmark (specified by the `-cols` option). For instance, to plot the total number of cycles taken over a set of benchmarks, each run under different PTLsim configurations, use the following example:

```
ptlstats -bargraph /final/summary/cycles -rows gzip,gcc,perlbmk,mesa -cols small,large,huge
```

In this case, groups of three bars (for the trials "small", "large", "huge") appear for each benchmark.

The graph's layout can be extensively customized using the options `-title`, `-width`, `-height`.

Inkscape (http://www.inkscape.org) is an excellent vector graphics system for editing and formatting SVG files generated by PTLstats.

## 8.7 Histogram Generation

Certain array nodes in the statistics tree can be tagged as "histogram" nodes by using the `histo:` or `label:` attributes, as described in Section 8.1.2. For instance, the `ooocore/frontend/consumer-count` node in the out-of-order core is a histogram node. PTLstats can directly create graphs (in Scalable Vector Graphics (SVG) format) for these special nodes, using the `-histogram` option:

```
ptlstats -histogram /ooocore/frontend/consumer-count > example.svg
```

The histogram's layout can be extensively customized using the options `-title`, `-width`, `-height`. In addition, the `-percentile` option is useful for controlling the displayed data range by excluding data under the Nth percentile. The `-logscale` and `-logk` options can be used to apply a log scale (instead of a linear scale) to the histogram bars. The syntax of these options can be obtained by running `ptlstats` without arguments.

# Chapter 9

# Benchmarking Techniques

## 9.1 Trigger Mode and other PTLsim Calls From User Code

PTLsim optionally allows user code to control the simulator mode through the `ptlcall_xxx()` family of functions found in `ptlcalls.h` when trigger mode is enabled (`-trigger` configuration option). This file should be included by any PTLsim-aware user programs; these programs must be recompiled to take advantage of these features. Amongst the functions provided by `ptlcalls.h` are:

- `ptlcall_switch_to_sim()` is only available while the program is executing in native mode. It forces PTLsim to regain control and begin simulating instructions as soon as this call returns.
- `ptlcall_switch_to_native()` stops simulation and returns to native execution, effectively removing PTLsim from the loop.
- `ptlcall_marker()` simply places a user-specified marker number in the PTLsim log file
- `ptlcall_capture_stats()` adds a new statistics data store snapshot at the time it is called. You can pass a string to this function to name your snapshot, but all names must be unique.
- `ptlcall_nop()` does nothing but test the call mechanism.

In userspace PTLsim, these calls work by forcing execution to code on a "gateway page" at a specific fixed address (`0x1000` currently); PTLsim will write the appropriate call gate code to this page depending on whether the process is in native or simulated mode. In native mode, the call gate page typically contains a 64-to-64-bit or 32-to-64-bit far jump into PTLsim, while in simulated mode it contains a reserved x86 opcode interpreted by the x86 decoder as a special kind of system call. If PTLsim is built on a 32-bit only system, no mode switch is required.

In full system PTLsim/X, the x86 opcodes used to implement these calls are directly handled by the PTLsim/X hypervisor as if they were actually part of the native x86 instruction set.

Generally these calls are used to perform "intelligent benchmarking": the `ptlcall_switch_to_sim()` call is made at the top of the main loop of a benchmark after initialization, while the `ptlcall_switch_to_native()` call is inserted after some number of iterations to stop simulation after a representative subset of the code has completed. This intelligent approach is far better than the blind "sample for N million cycles after S million startup cycles" approach used by most researchers.

Fortran programs will have to actually link in the `ptlcalls.o` object file, since they cannot include C header files. The function names that should be used in the Fortran code remain the same as those from the `ptlcalls.h` header file.

## 9.2 Notes on Benchmarking Methodology and "IPC"

The x86 instruction set requires some different benchmarking techniques than classical RISC ISAs. In particular, **uIPC (Micro-Instructions per Cycle) a NOT a good measure of performance for an x86 processor.** Because one x86 instruction may be broken up into numerous uops, it is never appropriate to compare IPC figures for committed x86 instructions per clock with IPC values from a RISC machine. Furthermore, different x86 implementations use varying numbers of uops per x86 instruction as a matter of encoding, so even comparing the uop based IPC between x86 implementations or RISC-like machines is inaccurate.

Users are strongly advised to use relative performance measures instead. Comparing the total simulated cycle count required to complete a given benchmark between different simulator configurations is much more appropriate than IPC with the x86 instruction set. An example would be "the baseline took 100M cycles, while our improved system took 50M cycles, for a 2x improvement".

## 9.3 Simulation Warmup Periods

In some simulators, it is possible to quickly skip through a specific number of instructions before starting to gather statistics, to avoid including initialization code in the statistics. In PTLsim, this is neither necessary nor desirable. Because PTLsim directly executes your program on the host CPU until it switches to cycle accurate simulation mode, there is no way to count instructions in this manner.

Many researchers have gotten in the habit of blindly skipping a large number of instructions in benchmarks to avoid profiling initialization code. However, this is not a very intelligent policy: different benchmarks have different startup times until the top of the main loop is reached, and it is generally evident from the benchmark source code where that point should be. Therefore, PTLsim supports **trigger points:** by inserting a special function call (`ptlcall_switch_to_sim`) within the benchmark source code and recompiling, the `-trigger` PTLsim option can be used to run the code on the host CPU until the trigger point is reached. If the source code is unavailable, the `-startrip` *`0xADDRESS`* option will start full simulation only at a specified address (e.g. function entry point).

If you want to warm up the cache and branch predictors prior to starting statistics collection, combine the `-trigger` option with the `-snapshot-cycles` *`N`* option, to start full simulation at the top of the benchmark's main loop (where the trigger call is), but only start gathering statistics *N* cycles later, after the processor is warmed up. Remember, since the trigger point is placed *after* all initialization code in the benchmark, in general it is only necessary to use 10-20 million cycles of warmup time before taking the first statistics snapshot. In this time, the caches and branch predictor will almost always be completely overwritten many times. This approach significantly speeds up the simulation without any loss of accuracy compared to the "fast simulation" mode provided by other simulators.

In PTLstats, use the `-subtract` option to make sure the final statistics don't include the warmup period before the first snapshot. To subtract the final snapshot from snapshot 0 (the first snapshot after the warmup period), use a command similar to the following:

```
ptlstats -subtract 0 ptlsim.stats
```

## 9.4 Sequential Mode

PTLsim supports *sequential mode*, in which instructions are run on a simple, in-order processor model (in `seqcore.cpp`) without accounting for cache misses, branch mispredicts and so forth. This is much faster than the out of order model, but is obviously slower than native execution. The purpose of sequential mode is mainly to aid in testing the x86 to uop decoder, microcode functions and RTL-level uop implementation code. It may also be useful for gathering certain statistics on the instruction mix and count without running a full simulation.

*NOTE:* Sequential mode is *not* intended as a "warmup mode" for branch predictors and caches. If you want this behavior, use statistical snapshot deltas as described in Section 9.3.

Sequential mode is enabled by specifying the "`-core seq`" option. It has no other core-specific options.

# Part II

# PTLsim Classic: Userspace Linux Simulation

# Chapter 10

# Getting Started with PTLsim

*NOTE:* This part of the manual is relevant only if you are using the classic userspace-only version of PTLsim. If you are looking for the full system SMP/SMT version, PTLsim/X, please skip this entire part and read Part III instead.

## 10.1 Building PTLsim

Prerequisites:

- PTLsim can be built on **both 64-bit x86-64 machines** (AMD Athlon 64 / Opteron / Turion, Intel Pentium 4 with EM64T and Intel Core 2) **as well as ordinary 32-bit x86 systems**. In either case, your system must support SSE2 instructions; all modern CPUs made in the last few years (such as Pentium 4 and Athlon 64) support this, but older CPUs (Pentium III and earlier) specifically do *not* support PTLsim.
- If built for x86-64, PTLsim will run both 64-bit and 32-bit programs automatically. If built on a 32-bit Linux distribution and compiler, PTLsim only supports ordinary x86 programs and will typically be slower than the 64-bit build**,** even on 32-bit user programs.
- PTLsim runs on any recent Linux 2.6 based distribution.
- We have successfully built PTLsim with gcc 3.3, 3.4.x and 4.1.x+ (gcc 4.0.x has documented bugs affecting some of our code).

Quick Start Steps:

- Download PTLsim from our web site (`http://www.ptlsim.org/download.php`). We recommend starting with the "stable" version, since this contains all the files you need and can be updated later if desired.
- Unpack `ptlsim-2006xxxx-rXXX.tar.gz` to create the `ptlsim` directory.

- Run `make`.
  - The Makefile will detect your platform and automatically compile the correct version of PTLsim (32-bit or 64-bit).

## 10.2 Running PTLsim

PTLsim invocation is very simple: after compiling the simulator and making sure the `ptlsim` executable is in your path, simply run:

`ptlsim` *full-path-to-executable arguments...*

PTLsim reads configuration options for running various user programs by looking for a configuration file named `/home/`*`username`*`/.ptlsim/`*`path/to/program/executablename`*`.conf`. To set options for each program, you'll need to create a directory of the form `/home/`*`username`*`/.ptlsim` and make sub-directories under it corresponding to the full path to the program. For example, to configure `/bin/ls` you'll need to run "`mkdir /home/`*`username`*`/.ptlsim/bin`" and then edit "`/home/`*`username`*`/.ptlsim/bin/ls.conf`" with the appropriate options. For example, try putting the following in `ls.conf` as described:

```
-logfile ptlsim.log -loglevel 9 -stats ls.stats -stopinsns 10000
```

Then run:

```
ptlsim /bin/ls -la
```

PTLsim should display its system information banner, then the output of simulating the directory listing. With the options above, PTLsim will simulate `/bin/ls` starting at the first x86 instruction in the dynamic linker's entry point, run until 10000 x86 instructions have been committed, and will then switch back to native mode (i.e. the user code will run directly on the real processor) until the program exits. During this time, it will compile an extensive log of the state of every micro-operation executed by the processor and will save it to "`ptlsim.log`" in the current directory. It will also create "`ls.stats`", a binary file containing snapshots of PTLsim's internal performance counters. The `ptlstats` program (Chapter 8) can be used to print and analyze these statistics by running "`ptlstats ls.stats`".

## 10.3 Configuration Options

PTLsim supports a variety of options in the configuration file of each program; you can run "`ptlsim`" without arguments to get a full list of these options. The following sections only list the most useful options, rather than every possible option.

The configuration file can also contain comments (starting with "`#`" at any point on a line) and blank lines; the first non-comment line is used as the active configuration.

PTLsim supports multiple models of various microprocessor cores; the "`-core` *corename*" option can be used to choose a specific core. The default core is "`ooo`", the dynamically scheduled out of order superscalar core described in great detail in Part IV. PTLsim also comes with a simple sequential in-order core, "`seq`". It is most useful for debugging decoding and microcode issues rather than actual performance profiling.

## 10.4 Logging Options

PTLsim can log all simulation events to a log file, or can be instructed to log only a subset of these events, starting and stopping at various points:

- `-logfile` *filename*

  Specifies the file to which log messages will be written.

- `-loglevel` *level*

  Selects a subset of the events that will be logged:

  - 0 disables logging
  - 1 displays only critical events (such as system calls and state changes)
  - 2-3 displays less critical simulator-wide events
  - 4 displays major events within the core itself (like pipeline flushes, basic block decodes, etc)
  - 6 displays *all* events that occur within each pipeline stage of the core every cycle
  - 99 displays every possible event. This will create massive log files!

- `-startlog` *cycle*

  Starts logging only after *cycle* cycles have elapsed from the start of the simulation.

- `-startlogrip` *rip*

  Starts logging only after the first time the instruction at *rip* is decoded or executed. This is mutually exclusive with `-startlog`.

## 10.5 Event Log Ring Buffer

PTLsim also maintains an event log ring buffer. Every time the core takes some action (for instance, dispatching an instruction, executing a store, committing a result or annulling each uop after an exception), it writes that event to a circular buffer that contains (by default) the last 32768 events in chronological order (oldest to newest). This is extremely useful for debugging in cases where you want to "look backwards in time" from the point where a specific but unknown "bad" event occurred, but cannot leave logging at e.g. "`-loglevel 99`" enabled all the time (because it is far too slow and space consuming).

The event log ring buffer must be enabled via the `-ringbuf` option. This is disabled by default since it exacts a 25-40% performance overhead (but this is much better than the 10000%+ overhead of full logging).

PTLsim will always print the ring buffer to the log file whenever:

- Any `assert` statement fails within the out of order simulator core;
- Any fatal exception occurs;
- At user-specified points, by inserting "`core.eventlog.print(logfile);`" anywhere within the code;
- Whenever the "`-ringbuf-trigger-rip` *rip*" option is used to specify a specific trigger RIP. When the last uop at this RIP is committed, the ring buffer is printed, exposing all events that happened over the past few thousand cycles (going backwards in time from the cycle in which the trigger instruction committed)
- The event log ring buffer is automatically enabled whenever `-loglevel` is 6 or higher; in this case all events are logged to the logfile after every cycle.

## 10.6 Simulation Start Points

Normally PTLsim starts in simulation mode at the first instruction in the target program (or the Linux dynamic linker, assuming the program is dynamically linked). It may be desirable to skip time-consuming initialization parts of the program, using one of two methods.

The `-startrip` *rip* option places a breakpoint at *rip*, then immediately switches to native mode until that breakpoint is hit, at which point PTLsim begins simulation.

Alternatively, if the source code to the program is available, it may be recompiled with call(s) to a special function, `ptlcall_switch_to_sim()`, provided in `ptlcalls.h`. PTLsim is then started with the `-trigger` option, which switches it to native mode until the first call to the `ptlcall_switch_to_sim()` function, at which point simulation begins. This function, and other special code that can be used within the target program, is described in Section 9.1.

## 10.7 Simulation Stop Points

By default, PTLsim continues executing in simulation mode until the target program exits on its own. However, typically programs are profiled for a fixed number of committed x86 instructions, or until a specific point is reached, so as to ensure an identical span of instructions is executed on every trial, without waiting for the entire program to finish. The following options support this behavior:

- `-stopinsns` *insns* will stop the simulation after *insns* x86 instructions have committed.
- `-stop` *cycles* stops after *cycles* cycles have been simulated.
- `-stoprip` *rip* stops after the instruction at rip is decoded and executed the first time.

PTLsim will normally switch back to native mode after finishing simulation. If the program should be terminated instead, the `-exitend` option will do so.

The node is at the root of the statistics tree (typically this only applies to the PTLsimStats structure itself)

## 10.8 Statistics Collection

PTLsim supports the collection of a wide variety of statistics and counters as it simulates your code, and can make regular or triggered snapshots of the counters. Chapter 8 describes this support, while Section 8.1.3 documents the configuration options associated with statistics collection, including `-stats`, `-snapshot-cycles`, `-snapshot-now`.

# Chapter 11

# PTLsim Classic Internals

## 11.1 Low Level Startup and Injection

*Note:* This section deals with the internal operation of the PTLsim low level code, independent of the out of order simulation engine. If you are only interested in modifying the simulator itself, you can skip this section.

*Note:* This section does not apply to the full system PTLsim/X; please see the corresponding sections in Part III instead.

### 11.1.1 Startup on x86-64

PTLsim is a very unusual Linux program. It does its own internal memory management and threading without help from the standard libraries, injects itself into other processes to take control of them, and switches between 32-bit and 64-bit mode within a single process image. For these reasons, it is very closely tied to the Linux kernel and uses a number of undocumented system calls and features only available in late 2.6 series kernels.

PTLsim always starts and runs as a 64-bit process even when running 32-bit threads; it context switches between modes as needed. The statically linked `ptlsim` executable begins executing at `ptlsim_preinit_entry` in `lowlevel-64bit.S`. This code calls `ptlsim_preinit()` in `kernel.cpp` to set up our custom memory manager and threading environment before any standard C/C++ functions are used. After doing so, the normal `main()` function is invoked.

The `ptlsim` binary can run in two modes. If executed from the command line as a normal program, it starts up in *inject* mode. Specifically, `main()` in `ptlsim.cpp` checks if the `inside_ptlsim` variable has been set by `ptlsim_preinit_entry`, and if not, PTLsim enters inject mode. In this mode, `ptlsim_inject()` in `kernel.cpp` is called to effectively inject the `ptlsim` binary into another process and pass control to it before even the dynamic linker gets to load the program. In `ptlsim_inject()`, the PTLsim process is forked and the child is placed under the parent's control using `ptrace()`. The child process then uses `exec()` to start the user program to simulate (this can be either a 32-bit or 64-bit program).

However, the user program starts in the stopped state, allowing `ptlsim_inject()` to use `ptrace()` and related functions to inject either 32-bit or 64-bit boot loader code directly into the user program address space, overwriting the entry point of the dynamic linker. This code, derived from `injectcode.cpp` (specifically compiled as `injectcode-32bit.o` and `injectcode-64bit.o`) is completely position independent. Its sole function is to map the rest of `ptlsim` into the user process address space at virtual address `0x70000000` and set up a special `LoaderInfo` structure to allow the master PTLsim process and the user process to communicate. The boot code also restores the old code at the dynamic linker entry point after relocating itself. Finally, `ptlsim_inject()` adjusts the user process registers to start executing the boot code instead of the normal program entry point, and resumes the user process.

At this point, the PTLsim image injected into the user process exists in a bizarre environment: if the user program is 32 bit, the boot code will need to switch to 64-bit mode before calling the 64-bit PTLsim entrypoint. Fortunately x86-64 and the Linux kernel make this process easy, despite never being used by normal programs: a regular far jump switches the current code segment descriptor to `0x33`, effectively switching the instruction set to x86-64. For the most part, the kernel cannot tell the difference between a 32-bit and 64-bit process: as long as the code uses 64-bit system calls (i.e. `syscall` instruction instead of `int 0x80` as with 32-bit system calls), Linux assumes the process is 64-bit. There are some subtle issues related to signal handling and memory allocation when performing this trick, but PTLsim implements workarounds to these issues.

After entering 64-bit mode if needed, the boot code passes control to PTLsim at `ptlsim_preinit_entry`. The `ptlsim_preinit()` function checks for the special `LoaderInfo` structure on the stack and in the ELF header of PTLsim as modified by the boot code; if these structures are found, PTLsim knows it is running inside the user program address space. After setting up memory management and threading, it captures any state the user process was initialized with. This state is used to fill in fields in the global `ctx` structure of class `CoreContext`: various floating point related fields and the user program entry point and original stack pointer are saved away at this point. If PTLsim is running inside a 32-bit process, the 32-bit arguments, environment and kernel auxiliary vector array (auxv) need to be converted to their 64-bit format for PTLsim to be able to parse them from normal C/C++ code. Finally, control is returned to `main()` to allow the simulator to start up normally.

### 11.1.2 Startup on 32-bit x86

The PTLsim startup process on a 32-bit x86 system is essentially a streamlined version of the process above (Section 11.1.1), since there is no need for the same PTLsim binary to support both 32-bit and 64-bit user programs. The injection process is very similar to the case where the user program is always a 32-bit program.

## 11.2 Simulator Startup

In `kernel.cpp`, the `main()` function calls `init_config()` to read in the user program specific configuration as described in Sections 13.2 and 10.3, then starts up the various other simulator subsystems.

If one of the `-excludeld` or `-startrip` options were given, a breakpoint is inserted at the RIP address where the user process should switch from native mode to simulation mode (this may be at the dynamic linker entry point by default).

Finally, `switch_to_native_restore_context()` is called to restore the state that existed before PTLsim was injected into the process and return to the dynamic linker entry point. This may involve switching from 64-bit back to 32-bit mode to start executing the user process natively as discussed in Section 11.1.

After native execution reaches the inserted breakpoint thunk code, the code performs a 32-to-64-bit long jump back into PTLsim, which promptly restores the code underneath the inserted breakpoint thunk. At this point, the `switch_to_sim()` function in `kernel.cpp` is invoked to actually begin the simulation. This is done by calling `simulate()` in `ptlsim.cpp`.

At some point during simulation, the user program or the configuration file may request a switch back to native mode for the remainder of the program. In this case, the `switch_to_native_restore_context()` function gets called to save the statistics data store, map the PTLsim internal state back to the x86 compatible external state and return to the 32-bit or 64-bit user code, effectively removing PTLsim from the loop.

While the real PTLsim user process is running, the original PTLsim injector process simply waits in the background for the real user program with PTLsim inside it to terminate, then returns its exit code.

## 11.3 Address Space Simulation

PTLsim maintains the `AddressSpace` class as global variable `asp` (see `kernel.cpp`) to track the attributes of each page within the virtual address space. When compiled for x86-64 systems, PTLsim uses Shadow Page Access Tables (SPATs), which are essentially large two-level bitmaps. Since pages are 4096 bytes in size, each 64 kilobyte chunk of the bitmap can track 2 GB of virtual address space. In each SPAT, each top level array entry points to a chunk mapping 2 GB, such that with 131072 top level pointers, the full 48 bit virtual address space can typically be mapped with under a megabyte of SPAT chunks, assuming the address space is sparse.

When compiled for 32-bit x86 systems, each SPAT is just a 128 KByte bitmap, with one bit for each of the 1048576 4 KB pages in the 4 GB address space.

In the AddressSpace structure, there are separate SPAT tables for readable pages (`readmap` field), writable pages (`writemap` field) and executable pages (`execmap` field). Two additional SPATs, `dtlbmap` and `itlbmap`, are used to track which pages are currently mapped by the simulated translation lookaside buffers (TLBs); this is discussed further in Section 25.4.

When running in native mode, PTLsim cannot track changes to the process memory map made by native calls to `mmap()`, `munmap()`, etc. Therefore, at every switch from native to simulation mode, the `resync_with_process_maps()` function is called. This function parses the `/proc/self/maps` metafile maintained by the kernel to build a list of all regions mapped by the current process. Using this list, the SPATs are rebuilt to reflect the current memory map. This is absolutely critical for correct operation, since during simulation, speculative loads and stores will only read and write memory if the appropriate SPAT indicates the address is accessible to user code. If the SPATs become out of sync with the real memory map, PTLsim itself may crash rather than simply marking the offending load or store as invalid. The `resync_with_process_maps()` function (or more specifically, the `mqueryall()` helper function) is fairly kernel version specific since the format of `/proc/self/maps` has changed between Linux 2.6.x kernels. New kernels may require updating this function.

## 11.4 Debugging Hints

When adding or modifying PTLsim, bugs will invariably crop up. Fortunately, PTLsim provides a trivial way to find the location of bugs which silently corrupt program execution. Since PTLsim can transparently switch between simulation and native mode, isolating the divergence point between the simulated behavior and what a real reference machine would do can be done through binary search. The `-stopinsns` configuration option can be set to stop simulation before the problem occurs, then incremented until the first x86 instruction to break the program is determined.

The out of order simulator (`ooocore.cpp`) includes extensive debugging and integrity checking assertions. These may be turned off by default for improved performance, but they can be easily re-enabled by defining the `ENABLE_CHECKS` symbol at the top of `ooocore.cpp`, `ooopipe.cpp` and `oooexec.cpp`. Additional check functions are in the code but commented out; these may be used as well.

You can also debug PTLsim with `gdb`, although the process is non-standard due to PTLsim's co-simulation architecture:

- Start PTLsim on the target program like normal. Notice the `Thread N is running in XX-bit mode` message printed at startup: this is the PID you will be debugging, not the "`ptlsim`" process that may also be running.
- Start GDB and type "`attach 12345`" if *12345* was the PID listed above
- Type "`symbol-file ptlsim`" to load the PTLsim internal symbols (otherwise gdb only knows about the benchmark code itself). You should specify the full path to the PTLsim executable here.
- You're now debugging PTLsim. If you run the "`bt`" command to get a backtrace, it should show the PTLsim functions starting at address 0x70000000.

If the backtrace does not display enough information, go to the `Makefile` and enable the "no optimization" options (the "-O0" line instead of "-O99") since that will make more debugging information available to you.

The "`-pause-at-startup` *seconds*" configuration option may be useful here, to give you time to attach with a debugger before starting the simulation.

## 11.5 Timing Issues

PTLsim uses the `CycleTimer` class extensively to gather data about its own performance using the CPU's timestamp counter. At startup in `superstl.cpp`, the CPU's maximum frequency is queried from the appropriate Linux kernel sysfs node (if available) or from `/proc/cpuinfo` if not. Processors which dynamically scale their frequency and voltage in response to load (like all Athlon 64 and K8 based AMD processors) require special handling. It is assumed that the processor will be running at its maximum frequency (as reported by sysfs) or a fixed frequency (as reported by `/proc/cpuinfo`) throughout the majority of the simulation time; otherwise the timing results will be bogus.

## 11.6 External Signals and PTLsim

PTLsim can be forced to switch between native mode and sequential mode by sending it standard Linux-style signals from the command line. If your program is called "myprogram", start it under PTLsim and run this command from another terminal:

```
killall -XCPU myprogram
```

This will force PTLsim to switch between native mode and simulation mode, depending on its current mode. It will print a message to the console and the logfile when you do this. The initial mode (native or simulation) is determined by the presence of the `-trigger` option: with `-trigger`, the program starts in native mode until the trigger point (if any) is reached.

# Part III

# PTLsim/X: Full System SMP/SMT Simulation

# Chapter 12

# Background

## 12.1 Virtual Machines and Full System Simulation

Full system simulation and virtualization has been around since the dawn of computers. Typically *virtual machine* software is used to run *guest* operating systems on a physical *host* system, such that the guest believes it is running directly on the bare hardware. Modern full system simulators in the x86 world can be roughly divided into two groups (this paper does not consider systems for other instruction sets).

*Hypervisors* execute most unprivileged instructions on the native CPU at full speed, but trap privileged instructions used by the operating system kernel, where they are emulated by hypervisor software so as to maintain isolation between virtual machines and make the virtual machine nearly indistinguishable from the real CPU. In some cases (particularly on x86), additional software techniques are needed to fully hide the hypervisor from the guest OS.

- *Xen* [6, 7, 5, 8, 9, 2] represents the current state of the art in this field; it will be described in great detail later on.
- *VMware* [12] is a very well known commercial product that allows unmodified x86 operating systems to run inside a virtual machine. Because the x86 instruction set is not fully virtualizable, VMware must employ x86-to-x86 binary translation techniques on kernel code (but not user mode code) to make the virtual CPU indistinguishable from the real CPU for compatibility reasons. These translations are typically cached in a hidden portion of the guest address space to improve performance compared to simply interpreting sensitive x86 instructions. While this approach is sophisticated and effective, it exacts a heavy performance penalty on I/O intensive workloads [9]. Interestingly, the latest microprocessors from Intel and AMD include hardware features (Intel VT [15], AMD SVM [16]) to eliminate the binary translation and patching overhead. Xen fully supports these technologies to allow running Windows and other OS's at full speed, while VMware has yet to include full support.

  VMware comes in two flavors. ESX is a true hypervisor that boots on the bare hardware underneath the first guest OS. GSX and Workstation use a userspace frontend process containing all virtual device drivers and the binary translator, while the *vmmon* kernel module (open source in the Linux version) handles memory virtualization and context switching tasks similar to Xen.

- Several other products, including Virtual PC and Parallels, provide features similar to VMware using similar technology.

- *KVM* (Kernel Virtual Machine) is a new hypervisor infrastructure built into all Linux kernels after 2.6.19. It depends on the hardware virtualization extensions (Intel VT and AMD SVM) built into modern x86 chips, whereas Xen and VMware also support running on older processors without special hardware support. KVM is an attractive foundation for future virtual machine development since it's built into Linux (so it requires far less setup work than Xen or VMware) and provides excellent performance.

Unlike hypervisors, *simulators* perform cycle accurate execution of x86 instructions using interpreter software, without running any guest instructions on the native CPU.

- *Bochs* [11] is the most well known open source x86 simulator; it is considered to be a nearly RTL (register transfer language) level description of every x86 behavior from legacy 16-bit features up through modern x86-64 instructions. *Bochs* is very useful for the functional validation of real x86 microprocessors, but it is very slow (around 5-10 MHz equivalent) and is not useful for implementing cycle accurate models of modern uop-based out of order x86 processors (for instance, it does not model caches, memory latency, functional units and so on).

- *QEMU* [10] is similar in purpose to VMware, but unlike VMware, it supports multiple CPU host and guest architectures (PowerPC, SPARC, ARM, etc). QEMU uses binary translation technology similar to VMware to hide the hypervisor's presence from the guest kernel. However, due to its cross platform design, both kernel and user code is passed through x86-to-x86 binary translation (even on x86 platforms) and stored in a translation cache. Interestingly, Xen uses a substantial amount of QEMU code to model common hardware devices when running unmodified operating systems like Windows, but Xen still uses its own hardware-assisted technology to actually achieve virtualization. QEMU supports a proprietary hypervisor module to add VMware's and Xen's ability to run user mode code natively on the CPU to reduce the performance penalty; hence it is also in the hypervisor category.

- *Simics* [13] is a commercial simulation suite for modeling both the functional aspects of various x86 processors (including vendor specific extensions) as well as user-designed plug-in models of real hardware devices. It is used extensively in industry for modeling new hardware and drivers, as well as firmware level debugging. Like QEMU, Simics uses x86-to-x86 binary translation to instrument code at a very low level while achieving good performance (though noticeably slower than a hypervisor provides). Unlike QEMU, Simics is fully extensible and supports a huge range of real hardware models, but it is not possible to add cycle accurate simulation features below the x86 instruction level, making it less useful to microarchitects (both because of technical considerations as well as its status as a closed source product).

- *SimNow* [14] is an AMD simulation tool used during the design and validation of AMD's x86-64 hardware. Like Simics, it is a functional simulator only, but it models a variety of AMD-built hardware devices. SimNow uses x86-to-x86 binary translation technology similar to Simics and QEMU to achieve good performance. Because SimNow does not provide cycle accurate timing data, AMD uses its own TSIM trace-based simulator, derived from the K8 RTL, to do actual validation and timing studies. SimNow is available for free to the public, albeit as closed source.

All of these tools share one common disadvantage: they are unable to model execution at a level below the granularity of x86 instructions, making them unsuitable to microarchitects. PTLsim/X seeks to fill this void by allowing extremely detailed uop-level cycle accurate simulation of x86 and x86-64 microprocessor cores, while simultaneously delivering all the performance benefits of true native-mode hypervisors like Xen, selective binary translation based hypervisors like VMware and QEMU, and the detailed hardware modeling capabilities of Bochs and Simics.

## 12.2 Xen Overview

Xen [7, 6, 5, 8, 9, 2] is an open source x86 virtual machine monitor, also known as a *hypervisor*. Each virtual machine is called a "domain", where domain 0 is privileged and accesses all hardware devices using the standard drivers; it can also create and directly manipulate other domains. Guest domains typically do not have hardware access do not have this access; instead, they relay requests back to domain 0 using Xen-specific virtual device drivers. Each guest can have up to 32 VCPUs (virtual CPUs). Xen itself is loaded into a reserved region of physical memory before loading a Linux kernel as domain 0; other operating systems can run in guest domains. Xen is famous for having essentially zero overhead due to its unique and well planned design; it's possible to run a normal workstation or server under Xen with full native performance.

Under Xen's "paravirtualized" mode, the guest OS runs on an architecture nearly identical to x86 or x86-64, but a few small changes (critical to preserving native performance levels) must be made to low-level kernel code, similar in scope to adding support for a new type of system chipset or CPU manufacturer (e.g. instead of an AMD x86-64 on an nVidia chipset, the kernel would need to support a Xen-extended x86-64 CPU on a Xen virtual "chipset"). These changes mostly concern page tables and the interrupt controller:

- Paging is always enabled, and any physical pages (called "machine frame numbers", or MFNs) used to form a page table must be marked read-only (a.k.a. "pinned") everywhere. Since the processor can only access a physical page if it's referenced by some page table, Xen can guarantee memory isolation between domains by forcing the guest kernel to replace any writes to page table pages with special *mmu_update()* hypercalls (a.k.a. system calls into Xen itself). Xen makes sure each update points to a page owned by the domain before updating the page table. This approach has essentially zero performance loss since the guest kernel can read its own page tables without any further indirections (i.e. the page tables point to the actual physical addresses), and hypercalls are only needed for batched updates (e.g. validating a new page table after a *fork()* requires only a single hypercall).

  - Xen also supports *pseudo-physical* pages, which are consecutively numbered from 0 to some maximum (i.e. 65536 for a 256 MB domain). This is required because most kernels (including Linux and Windows) do not support "sparse" (discontiguous) physical memory ranges very well (remember that every domain can still address every physical page, including those of other domains - it just can't access all of them). Xen provides pseudo-to-machine (P2M) and machine-to-pseudo (M2P) tables to do this mapping. However, the physical page tables still continue to reference physical addresses and are fully visible to the guest kernel; this is just a convenience feature.
  - Xen can save an entire domain to disk, then restore it later starting at that checkpoint. Since Xen tracks every read-only page that's part of some page table, it can restore domains even if the original physical pages are now used by something else: it automatically remaps all MFNs in every page table page it knows about (but the guest kernel must never store machine page numbers outside of page table pages - it's the same concept as in garbage collection, where pointers must only reside in the obvious places).
  - Xen can migrate running domains between machines by tracking which physical pages become dirty as the domain executes. Xen uses *shadow page tables* for this: it makes copy-on-write duplicates of the domain's page tables, and presents these internal tables to the CPU, while the guest kernel still thinks it's using the original page tables. Once the migration is complete, the shadow page tables are merged back into the real page tables (as with a save and restore) and the domain continues as usual.
  - The memory allocation of each domain is elastic: the domain can give any free pages back to Xen via the "balloon" mechanism; these pages can then be re-assigned to other domains that need more memory (up to a per-domain limit).
  - Domains can share some of their pages with other domains using the *grant mechanism.* This is used for zero-copy network and disk I/O between domain 0 and guest domains.

- Interrupts are delivered using an *event channel* mechanism, which is functionally identical to the IO-APIC hardware on the bare CPU (essentially it's a "Xen APIC" instead of the Intel and AMD models already supported by the guest kernel). Xen sets up a *shared info* page containing bit vectors for masked and pending interrupts (just like an APIC's memory mapped registers), and lets the guest kernel register an event handler function. Xen then does an upcall to this function whenever a virtual interrupt arrives; the guest kernel manipulates the mask and pending bits to ensure race-free notifications. Xen automatically maps physical IRQs on the APIC to event channels in domain 0, plus it adds its own virtual interrupts (for the usual timer and a Xen-specific notification port; use *cat /proc/interrupts* on a Linux system under Xen to see this). When the guest domain has multiple VCPUs, interprocessor interrupts (IPIs) are done through the Xen event controller in a manner identical to hardware IPIs.

  - Xen is unique in that PCI devices can be assigned to any domain, so for instance each guest domain could have its own dedicated PCI network card and disk controller - there's no need to relay requests back to domain 0 in this configuration, although it only works with hardware that supports IOMMU virtualization (otherwise it's a security risk, since DMA can be used to bypass Xen's page table protections).

- Xen provides the guest with additional timers, so it can be aware of both "wall clock" time as well as execution time (since there may be gaps in the latter as other domains use the CPU); this lets it provide a smooth interactive experience in a way systems like VMware cannot. The timers are delivered as virtual interrupt events.

- All other features of the paravirtualized architecture perfectly match x86. The guest kernel can still use most x86 privileged instructions, such as *rdmsr*, *wrmsr*, and control register updates (which Xen transparently intercepts and validates), and in domain 0, it can access I/O ports, memory mapped I/O, the normal x86 segmentation (GDT and LDT) and interrupt mechanisms (IDT), etc. This makes it possible to run a normal Linux distribution, with totally unmodified drivers and software, at full native speed (we do just this on all our development workstations and servers). Benchmarks [9] have shown Xen to have ~2-3% performance decrease relative to a traditional Linux kernel, where as VMware and similar solutions yield a 20-70% decrease under heavy I/O.

Xen also supports "HVM" (hardware virtual machine) mode, which is equivalent to what VMware [12], QEMU [10], Bochs [11] and similar systems provide: nearly perfect emulation of the x86 architecture and some standard peripherals. The advantage is that an uncooperative guest OS never knows it's running in a virtual machine: Windows XP and Mac OS X have been successfully run inside Xen in this mode. Unfortunately, this mode has a well known performance cost, even when Xen leverages the specialized hardware support for full virtualization in newer Intel [15] and AMD [16] chips. The overhead comes from the requirement that the hypervisor still trap and emulate all sensitive instructions, whereas paravirtualized guests can intelligently batch together requests in one hypercall and can avoid virtual device driver overhead.

# Chapter 13

# Getting Started with PTLsim/X

*NOTE:* This part of the manual is relevant only if you are using the full-system PTLsim/X. If you are looking for the userspace-only version, please skip this entire part and read Part II instead.

***WARNING:*** PTLsim/X assumes fairly high level of familiarity with Xen and the Linux kernel. If you have never compiled your own Linux kernel or if you are not yet running Xen or are unsure how to create and use domains, **STOP NOW** and become familiar with Xen itself before attempting to use PTLsim/X. The following sections all assume you are familiar with Xen, at least from a system administration perspective. We cannot provide support for Xen-related issues unless they are caused by PTLsim.

## 13.1 Building PTLsim/X

**Prerequisites:**

- PTLsim/X requires a **modern 64-bit x86-64 machine**. This means an AMD Athlon 64 / Opteron / Turion or an Intel Pentium 4 (specifically with EM64T) or Intel Core 2**.** We do *not* plan to offer a 32-bit version of PTLsim/X due to the technical deficiencies in 32-bit x86 that make it difficult to properly implement a full system simulator with all of PTLsim's features. Besides, 64-bit hardware is now the standard (in some cases the only option) from all the major x86 processor vendors and is very affordable.

- The 64-bit requirement *only* applies to the host system running PTLsim/X. Inside the virtual machine, you are still free to use standard 32-bit Linux distributions, applications and so forth under PTLsim/X

- PTLsim/X assumes you have root access to your machine. The PTLsim/X hypervisor runs below Linux itself, so you must use a Xen compatible kernel in domain 0 (more on this later).

- We *highly recommend* you use a Linux distribution already designed to work with Xen 3.x. We use SuSE 10.2 and highly recommend it; most other distributions now support Xen. This requirement only applies to domain 0 - the virtual machines you'll be running can use any distribution and do not even need to know about Xen at all (other than the kernel, which must support Xen hypercalls and block/network drivers).
- We have successfully built PTLsim/X with gcc 4.1.x+ (gcc 4.0.x has documented bugs affecting some of our code).

**Quick Start Steps:**

All files listed below can be downloaded from `http://www.ptlsim.org/download.php`.

***IMPORTANT:*** The instructions below refer to specific versions of various files (i.e., Xen hypervisor, Linux kernel, etc.). We regularly update the versions of these files, and newer PTLsim/X versions may not work correctly with older kernel and/or hypervisor versions (i.e. the versions should be matching). The following instructions are therefore for informational purposes only; always check the PTLsim web site's download page for the latest versions of these files. The following versions are correct as of September 20th, 2007

1. **Set up Xen with PTLsim/X extensions:**
   - **Download** our modified Xen source tree (`xen-3.1-ptlsim.tar.bz2`) from `http://www.ptlsim.org/downl` This is the easiest way to make sure you have the correct PTLsim-compatible version of Xen with all patches pre-applied.
     - We also provide `ptlsim-xen-hypervisor.diff` in case you want to manually apply the patches to a development version of Xen; the patches are fairly simple and can be adapted as needed.
   - **Build and install** both the Xen hypervisor and the userspace Xen tools:
     - In `xen-3.1-ptlsim/xen`, run `make.` You can optionally copy the compiled Xen hypervisor (in `xen/xen`) somewhere else (such as wherever your kernel and initrd files are stored).
     - In `xen-3.1-ptlsim/tools`, run `make`, then run `make install`.
   - **Download** our sample kernel and modules (`linux-2.6.22.6-mtyrel-64bit-xen.tar.bz2`) and extract in the root directory (via `tar jxvf linux-2.6.22.6-mtyrel-64bit-xen.tar.bz2`) to create `/lib/modules/2.6.22.6-mtyrel-64bit-xen/....`
     - This is a SMP kernel based on 2.6.22.6 with the Xen patches maintained by SuSE Linux. The complete source is in `linux-2.6.22-mtyrel-source.tar.gz`, if you want to recompile it.
     - This is just a sample kernel we use - PTLsim/X should work even if you use the Xen-compatible kernel shipped with your Linux distribution of choice. However, we recommend you run this same kernel in domain 0 as well as in the target domain under simulation, simply because we know it works correctly and has all the latest Xen patches.

    - In addition, our kernels feature the ability to create Xen checkpoints and initiate PTLsim actions from within the domain by writing to `/proc/xen/checkpoint` and `/proc/xen/ptlsim`, respectively. The major changes are in `linux-2.6.22-mtyrel/patches.mty/linux` if you want to apply them to a different kernel or learn how they work.

- **Activate** the new Xen hypervisor and kernel:
    - Install the new kernel and Xen hypervisor in a manner specific to your distribution. While we cannot provide instructions for every distribution, on SuSE, you need to run mkinitrd to collect the required boot drivers like this:

      ```
      mkinitrd -k /lib/modules/2.6.22.6-mtyrel-64bit-xen/linux
               -i /lib/modules/2.6.22.6-mtyrel-64bit-xen/initrd
               -M /lib/modules/2.6.22.6-mtyrel-64bit-xen/System.map
      ```

      *IMPORTANT:* All parts of this command should be on a single line (this manual makes long lines difficult to show)
    - Edit the GRUB bootloader configuration (usually in `/boot/grub/menu.lst` on most distributions) to specify the new Xen-enabled kernel and hypervisor. The first entry should be similar to:

      ```
      title Linux 2.6.22.6-mtyrel-64bit-xen
      kernel (hd0,0)/project/xen-3.1-ptlsim/xen/xen console=vga
      module (hd0,0)/lib/modules/2.6.22.6-mtyrel-64bit-xen/linux root=/dev/...
      module (hd0,0)/lib/modules/2.6.22.6-mtyrel-64bit-xen/initrd
      ```

      Obviously you may need to adjust the file locations, if you're booting from a different hard drive or compiled Xen in a location other than `/project/xen-3.1-ptlsim`.

- **Reboot**, and make sure the PTLsim/X extensions to Xen are actually running: "`cat /sys/hypervisor/properties/capabilities`" should list "`ptlsim`". If this file doesn't exist, you're not running under Xen at all.

2. **Set up sample virtual machine and disk images:**

    - **Download** our pre-configured example disk image (`ptlsim-disk-image-example.tar.bz2`) and uncompress with `tar jxvf ptlsim-disk-image-example.tar.bz2`. The sample scripts inside this archive assume that the files were extracted into `/project/ptlsim-disk-image-example`.
        - We recommend placing this disk image on a local hard disk rather than NFS. However, if you're running Cluster NFS and/or are using the `no_root_squash` NFS option, it's perfectly fine if you put the disk image on an NFS volume.
    - You already downloaded our Xen-compatible kernel above.
    - The disk image archive contains a sample Xen configuration file (sample-xen-domain) and some helpful scripts (e.g. `run-domain`, `restore-domain`, etc.)
    - Make sure you can create this domain "`xm create sample-xen-domain -c`". You should get a console with the text "Welcome to the PTLsim Demo Machine".

3. **Setup PTLsim itself:**

- Download the stable version of PTLsim from our web site (in `ptlsim-2007xxxx-rXXX.tar.gz`) and unpack this file to create the `ptlsim` directory.
- Edit the PTLsim `Makefile` and uncomment the "`PTLSIM_HYPERVISOR=1`" line to enable full system PTLsim/X support.
- Run `make`.
  - If the build process complains about missing header files, make sure `/usr/include/xen` is a symlink to `/project/xen-3.1-ptlsim/tools/libxc/xen` (or wherever you put the PTLsim-modified `xen-3.1-ptlsim` tree you downloaded). Delete `/usr/include/xen` beforehand if needed.

## 13.2 Running PTLsim

PTLsim is run in domain 0 as root, for instance by using the "`sudo ptlsim ...`" command. The `-domain` *N* option is used to specify the domain to access. The following scenarios show by example how this is done.

## 13.3 Booting Linux under PTLsim

In the following examples, we will assume the target domain is called `ptlvm`.

Start your domain as follows:

```
sudo xm create domainname --paused
sudo xm list
sudo xm console domainname
```

The `--paused` option tells Xen to pause the domain as soon as it's created, so we can run the entire boot process under PTLsim.

The `xm list` command will print the domain ID assigned to `ptlvm`. On our test machine, the output looks like:

```
yourst [typhoon /project/ptlsim] sudo xm create ptlvm --paused; sudo xm list; sudo xm console ptlv
Using config file "ptlvm".
Started domain ptlvm
Name                                      ID Mem(MiB) VCPUs State   Time(s)
Domain-0                                   0     7877     4 r-----    137.9
ptlvm                                     21      128     1 --p---      0.0
```

You may also want to give the PTLsim domain a low priority; otherwise it may cause the system to respond slowly. This can be done by adding:

```
sudo xm sched-credit -d ptlvm -w 16
```

Open another console and start PTLsim on this domain (using the domain ID “21” given in the example above):

```
sudo ./ptlsim -domain ptlvm -logfile ptlsim.log -native
```

The resulting output:

```
//
//  PTLsim: Cycle Accurate x86-64 Full System Simulator
//  Copyright 1999-2007 Matt T. Yourst <yourst@yourst.com>
//
//  Revision 225 (2007-09-21)
//  Built Sep  21 2007 16:21:36 on tidalwave.lab.ptlsim.org using gcc-4.2
//  Running on typhoon.lab.ptlsim.org
//
Processing -domain 21 -logfile ptlsim.log -native
System Information:
  Running on hypervisor version xen-3.0-x86_64-ptlsim
  Xen is mapped at virtual address 0xffff800000000000
  PTLsim is running across 1 VCPUs:
    VCPU 0: 2202 MHz
Memory Layout:
  System:                 524208 pages,    2096832 KB
  Domain:                  32768 pages,     131072 KB
  PTLsim reserved:          8192 pages,      32768 KB
  Page Tables:               275 pages,       1100 KB
  PTLsim image:              407 pages,       1628 KB
  Heap:                     7510 pages,      30040 KB
  Stack:                     256 pages,       1024 KB
Interfaces:
  PTLsim page table:      282898
  Shared info mfn:          4056
  Shadow shinfo mfn:      295164
  PTLsim hostcall:                event channel    3
  PTLsim upcall:                  event channel    4
  Switched to native mode
```

Back in the Xen console for the domain, you’ll see the familiar Linux boot messages:

```
Bootdata ok (command line is  nousb noide root=/dev/hda1 xencons=ttyS console=ttyS0)
Linux version 2.6.18-mtyrel-k8-64bit-xen (yourst@tidalwave) (gcc version 4.1.0 (SUSE Linux)) #2 Su
BIOS-provided physical RAM map:
 Xen: 0000000000000000 - 0000000008800000 (usable)
No mptable found.
Built 1 zonelists.  Total pages: 34816
Kernel command line:  nousb noide root=/dev/hda1 xencons=ttyS console=ttyS0
Initializing CPU#0
```

```
PID hash table entries: 1024 (order: 10, 8192 bytes)
Xen reported: 2202.808 MHz processor.
Console: colour dummy device 80x25
Dentry cache hash table entries: 32768 (order: 6, 262144 bytes)
Inode-cache hash table entries: 16384 (order: 5, 131072 bytes)
Software IO TLB disabled
Memory: 123180k/139264k available (2783k kernel code, 7728k reserved, 959k data, 184k init)
Calibrating delay using timer specific routine.. 4407.14 BogoMIPS (lpj=2203570)
...
NET: Registered protocol family 1
NET: Registered protocol family 17
VFS: Mounted root (ext2 filesystem) readonly.
Welcome to the PTLsim demo machine!
root [ptlsim /] cat /proc/cpuinfo
```

You'll notice how we specified the "`-native`" option to speed up the boot process by running all code on the real CPU rather than PTLsim's synthetic CPU model. Booting Linux within PTLsim is slow since the kernel often executes several billion instructions before finally presenting a command line.

## 13.4 Running Simulations: PTLctl

At this point, we would like to start an actual simulation run. For purposes of illustration, this run is composed of three actions:

- Simulate 100 million x86 instructions using PTLsim's out of order superscalar model
- Simulate another 100 million using PTLsim's sequential model. The sequential model is much faster than the out of order superscalar model, so it's useful for testing and debugging functional issues, as well as simply interacting with the domain. However, it does not collect any cycle accurate timing data. Section 9.4 gives more information on the sequential model.
- Return to native mode

In the first example, we will start this run from within the running domain using `ptlctl` (PTLsim controller), a program supplied with PTLsim. PTLctl is actually an example program showing the use of PTLsim hypercalls ("PTL calls"), special x86 instructions that can be used to control a domain's own simulation. More information on the PTLcall API is in Section 14.4.

To conduct this simulation, the `ptlctl` command is used *within* the running virtual machine (by typing it at the domain's console); it is not run on the host system at all:

```
root [ptlsim /] tar zc usr lib | tar ztv > /tmp/allfiles.txt &
[1] 775
root [ptlsim /] ptlctl -core ooo -stopinsns 100m -run : -core seq -stopinsns 200m -run : -native
Sending flush and command list to PTLsim hypervisor:
  -core ooo -stopinsns 100m -run
  -core seq -stopinsns 200m -run
  -native
PTLsim returned rc 0
root [ptlsim /]
```

The first command simply runs several CPU-intensive multi-threaded processes in the background for simulation purposes (in this case, compressing and uncompressing files in the virtual machine's filesystem).

The second `ptlctl` command submits the three simulation actions to PTLsim, separated by colons (":").

At the PTLsim console, the following output is produced (the cycle counters will update regularly):

```
...
Breakout request received from native mode
  Switched to simulation mode
Returned from switch to native: now back in sim
Processing -core ooo -stopinsns 100m -run
  Completed       75258330 cycles,      100000000 commits:    461819 cycles/sec,    795201, insns/
Processing -core seq -stopinsns 200m -run
  Completed      200000000 cycles,      200000000 commits:   6941302 cycles/sec,   6941302, insns/
Processing -native
  Switched to native mode
```

Notice how the command list is always terminated by a final simulation action (in this case, `-native`). If the command list only had one simulation run with a fixed duration, once that simulation ended, the domain would freeze, since PTLsim would pause until another command arrived. However, since the domain is frozen, the next command would *never* arrive: there is no way to execute the `ptlctl` program a second time if the domain is stopped. To avoid this sort of deadlock, `ptlctl` lets the user atomically submit batches of multiple commands as shown ahove.

This powerful capability allows "self-directed" simulation scripts (i.e. standard shell scripts), in which `ptlctl` is run immediately before starting a benchmark program, then `ptlctl` is run again after the program exits to end the simulation and switch back to native mode.

## 13.5 PTLsim/X Options

In Section 10.3, the configuration options common to both userspace PTLsim and full system PTLsim/X wer listed. PTLsim/X also introduces a number of special options only applicable to full system simulation:

Actions:

- `-run`

  Start a simulation run, using the core model specified by the `-core` option (the default core is "`ooo`").

- `-stop`

  Stop the simulation run currently in progress, and wait for further commands. This is generally issued from another console window.

- `-native`

  Switch the domain to native mode.

- `-kill`

  Kill the domain. This is equivalent to "`xm destroy`", but it also allows PTLsim to perform cleanup actions and flush all files before exiting.

## 13.6 Live Updates of Configuration Options

PTLsim/X provides the ability to send commands and modify configuration options in the running simulation from another console on the host system. This is different from how the `ptlctl` program is used inside the target domain to script simulations: in this case, the commands are submitted asynchronously from the host system.

For instance,

```
sudo ptlsim -native -domain ptlvm
```

will immediately switch the target domain back to native mode.

To reset the log level in the middle of a simulation run, use the following:

```
sudo ptlsim -domain ptlvm -loglevel 99 : -run
ptlsim: Sending request '-domain ptlvm -loglevel 99 : -run' to domain 12...OK
```

(This is an example only! Using `-loglevel 99` will create huge log files).

Most options (such as `-loglevel`, `-stoprip`, etc.) can be updated at any time in this manner.

To end a simulation currently in progress, use this:

```
sudo ptlsim -domain ptlvm -kill
```

This will force PTLsim to cleanly exit.

## 13.7 Command Scripts

PTLsim supports *command scripts*, in which a file containing a list of commands is passed on the PTLsim command line as follows:

```
sudo ./ptlsim -domain name @ptlvm.cmd
```

where `ptlvm.cmd` (specified following the "`@`" operator) contains the example lines:

```
# Configuration options:
-logfile ptlsim.log -loglevel 4 -stats ptlsim.stats -snapshot-cycles 10m
# Run the simulation
-core seq -run -stopinsns 20m
-core ooo -run -stopinsns 100m
-native      # All done (switch to native mode)
```

These commands are executed by PTLsim one at a time, waiting until the previous command completes before starting the next. Notice the use of comments (starting with "`#`"), and how configuration options can be spread across lines if desired. This mode is very useful for specifying breakpoints using `-stoprip` and similar options; when the target RIP is reached, the simulation stops and the next command in the command list is executed.

Command scripts can be nested (i.e. a script can itself include other scripts using `@scriptname`). When multiple commands are given on the command line separated by colons ("`:`"), any `@scriptname` clauses are processed after all other commands on the command line.

## 13.8 Working with Checkpoints

We maintain a tutorial on how to set up checkpoints and perform advanced checkpointing techniques at `http://www.ptlsim.org/capswiki/index.php/SPEC_2006`. Note that this address is subject to change.

Xen provides the ability to capture the state of a domain into a *checkpoint file* stored on disk. PTLsim can leverage this capability to start simulation from a checkpoint, avoiding the need to go through the entire boot process, and allowing precisely reproducable results across multiple simulation runs.

To create a checkpoint, boot the domain in native mode without PTLsim running, and bring the domain to the point where you would like to begin simulation. Then, in another console, run:

```
sudo xm save ptlvm /tmp/ptlvm.img
```

If you're using our sample disk images, this command will pause until you do the following from *within* the domain:

```
echo checkpoint > /proc/xen/checkpoint
```

This facility allows very precise checkpoint placement, even by writing to this special file from within a benchmark.

To restore the domain to that checkpoint, run:

```
sudo xm restore /tmp/ptlvm.img --paused
sudo xm list
sudo xm console ptlvm
```

PTLsim can then be started in the normal manner, by specifying `-domain domainname`. If the checkpoint was made while the domain waited for input (e.g. at a shell command line), you may have to press a few keys to get any response from its console.

To exit PTLsim, use "`sudo ptlsim -kill -domain X`" from another console. To abort PTLsim immediately, use Ctrl+C on the ptlsim process, then type "`xm kill ptlvm`" to destroy the domain.

## 13.9 The Nature of Time

Full system simulation poses some difficult philosophical questions about the nature of time itself and the relativistic phenomenon of "time dilation". Specifically, if a simulator runs X times slower than the native CPU, both external interrupts and timer interrupts should theoretically be generated X times slower than in the real world. This is critical for obtaining accurate simulation results: for events like network traffic, if a real network device fed interrupts into the domain in realtime, and the simulator injected these interrupts into the simulation at the same rate, they would appear to arrive thousands of times faster than any physical network interface could deliver them. This can easily result in a livelock situation not possible in a real machine; at the very least it will deliver misleading performance results.

On the other hand, interacting with a domain running at the "correct" rate according to its own simulated clock can be unpleasant for users. For instance, if the "`sleep 1`" command is run in a Linux domain under PTLsim, instead of sleeping for 1 second of wall clock time (as perceived by the user), the domain will wait until 1 billion cycles have been fully simulated (assuming the simulated processor frequency is 1 GHz). This is because PTLsim keys interrupt delivery and all timers to the simulated cycle number in which the interrupt should arrive (based on the core clock frequency). In addition to being annoying, this behavior will massively confuse network applications that rely on precise timing information: a TCP/IP endpoint outside the domain will not expect packets to arrive thousands of times slower than its own realtime clock expects, resulting in retransmissions and timeouts that would never occur if both endpoints were inside the same "time dilated" domain.

Rather than attempt to solve this philosophical dilemma, PTLsim allows users to choose the options that best suit their simulation accuracy needs. The following options control the notion of time inside the simulation:

- `-corefreq` *Hz*

  Specify the CPU core frequency (in Hz) reported to the domain. To specify a 2.4 GHz core, use "`-corefreq 2400m`". This option is used to calculate the number of cycles between timer interrupts, as described below.

  *NOTE:* If you plan on switching the domain between simulation and native mode, we strongly recommend avoiding this option, to allow the host machine frequency to match the simulated frequency.

- `-timerfreq` *Hz*

  Specify the timer interrupt frequency in interrupts per second. By default, 100 interrupts per second are used, since this is the standard for Linux kernels.

  ***Hint:*** if keyboard interaction with the domain seems slow or sluggish, this is because Linux only flushes console buffers to the screen at every clock tick. Specifying `-timerfreq 1000` will greatly improve interactive response at the expense of more interrupt overhead.

- `-pseudo-rtc`

  By default, the realtime clock reported to the domain is the current time of day. This option forces the clock to reset to whatever time the domain's checkpoint (if any) was created. This may allow better cycle accurate reproducibility of random number generators, for instance.

- `-realtime`

  PTLsim normally delivers all interrupts at the time dilated rate, as described above. While this provides the most realistic simulation accuracy, it may be undesirable for some applications, particularly in networking. The `-realtime` option delivers external interrupts to the domain as soon as they arrive at PTLsim's interrupt handler; they are not deferred. The realtime clock reported to the domain is also not dilated; it is locked to the current wall clock time. This option does not affect the timer interrupt frequency; use the `-timerfreq` option to directly manipulate this.

- `-maskints`

  Do not allow *any* external interrupts or events to reach the domain; only the timer interrupt is delivered at the specified rate by PTLsim. This mode is necessary to provide guaranteed reproducable cycle accurate behavior across runs; it eliminates almost all non-deterministic events (like outside device interrupts) from the simulation. However, it is not very practical, since disk and network access is impossible in this mode (since the Xen disk and network drivers could never wake up the domain when data arrives). This mode is most useful for debugging starting at a checkpoint, or when using a ramdisk with pre-scripted boot actions.

## 13.10 Other Options

PTLsim/X has a few additional options related to full system simulation:

- `-reservemem` *M*

  Reserves *M* megabytes of physical memory for PTLsim and its translation cache. The default is 32 MB; the valid range is from 16 MB to 512 MB. See Chapter 14 for details.

All other options in Section 10.3 (unless otherwise noted) are common to both userspace PTLsim and full system PTLsim/X.

# Chapter 14

# PTLsim/X Architecture Details

The following sections provide insight into the internal architecture of full system PTLsim/X, and how a simulator is built to run on the bare hardware. It is not necessary to understand this information to work with or customize machine models in PTLsim, but it may still be fascinating to those working with the low level infrastructure components.

## 14.1 Basic PTLsim/X Components

PTLsim/X works in a conceptually similar manner to the normal userspace PTLsim: the simulator is "injected" into the target user process address space and effectively becomes the CPU executing the process. PTLsim/X extends this concept, but instead of a process, the core PTLsim code runs on the bare hardware and accesses the same physical memory pages owned by the guest domain. Similarly, each VCPU is "collapsed" into a context structure within PTLsim when simulation begins; each context is then copied back onto the corresponding physical CPU context(s) when native mode is entered.

PTLsim/X consists of three primary components: the modified Xen hypervisor, the PTLsim monitor process, and the PTLsim core.

### 14.1.1 Xen Modifications

The Xen hypervisor requires some modifications to work with PTLsim. Specifically, several new major hypercalls were added:

- `XEN_DOMCTL_contextswap` atomically swaps all context information in all VCPUs of the target domain, saving the old context and writing in a new context. In addition to per-VCPU data (including all registers and page tables), the shared info page is also swapped. This is done as a hypercall so as to eliminate race conditions between the hypervisor, PTLsim monitor process in domain 0 and the target domain. The domain is first de-scheduled from all physical CPUs in the host system, the old context is saved, the new context is validated and written, and finally the paused domain wakes up to the new context.

- `MMUEXT_GET_GDT_TEMPLATE` gets the x86 global descriptor table (GDT) page Xen transparently maps into the `FIRST_RESERVED_GDT_PAGE gdt_frames[]` slot. PTLsim needs this data to properly resolve segment references.

- `MMUEXT_QUERY_PAGES` queries the page type and reference count of a given guest MFN.

- `VCPUOP_set_breakout_insn_action` tells the hypervisor about a special *breakout instruction*. This is a normally undefined x86 instruction that the `ptlctl` program (and PTL calls from user code) can use to request services from PTLsim. The hypervisor uses the x86 invalid opcode trap to intercept this instruction, and in response it may perform several actions, including pausing the domain and sending an interrupt to domain 0 for the PTLsim monitor process to receive. This is the mechanism by which a domain operating in native mode can request a switch back into simulation mode.

- `VCPUOP_set_timestamp_bias` is used to virtualize the processor timestamp counter (TSC) read by the x86 `rdtsc` instruction. This support is needed to ensure a seamless transition between simulation mode and native mode without the target domain noticing any cycles are missing. Since PTLsim runs much slower than the native CPU, a negative bias must be applied to the TSC to provide timing continuity when returning to native mode. The hypervisor will trap `rdtsc` instructions and emulate them when a bias is in effect.

These changes are provided by `ptlsim-xen-hypervisor.diff` as described in the installation instructions.

### 14.1.2 PTLsim Monitor (PTLmon)

The PTLsim monitor (*ptlmon.cpp*) is a normal Linux program that runs in domain 0 with root privileges. After connecting to the specified domain, it increases the domain's memory reservation so as to reserve a range of physical pages for PTLsim (by default, 32 MB of physical memory). PTLmon maps all these reserved pages into its own address space, and loads the real PTLsim core code into these pages. The PTLsim core is linked separately as *ptlxen.bin*, but is then linked as a binary object into the final self-contained *ptlsim* executable. PTLmon then builds page tables to map PTLsim space into the target domain. Finally, PTLmon fills in various other fields in the boot info page (including a pointer to the *Context* structures (a modified version of Xen's *vcpu_context_t*) holding the interrupted guest's state for each of its VCPUs), prepares the initial registers and page tables to map PTLsim's code, then unmaps all PTLsim reserved pages except for the first few pages (as shown in Table 14.1). This is required since the monitor process cannot have writable references to any of PTLsim's pages or PTLsim may not be able to pin those pages as page table pages. At this point, PTLmon atomically restarts the domain inside PTLsim using the new `contextswap` hypercall. The old context of the domain is thus available for PTLsim to use and update via simulation.

PTLmon also sets up two event channels: the *hostcall* channel and the *upcall* channel. PTLsim notifies the monitor process in domain 0 via the *hostcall* event channel whenever it needs to access the outside world. Specifically, PTLsim will fill in the *bootpage.hostreq* structure with parameters to a standard Linux system calls, and will place any larger buffers in the *transfer page* (see Table 14.1) visible to both PTLmon and PTLsim itself. PTLsim will then notify the *hostcall* channel's port. The domain 0 kernel will then forward this notification to PTLmon, which will do the system call on PTLsim's behalf (while PTLsim remains blocked in the `synchronous_host_call()` function). PTLmon will then notify the hostcall port in the opposite direction (waking up PTLsim) when the system call is complete. This is very similar to a remote procedure call, but using shared memory. It allows PTLsim to use standard system calls (e.g. for reading and writing log files) without modification, yet remains suitable for a bare-metal embedded environment.

PTLmon can also use the *upcall* channel to interrupt PTLsim, for instance to switch between native and simulation mode, trigger a snapshot, or request that PTLsim update its internal parameters. The PTLmon process sets up a socket in `/tmp/ptlsim-domain-XXX` and waits for requests on this socket. The user can then run the `ptlsim` command again, which will connect to this socket and tell the main monitor process for the domain to enqueue a text string (usually the command line parameters to `ptlsim`) and send an interrupt to PTLsim on the *upcall* channel. In response, PTLsim uses the `ACCEPT_UPCALL` hostcall to read the enqueued command line, then parses it and acts on any listed actions or parameter updates.

It should be noted that this design allows live configuration updates, as described in Section 13.6.

## 14.2 PTLsim Core

PTLsim runs directly on the "bare metal" and has no access to traditional OS services except through the DMA and interrupt based host call requests described above. Execution begins in *ptlsim_init()* in *ptlxen.cpp*. PTLsim first sets up its internal memory management (page pool, slab allocator, extent allocator in *mm.cpp* as described in Section 7.3) using the initial page tables created by PTLmon in conjunction with the modified Xen hypervisor. PTLsim owns the virtual address space range starting at `0xffffff0000000000` (i.e. x86-64 PML4 slot 510, of $2^{39}$ bytes). This memory is mapped to the physical pages reserved for PTLsim. The layout is shown in Table 14.1 (assuming 32 MB is allocated for PTLsim):

Starting at virtual address `0xfffffe0000000000` (i.e. x86-64 PML4 slot 508, of $2^{40}$ bytes), space is reserved to map all physical memory pages (MFNs) belonging to the guest domain. This mapping is sparse, since only a subset of the physical pages are accessible by the guest. When PTLsim is first injected into a domain, this space starts out empty. As various parts of PTLsim attempt to access physical addresses, PTLsim's internal page fault handler will map physical pages into this space. Normally all pages are mapped as writable, however Xen may not allow writable mappings to some types of pinned pages (L1/L2/L3/L4 page table pages, GDT pages, etc.). Therefore, if the writable mapping fails, PTLsim tries to map the page as read only. PTLsim monitors memory management related hypercalls as they are simulated and remaps physical pages as read-only or writable if and when they are pinned or unpinned, respectively. When PTLsim switches back to native mode, it quickly unmaps all guest pages, since we cannot hold writable references to any

Table 14.1: Memory Layout for PTLsim Space

| Page | Size | Description |
|---|---|---|
| (Pages below this point are shared by PTLmon in domain 0 and PTLsim in the target domain) | | |
| 0 | 4K | Boot info page and ptlxen.bin ELF header (see *xc_ptlsim.h* and *ptlxen.h* for the structures) |
| 1 | 4K | Hypercall entry points (filled in by Xen) |
| 2 | 4K | Shared info page for the domain |
| 3 | 4K | Shadow shared info page (as seen by guest) |
| 4 | 4K | Transfer page (for DMA between PTLmon in dom0 and target) |
| 5 | 128K | 32 VCPU Context structure pages |
| (Pages below this point are private to PTLsim in the target domain) | | |
| 37 | ~2M | PTLsim binary |
| - | ~28M | PTLsim heap (page pool, slab allocator, extent allocator) |
| - | ~256K | PTLsim stack |
| ... | ~64K | Page tables mapping 32 MB PTLsim space |
| ... | ~1MB | Page tables (level 1) mapping all physical pages (reserved but not filled in) |
| (32MB) | ~64K | Higher level page tables (levels 4/3/2) pointing to other tables |

pages the guest kernel may later attempt to pin as page table pages. This unmapping is done very quickly by simply clearing all present bits in the physical map's L2 page table page; the PTLsim page fault handler will re-establish the L2 entries as needed.

## 14.3 Implementation Details

### 14.3.1 Page Translation

The Xen-x86 architecture always has paging enabled, so PTLsim uses a simulated TLB for all virtual-to-physical translations. Each TLB entry has x86 accessed and dirty bits; whenever these bits transition from 0 to 1, PTLsim must walk the page table tree and actually update the corresponding PTE's accessed and/or dirty bit. Since page table pages are mapped read-only, our modified *update_mmu* hypercall is used to do this. TLB misses are serviced in the normal x86 way: the page tables are walked starting from the MFN in CR3 until the page is resolved. This is done by the `Context.virt_to_pte()` method, which returns the L1 page table entry (PTE) providing the physical address and accumulated permissions (x86 has specific rules for deriving the effective writable/executable/supervisor permissions for each page). Internally, the `page_table_walk()` function actually follows the page table tree, but PTLsim maintains a small 16-entry direct mapped cache (like a TLB) to accelerate repeated translations (this is not related to any true TLB maintained by specific cores). The `pte_to_ptl_virt()` function then translates the PTE and original virtual address into a pointer PTLsim can actually access (inside PTLsim's mapping of the domain's physical memory pages). The software TLB is also flushed under the normal x86 conditions (`MOV CR3`, `WBINVD`, `INVLPG`, and Xen hypercalls like `MMUEXT_NEW_BASE_PTR`). Presently TLB support is in `dcache.cpp`; the features above are incorporated into this TLB. In addition, `Context.copy_from_user()`

and `Context.copy_to_user()` functions are provided to walk the page tables and copy user data to or from a buffer inside PTLsim.

In 32-bit versions of Xen, the x86 protection ring mechanism is used to allow the guest kernel to run at ring 1 while guest userspace runs in ring 3; this allows the "supervisor" bit in PTEs to retain its traditional meaning. However, in its effort to clean up legacy ISA features, x86-64 has no concept of privilege rings (other than user/supervisor) or segmentation. This means the supervisor bit in PTEs is never set (only Xen internal pages not accessible to guest domains have this bit set). Instead, Xen puts the kernel in a separate address space from user mode; the top-level L4 page table page for kernel mode points to both kernel-only and user pages. Fortunately, Xen uses TLB global bits and other x86-64 features to avoid much of the context switch overhead from this approach. PTLsim does not have to worry about this detail during virtual-to-PTE translations: it just follows the currently active page table based on physical addresses only.

### 14.3.2 Exceptions

Under Xen, the `set_trap_table()` hypercall is used to specify an array of pointers to exception handlers; this is equivalent to the x86 LIDT (load interrupt descriptor table) instruction. Whenever we switch from native mode to simulation mode, PTLmon copies this array back into the `Context.idt[]` array. Whenever PTLsim detects an exception during simulation, it accesses `Context.idt[vector_id]` to determine where the pipeline should be restarted (CS:RIP). In the case of page faults, the simulated CR2 is loaded with the faulting virtual address. It then constructs a stack frame equivalent to Xen's structure (i.e. `iret_context`) at the stack segment and pointer stored in `Context.kernel_sp` (previously set by the `stack_switch()` hypercall, which replaces the legacy x86 TSS structure update). Finally, PTLsim propagates the page fault to the selected guest handler by redirecting the pipeline. This is essentially the same work performed within Xen by the `create_bounce_frame()` function, `do_page_fault()` (or its equivalent) and `propagate_page_fault()` (or its equivalent); all the same boundary conditions must be handled.

### 14.3.3 System Calls and Hypercalls

On 64-bit x86-64, the `syscall` instruction has a different meaning depending on the context in which it is executed. If executed from userspace, *syscall* arranges for execution to proceed directly to the guest kernel system call handler (in `Context.syscall_rip`). This is done by the `assist_syscall()` microcode handler. A similar process occurs when a 32-bit application uses "`int 0x80`" to make system calls, but in this case, `Context.propagate_x86_exception()` is used to redirect execution to the trap handler registered for that virtual software interrupt.

If executed from kernel space, the `syscall` instruction is interpreted as a hypercall into Xen itself. PTLsim emulates all Xen hypercalls. In many simple cases, PTLsim handles the hypercall all by itself, for instance by simply updating its internal tables. In other cases, the hypercall can safely be passed down to Xen without corrupting PTLsim's internal state. We must be very careful as to which hypercalls are passed through: for instance, before updating the page table base, we must ensure the new page table still maps PTLsim and the physical address space before we allow

Xen to update the hardware page table base. These cases are all documented in the comments of `handle_xen_hypercall()`.

Note that the definition of "user mode" and "kernel mode" is maintained by Xen itself: from the CPU's viewpoint, both modes are technically userspace and run in ring 3.

An interesting issue arises when PTLsim passes hypercalls through to Xen: some buffers provided by the guest kernel may reside in virtual memory not mapped by PTLsim. Normally PTLsim avoids this problem by copying any guest buffers into its own address space using `Context.copy_from_user()`, then copying the results back after the hypercall. However, to avoid future complexity, PTLsim currently switches its own page tables every time the guest requests a page table switch, such that Xen can see all guest kernel virtual memory as well as PTLsim itself. Obviously this means PTLsim injects its two top-level page table slots into every guest top level page table. For multi-processor simulation, PTLsim needs to swap in the target VCPU's page table base whenever it forwards a hypercall that depends on virtual addresses.

### 14.3.4 Event Channels

Xen delivers outside events, virtual interrupts, IPIs, etc. to the domain just like normal, except they are redirected to a special PTLsim upcall handler stub (in `lowlevel-64bit-xen.S`). The handler checks which events are pending, and if any events (other than the PTLsim hostcall and upcall events) are pending, it sets a flag so the guest's event handler is invoked the next time through the main loop. This process is equivalent to exception handling in terms of the stack frame setup and call/return sequence: the simulated pipeline is simply redirected to the handler address. It should be noted that the PTLsim handler does not set or clear any mask bits in the shared info page, since it's the (emulated) guest OS code that should actually be doing this, not PTLsim. The only exception is when the event in question is on the hostcall port or the upcall port; then PTLsim handles the event itself and never notifies the guest.

### 14.3.5 Privileged Instruction Emulation

Xen lets the guest kernel execute various privileged instructions, which it then traps and emulates with internal hypercalls. These are the same as in Xen's arch/x86/traps.c: CLTS (FPU task switches), MOV from CR0-CR4 (easy to emulate), MOV to and from DR0-DR7 (get or set debug registers), RDMSR and WRMSR (mainly to set segment bases). PTLsim decodes and executes these instructions on its own, just like any other x86 instruction.

## 14.4 PTLcalls

PTLsim defines the special x86 opcode `0x0f37` as a breakout opcode. It is undefined in the normal x86 instruction set, but when executed by any code running under PTLsim, it can be used to enqueue commands for PTLsim to execute.

The `ptlctl` program uses this facility to switch from native mode to simulation mode as follows. Whenever PTLsim is about to switch back to native mode, it uses the `VCPUOP_set_breakout_insn_action` to specify the opcode bytes to intercept. When the hypervisor sees an invalid instruction matching `0x0f37`, it freezes the domain and sends an event channel notification to domain 0. This event channel is read by PTLmon, which then uses the `contextswap` hypercall to switch back into PTLsim inside the domain. PTLsim then processes whatever command caused the switch back into simulation mode.

While executing *within* simulation mode, this is not necessary: since PTLsim is in complete control of the execution of each x86 instruction, it simply defines microcode to handle `0x0f37` instead of triggering an invalid opcode exception. This microcode branches into PTLsim, which uses the `PTLSIM_HOST_INJECT_UPCALL` hostcall to add the command(s) to the command queue. The queue is maintained inside PTLmon so as to ensure synchronization between commands coming from the host and commands from within the domain arriving via PTLcalls. The queue is typically flushed before adding new commands in this manner: otherwise, it would be impossible to get immediate results using `ptlctl`.

All PTL calls are defined in `ptlcalls.h`, which simply collects the call's arguments and executes opcode `0x0f37` as if it were a normal x86 `syscall` instruction:

- `ptlcall_multi_enqueue(const char* list[], size_t length)` enqueues a list of commands to process in sequence
- `ptlcall_multi_flush(const char* list[], size_t length)` flushes the queue before adding the commands
- `ptlcall_single_enqueue(const char* command)` adds one command to the end of the queue
- `ptlcall_single_flush(const char* command)` immediately flushes the queue and processes the specified command
- `ptlcall_nop()` is a simple no-operation command used to get PTLsim's attention
- `ptlcall_version()` returns version information about the running PTLsim hypervisor.

The `ptlcall` opcode `0x0f37` can be executed from both user mode and kernel mode, since it may be desirable to switch simulation options from a userspace program. This would be impossible if `wrmsr` (the traditional method) were used to effect PTLsim operations.

## 14.5 Event Trace Mode

In Section 13.9, we discussed the philosophical question of how to accurately model the timing of external events when cycle accurate simulation runs thousands of times slower than the outside world expects. To solve this problem, PTLsim/X offers *event trace* mode.

First, the user saves a checkpoint of the target domain, then instructs PTLsim to enter *event record* mode. The domain is then used interactively in native mode at full speed, for instance by starting benchmarks and waiting for their completion. In the background, PTLsim taps Xen's trace buffers to write any events delivered to the domain into an event trace file. "Events" refer to any time-dependent outside stimulus delivered to the domain, such as interrupts (i.e. Xen event channel notifications) and DMA traffic (i.e. the full contents of any grant pages from network or disk I/O transferred into the domain). Each event is timestamped with the relative cycle number (timestamp counter) in which it was delivered, rather than the wall clock time. When the benchmarks are done, the trace mode is terminated and recording stops.

The user then restores the domain from the checkpoint and re-injects PTLsim, but this time PTLsim reads the event trace file, rather than responding to any outside events Xen may deliver to the domain while in simulation mode. Whenever the timestamp of the event at the head of the trace file matches the current simulation cycle, that event is injected into the domain. PTLsim does this by setting the appropriate pending bits in the shared info page, and then simulates an upcall to the domain's shared info handler (i.e. by restarting the simulated pipeline at that RIP). Since the event channels used by PTLsim and those of the target domain may interfere, PTLsim maintains a shadow shared info page that's updated instead; whenever the simulated load/store pipeline accesses the real shared info page's physical address, the shadow page is used in its place. In addition, the wall clock time fields in the shadow shared info page are regularly updated by dividing the simulated cycle number by the native CPU clock frequency active during the record mode (since the guest OS will have recorded this internally in many places).

This scheme does require some extra software support, since we need to be able to identify which pages the outside source has overwritten with incoming data (i.e. as in a virtual DMA). The console I/O page is actually a guest page that domain 0 maps in *xenconsoled*; this is easy to identify and capture. The network and block device pages are typically grant pages; the domain 0 Linux device drivers must be modified to let PTLsim know what pages will be overwritten by outside sources.

## 14.6 Multiprocessor Support

PTLsim/X is designed from the ground up to support multiple VCPUs per domain. The `contextof(vcpuid)` function returns the Context structure allocated for each VCPU; this structure is passed to all functions and assists dealing with the domain. It is the responsibility of each core (e.g. sequential core, out of order core, user-designed cores, etc.) to update the appropriate context structure according to its own design.

VCPUs may choose to block by executing an appropriate hypercall (`sched_block`, `sched_yield`, etc.), typically suspending execution until an event arrives. PTLsim cores can simulate this by checking the `Context.running` field; if zero, the corresponding VCPU is blocked and no instructions should be processed until the `running` flag becomes set, such as when an event arrives. In realtime mode (where Xen relays real events like timer interrupts back to the simulated CPU), events and upcalls may be delivered to other VCPUs than the first VCPU which runs PTLsim; in this case, PTLsim must check the pending bitmap in the shared info page and simulate upcalls within the appropriate VCPU context (i.e. whichever VCPU context has its `upcall_pending` bit set).

Some Xen hypercalls must only be executed on the VCPU to which the hypercall applies. In cases where PTLsim cannot emulate the hypercall on its own internal state (and defer the actual hypercall until switching back to native mode), the Xen hypervisor has been modified to support an explicit *vcpu* parameter, allowing the first VCPU (which always runs PTLsim itself) to execute the required action on behalf of other VCPUs.

For simultaneous multithreading support, PTLsim is designed to run the simulation entirely on the first VCPU, while putting the other VCPUs in an idle loop. This is required because there's no easy way to parallelize an SMT core model across multiple simulation threads. In theory, a multi-core simulator could in fact be parallelized in this way, but it would be very difficult to reproduce cycle accurate behavior and debug deadlocks with asynchronous simulations running in different threads. For these reasons, currently PTLsim itself is single threaded in simulation mode, even though it simulates multiple virtual cores or threads.

Cache coherence is the responsibility of each core model. By default, PTLsim uses the "instant visibility" model, in which all VCPUs can have read/write copies of cache lines and all stores appear on all other VCPUs the instant they commit. More complex MOESI-compliant policies can be implemented on top of this basic framework, by stalling simulated VCPUs until cache lines travel across an interconnect network.

# Part IV

# Out of Order Processor Model

# Chapter 15

# Introduction

## 15.1 Out Of Order Core Features

PTLsim completely models a modern out of order x86-64 compatible processor, cache hierarchy and key devices with true cycle accurate simulation. The basic microarchitecture of this model is a combination of design features from the Intel Pentium 4, AMD K8 and Intel Core 2, but incorporates some ideas from IBM Power4/Power5 and Alpha EV8. The following is a summary of the characteristics of this processor model:

- The simulator directly fetches pre-decoded micro-operations (Section 17.1) but can simulate cache accesses as if x86 instructions were being decoded on fetch
- Branch prediction is configurable; PTLsim currently includes various models including a hybrid g-share based predictor, bimodal predictors, saturating counters, etc.
- Register renaming takes into account x86 quirks such as flags renaming (Section 5.4)
- Front end pipeline has configurable number of cycles to simulate x86 decoding or other tasks; this is used for adjusting the branch mispredict penalty
- Unified physical and architectural register file maps both in-flight uops as well as committed architectural register values. Two rename tables (speculative and committed register rename tables) are used to track which physical registers are currently mapped to architectural registers.
- Unified physical register file for both integer and floating point values.
- Operands are read from the physical register file immediately before issue. Unlike in some microprocessors, PTLsim does not do speculative scheduling: the schedule and register read loop is assumed to take one cycle.
- Issue queues based on a collapsing design use broadcast based matching to wake up instructions.

- Clustered microarchitecture is highly configurable, allowing multi-cycle latencies between clusters and multiple issue queues within the same logical cluster.
- Functional units, mapping of functional units to clusters, issue ports and issue queues and uop latencies are all configurable.
- Speculation recovery from branch mispredictions and load/store aliasing uses the forward walk method to recover the rename tables, then annuls all uops after and optionally including the mis-speculated uop.
- Replay of loads and stores after store to load forwarding and store to store merging dependencies are discovered.
- Stores may issue even before data to store is known; the store uop is replayed when all operands arrive.
- Load and store queues use partial chunk address matching and store merging for high performance and easy circuit implementation.
- Prediction of load/store aliasing to avoid mis-speculation recovery overhead.
- Prediction and splitting of unaligned loads and stores to avoid mis-speculation overhead
- Commit unit supports stalling until all uops in an x86 instruction are complete, to make x86 instruction commitment atomic

The PTLsim model is fully configurable in terms of the sizes of key structures, pipeline widths, latency and bandwidth and numerous other features.

## 15.2 Processor Contexts

PTLsim uses the concept of a *VCPU* (virtual CPU) to represent one user-visible microprocessor core (or a hardware thread if a SMT machine is being modeled). The `Context` structure (defined in `ptlhwdef.h`) maintains all per-VCPU state in PTLsim: this includes both user-visible architectural registers (in the `Context.commitarf[]` array) as well as all per-core control registers and internal state information. `Context` only contains general x86-visible context information; specific machine models must maintain microarchitectural state (like physical registers and so forth) in their own internal structures.

The `contextof(N)` macro is used to return the `Context` object for a specific VCPU, numbered 0 to `contextcount-1`. In userspace-only PTLsim, there is only one context, `contextof(0)`. In full system PTLsim/X, there may be up to 32 (i.e. `MAX_CONTEXTS`) separate contexts (VCPUs).

## 15.3 PTLsim Machine/Core/Thread Class Hierarchy

PTLsim easily supports user defined plug-in machine models. Two of these models, the out of order core (“`ooo`”) and the sequential in-order core (“`seq`”) ship with PTLsim; others can be easily added by users. PTLsim implements several C++ classes used to build simulation models by dividing a virtual machine into CPU sockets, cores and threads.

The `PTLsimMachine` class is at the root of the hierarchy. Every simulation model must subclass `PTLsimMachine` and define its virtual methods. Adding a machine model to PTLsim is very simple: simply define one instance of your machine class in a source file included in the Makefile. For instance, assuming `XYZMachine` subclasses `PTLsimMachine` and will be called “xyz”:

```
XyzMachine xyzmodel(“xyz”);
```

The constructor for `XyzMachine` will be called by PTLsim after all other subsystems are brought up. It should use the `addmachine(“`*`name`*`”)` static method to register the core model’s name with PTLsim, so it can be specified using the “`-core` *`xyz`*” option.

The machine models included with PTLsim (namely, `OutOfOrderMachine` and `SequentialMachine`) have been placed in their own C++ namespace. When adding your own core, copy the example source file(s) to new names and adjust the namespace specifiers to a new name to avoid clashes. You should be able to link any number of machine models defined in this manner into PTLsim all at once.

The `PTLsimMachine::init()` method is called to initialize each machine model the first time it is used. This function is responsible for dividing the *`contextcount`* contexts up into sockets, cores and threads, depending entirely on the machine model’s design and any configuration options specified by the `config` parameter.

`PTLsimMachine::run()` is called to actually run the simulation; more details will be given on this later.

`PTLsimMachine::update_stats()` is described in Section 8.

`PTLsimMachine::dump_state()` is called to aid debugging whenever an assertion fails, the simulator accesses a null pointer or invalid address, or from anywhere else it may be useful.

# Chapter 16

# Out Of Order Core Overview

The out of order core is spread across several source files:

- `ooocore.cpp` contains control logic, the definition of the `OutOfOrderMachine` class and its functions (see Section 15.3), the top-level pipeline control functions, all event printing logic (Section 16.1) and miscellaneous code.
- `ooopipe.cpp` contains all pipeline stages, except for execution stages and functional units.
- `oooexec.cpp` contains the functional units, load/store unit, issue queues, replay control and exception handling.
- `ooocore.h` defines all structures and lists easy to configure parameters.

The `OutOfOrderMachine` structure is divided into an array of one or more `OutOfOrderCore` structures (by default, one per VCPU). The `OutOfOrderMachine::init()` function creates *`contextcount`* cores and binds one per-VCPU `Context` structure to each core. The `init()` function is declared in `ooocore.h`, since some user configurable state is set up at this point.

The `OutOfOrderMachine::run()` function first flushes the pipeline in each core, using `core.flush_pipeline()` to copy state from the corresponding `Context` structure into the physical register file and other per-core structures (see Section 24.6 for details).

The `run()` function then enters a loop with one iteration per simulated cycle:

- `update_progress()` prints the current performance information (cycles, committed instructions and simulated cycles/second) to the console and/or log file.
- `inject_events()` injects any pending interrupts and outside events into the processor; these will be processed at the next x86 instruction boundary. This function only applies to full system PTLsim/X.
- The `OutOfOrderCore::runcycle()` function is called for each core in sequence, to step its entire state machine forward by one cycle (see below for details). If a given core is blocked (i.e. paused while waiting for some outside event), its Context.running field is zero; in this case, the core's `handle_interrupt()` method may be called to wake it up (see below).

- Any global structures (like memory controllers or interconnect networks) are clocked by one cycle using their respective `clock()` methods.
- `check_for_async_sim_break()` checks if the user has requested the simulation stop or switch back to native mode. This function only applies to full system PTLsim/X.
- The global cycle counter and other counters are incremented.

The `OutOfOrderCore::runcycle()` function is where the majority of the work in PTLsim's out of order model occurs. This function, in ooocore.cpp, runs one cycle in the core by calling functions to implement each pipeline stage, the per-core data caches and other clockable structure. If the core's commit stage just encountered a special event (e.g. barrier, microcode assist request, exception, interrupt, etc.), the appropriate action is taken at the cycle boundary.

In the following chapters, we describe every pipeline stage and structure in detail.

Every structure in the out of order model can obtain a reference to its parent `OutOfOrderCore` structure by calling its own `getcore()` method. Similarly, `getcore().ctx` returns a reference to the `Context` structure for that core.

## 16.1 Event Log Ring Buffer

Section 10.5 describes PTLsim's event log ring buffer system, in which the simulator can log all per-cycle events to a circular ring buffer when the `-ringbuf` option is given. The ring buffer can help developers look backwards in time from when an undesirable event occurs (for instance, as specified by `-ringbuf-trigger-rip`), allowing much easier debugging and experimentation.

In the out of order core, the `EventLog` structure provides this ring buffer. The buffer consists of an array of `OutOfOrderCoreEvent` structures (in `ooocore.h`); each structure contains a fixed header with subject information common to all events (e.g. the cycle, uuid, RIP, uop, ROB slot, and so forth), plus a union with sub-structures for each possible event type. The actual events are listed in an enum above this structure.

The `EventLog` class has various functions for quickly adding certain types of events and filling in their special fields. Specifically, calling one of the `EventLog::add()` functions allocates a new record in the ring buffer and returns a pointer to it, allowing additional event-specific fields to be filled in if needed. The usage of these functions is very straightforward and documented by example in the various out of order core source files.

In `ooocore.cpp`, the `OutOfOrderCoreEvent::print()` method lists all event types and gives code to nicely format the recorded event data. The `eventlog.print()` function prints every event in the ring buffer; this function can be called from anywhere an event backtrace is needed.

# Chapter 17

# Fetch Stage

## 17.1 Instruction Fetching and the Basic Block Cache

As described in Section 5.1, x86 instructions are decoded into transops prior to actual execution by the out of order core. Some processors do this translation as x86 instructions are fetched from an L1 instruction cache, while others use a trace cache to store pre-decoded uops. PTLsim takes a middle ground to allow maximum simulation flexibility. Specifically, the Fetch stage accesses the L1 instruction cache and stalls on cache misses as if it were fetching several variable length x86 instructions per cycle. However, actually decoding x86 instructions into uops over and over again during simulation would be extraordinarily slow.

Therefore, for *simulation purposes only*, the out of order model uses the PTLsim *basic block cache*. The basic block cache, described in Chapter 6, stores pre-decoded uops for each basic block, and is indexed using the `RIPVirtPhys` structure, consisting of the RIP virtual address, several context-dependent flags and the physical page(s) spanned by the basic block (in PTLsim/X only).

During the fetch process (implemented in the `OutOfOrderCore::fetch()` function in `ooopipe.cpp`), PTLsim looks up the current RIP to fetch from (`fetchrip`), uses the current context to construct a full `RIPVirtPhys` key, then uses this key to query the basic block cache. If the basic block has never been decoded before, `bbcache.translate()` is used to do this now. This is all done by the `fetch_or_translate_basic_block()` function.

Technically speaking, the cached basic blocks contain *transops*, rather than uops: as explained in Section 5.1, each transop gets transformed into a true uop after it is renamed in the rename stage. In the following discussion, the term uop is used interchangeably with transop.

## 17.2 Fetch Queue

Each transop fetched into the pipeline is immediately assigned a monotonically increasing *uuid* (universally unique identifier) to uniquely track it for debugging and statistical purposes. The fetch unit attaches additional information to each transop (such as the uop's uuid and the `RIPVirtPhys` of the corresponding x86 instruction) to form a `FetchBufferEntry` structure. This fetch buffer is then placed into the fetch queue (`fetchq`) assuming it isn't full (if it is, the fetch stage stalls). As the fetch unit encounters transops with their EOM (end of macro-op) bit set, the fetch RIP is advanced to the next x86 instruction according to the instruction length stored in the SOM transop.

Branch uops trigger the branch prediction mechanism (Section 26) used to select the next fetch RIP. Based on various information encoded in the branch transop and the next RIP *after* the x86 instruction containing the branch, the `branchpred.predict()` function is used to redirect fetching. If the branch is predicted not taken, the sense of the branch's condition code is inverted and the transop's `riptaken` and `ripseq` fields are swapped; this ensures all branches are considered correct only if taken. Indirect branches (jumps) have their `riptaken` field overwritten by the predicted target address.

PTLsim models the instruction cache by using the `caches.probe_icache()` function to probe the cache with the physical address of the current *fetch window*. Most modern x86 processors fetch aligned 16-byte or 32-byte blocks of bytes into the decoder and try to pick out 3 or 4 x86 instructions per cycle. Since PTLsim uses the basic block cache, it does not actually decode anything at this point, but it still attempts to pick out up to 4 uops (or whatever limit is specified in `ooocore.h`) within the current 16-byte window around the fetch RIP; switching to a new window must occur in the next cycle. The instruction cache is only probed when switching fetch windows.

If the instruction cache indicates a miss, or the ITLB misses, the `waiting_for_icache_fill` variable is set, and the fetch unit remains stalled in subsequent cycles until the cache subsystem calls the `OutOfOrderCoreCacheCallbacks::icache_wakeup()` callback registered by the core. The core's interactions with the cache subsystem will be described in great detail later on.

# Chapter 18

# Frontend and Key Structures

## 18.1 Resource Allocation

During the Allocate stage, PTLsim dequeues uops from the fetch queue, ensures all resources needed by those uops are free, and assigns resources to each uop as needed. These resources include Reorder Buffer (ROB) slots, physical registers and load store queue (LSQ) entries. In the event that the fetch queue is empty or any of the ROB, physical register file, load queue or store queue is full, the allocation stage stalls until some resources become available.

## 18.2 Reorder Buffer Entries

The Reorder Buffer (ROB) in the PTLsim out of order model works exactly like a traditional ROB: as a queue, entries are allocated from the tail and committed from the head. Each `ReorderBufferEntry` structure is the central tracking structure for uops in the pipeline. This structure contains a variety of fields including:

- The decoded uop (`uop` field). This is the fully decoded `TransOp` augmented with fetch-related information like the uop's UUID, RIP and branch predictor information as described in the Fetch stage (Section 17.1).
- Current state of the ROB entry and uop (`current_state_list`; see below)
- Pointers to the physical register (`physreg`), LSQ entry (`lsq`) and other resources allocated to the uop
- Pointers to the three physical register operands to the uop, as well as a possible store dependency used in replay scheduling (described later)
- Various cycle counters and related fields for simulating progress through the pipeline

### 18.2.1 ROB States

Each ROB entry and corresponding uop can be in one of a number of states describing its progress through the simulator state machine. ROBs are linked into linked lists according to their current state; these lists are named `rob_`*statename*`_list`. The `current_state_list` field specifies the list the ROB is currently on. ROBs can be moved between states using the `ROB::changestate(`*`statelist`*`)` method. The specific states will be described below as they are encountered.

***NOTE:*** the terms "ROB entry" (singular) and "uop" are used interchangeably from now on unless otherwise stated, since there is a 1:1 mapping between the two.

## 18.3 Physical Registers

### 18.3.1 Physical Registers

Physical registers are represented in PTLsim by the `PhysicalRegister` structure. Physical registers store several components:

- Index of the physical register (`idx`) and the physical register file id (`rfid`) to which it belongs
- The actual 64-bit register data
- x86 flags: Z, P, S, O, C. These are discussed below in Section 5.4.
- Waiting flag (`FLAG_WAIT`) for results not yet ready
- Invalid flag (`FLAG_INVAL`) for ready results which encountered an exception. The exception code is written to the data field in lieu of the real result
- Current state of the physical register (`state`)
- ROB currently owning this physical register, or architectural register mapping this physical register
- Reference counter for the physical register. This is required for reasons described in Section 24.5.

### 18.3.2 Physical Register File

PTLsim uses a flexible physical register file model in which multiple physical register files with different sizes and properties can optionally be defined. Each physical register file in the `OutOfOrderCore::physregfi` array can be made accessible from one or more clusters. For instance, uops which execute on floating point clusters can be forced to always allocate a register in the floating point register file, or each cluster can have a dedicated register file.

Various heuristics can also be used for selecting the register file into which a result is placed. The default heuristic simply finds the first acceptable physical register file with a free register. Acceptable physical register files are those register files in which the uop being allocated is allowed to write its result; this is configurable based on clustering as described below. Other allocation policies, such as alternation between available register files and dependency based register allocation, are all possible by modifying the `rename()` function where physical registers are allocated..

In each physical register file, physical register number 0 is defined as the *null register:* it always contains the value zero and is used as an operand anywhere the zero value (or no value at all) is required.

Physical register files are configured in `ooocore.h`. The `PhysicalRegisterFile[]` array is defined to declare each register file by name, register file ID (RFID, from 0 to the number of register files) and size. The `MAX_PHYS_REG_FILE_SIZE` parameter must be greater than the largest physical register in the processor.

### 18.3.3 Physical Register States

Each physical register can be in one of several states at any given time. For each physical register file, PTLsim maintains linked lists (the `PhysicalRegisterFile.states[`*statename*`]` lists) to track which registers are in each state. The `state` field in each physical register specifies its state, and implies that the physical register is on the list `physregfiles[physreg.rfid].states[physreg.state]`. The valid states are:

- ***free:*** the register is not allocated to any uop.
- ***waiting:*** the register has been allocated to a uop but that uop is waiting to issue.
- ***bypass:*** the uop associated with the register has issued and produced a value (or encountered an exception), but that value is only on the bypass network - it has not actually been written back yet. For simulation purposes only, uops immediately write their results into the physical register as soon as they issue, even though technically the result is still only on the bypass network. This helps simplify the simulator considerably without compromising accuracy.
- ***written:*** the uop associated with the register has passed through the writeback stage and the value of the physical register is now up to date; all future consumers will read the uop's result from this physical register.
- ***arch:*** the physical register is currently mapped to one of the architectural registers; it has no associated uop currently in the pipeline
- ***pendingfree:*** this is a special state described in Section 24.5.

One physical register is allocated to each uop and moved into the *waiting* state, regardless of which type of uop it is. For integer, floating point and load uops, the physical register holds the actual numerical value generated by the corresponding uop. Branch uops place the target RIP of the branch in a physical register. Store uops place the merged data to store in the register. Technically branches and stores do not need physical registers, but to keep the processor design simple, they are allocated registers anyway.

## 18.4 Load Store Queue Entries

Load Store Queue (LSQ) Entries (the `LoadStoreQueueEntry` structure in PTLsim) are used to track additional information about loads and stores in the pipeline that cannot be represented by a physical register. Specifically, LSQ entries track:

- **Physical address** of the corresponding load or store
- **Data** field (64 bits) stores the loaded value (for loads) or the value to store (for stores)
- **Address valid** bit flag indicates if the load or store knows its effective physical address yet. If set, the physical address field is valid.
- **Data valid** bit flag indicates if the data field is valid. For loads, this is set when the data has arrived from the cache. For stores, this is set when the data to store becomes ready and is merged.
- **Invalid** bit flag is set if an exception occurs in the corresponding load or store.

The `LoadStoreQueueEntry` structure is technically a superset of a structure known as an *SFR* (Store Forwarding Register), which completely represents any load or store and can be passed between PTLsim subsystems easily. One LSQ entry is allocated to each load or store during the Allocate stage.

In real processors, the load queue (LDQ) and store queue (STQ) are physically separate for circuit complexity reasons. However, in PTLsim a unified LSQ is used to make searching operations easier. One additional bit flag (`store` bit) specifies whether an LSQ entry is a load or store.

### 18.4.1 Register Renaming

The basic register renaming process in the PTLsim x86 model is very similar to classical register renaming, with the exception of the flags complications described in Section 5.4. Two versions of the register rename table (RRT) are maintained: a *speculative RRT* which is updated as uops are renamed, and a *commit RRT*, which is only updated when uops successfully commit. Since the simulator implements a unified physical and architectural register file, the commit process does not actually involve any data movement between physical and architectural registers: only the commit RRT needs to be updated. The commit RRT is used only for exception and branch mispredict recovery, since it holds the last known good mapping of architectural to physical registers.

Each rename table contains 80 entries as shown in Table 18.1. This table maps architectural registers and pseudo-registers to the most up to date physical registers for the following:

Table 18.1: Architectural registers and pseudo-registers used for renaming.

| Architectural Registers and Pseudo-Registers | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | rax | rcx | rdx | rbx | rsp | rbp | rsi | rdi |
| 8 | r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15 |
| 16 | xmml0 | xmmh0 | xmml1 | xmmh1 | xmml2 | xmmh2 | xmml3 | xmmh3 |
| 24 | xmml4 | xmmh4 | xmml5 | xmmh5 | xmml6 | xmmh6 | xmml7 | xmmh7 |
| 32 | xmml8 | xmmh8 | xmml9 | xmmh9 | xmml10 | xmmh10 | xmml11 | xmmh11 |
| 40 | xmml12 | xmmh12 | xmml13 | xmmh13 | xmml14 | xmmh14 | xmml15 | xmmh15 |
| 48 | fptos | fpsw | fptags | fpstack | tr4 | tr5 | tr6 | ctx |
| 56 | rip | flags | iflags | selfrip | nextrip | ar1 | ar2 | zero |
| 64 | temp0 | temp1 | temp2 | temp3 | temp4 | temp5 | temp6 | temp7 |
| 72 | zf | cf | of | imm | mem | temp8 | temp9 | temp10 |

- 16 x86-64 integer registers
- 16 128-bit SSE registers (represented as separate 64-bit high and low halves)
- ZAPS, CF, OF flag sets described in Section 5.4. These rename table entries point to the physical register (with attached flags) of the most recent uop in program order to update any or all of the ZAPS, CF, OF flag sets, respectively.
- Various integer and x87 status registers
- Temporary pseudo-registers `temp0-temp7` not visible to x86 code but required to hold temporaries (e.g. generated addresses or value to swap in `xchg` instructions).
- Special fixed values, e.g. `zero`, `imm` (value is in immediate field), `mem` (destination of stores)

Once the uop's three architectural register sources are mapped to physical registers, these physical registers are placed in the `operands[0,1,2]` fields. The fourth operand field, `operands[3]`, is used to hold a store buffer dependency for loads and stores; this will be discussed later. The speculative RRT entries for both the destination physical register and any modified flags are then overwritten. Finally, the ROB is moved into the ***frontend*** state.

### 18.4.2 External State

Since the rest of the simulator outside of the out of order core does not know about the RRTs and expects architectural registers to be in a standardized format, the per-core `Context` structure is used to house the architectural register file. These architectural registers, including `REG_flags` and `REG_rip`, are directly updated in program order by the out of order core as instructions commit.

## 18.5 Frontend Stages

To simulate various processor frontend pipeline depths, ROBs are placed in the *frontend* state for a user-selectable number of cycles. In the `frontend()` function, the `cycles_left` field in each ROB is decremented until it becomes zero. At this point, the uop is moved to the ***ready_to_dispatch*** state. This feature can be used to simulate various branch mispredict penalties by setting the `FRONTEND_STAGES` constant.

# Chapter 19

# Scheduling, Dispatch and Issue

## 19.1 Clustering and Issue Queue Configuration

The PTLsim out of order model can simulate an arbitrarily complex set of functional units grouped into *clusters*. Clusters are specified by the `Cluster` structure and are defined by the `clusters[]` array in `ooocore.h`. Each `Cluster` element defines the name of the cluster, which functional units belong to the cluster (`fu_mask` field) and the maximum number of uops that can be issued in that cluster each cycle (`issue_width` field)

The `intercluster_latency_map` matrix defines the forwarding latency, in cycles, between a given cluster and every other cluster. If `intercluster_latency_map[`*A*`][`*B*`]` is *L* cycles, this means that functional units in cluster *B* must wait *L* cycles after a uop *U* in cluster A completes before cluster B's functional units can issue a uop dependent on *U*'s result. If the latency is zero between clusters *A* and *B*, producer and consumer uops in *A* and *B* can always be issued back to back in subsequent cycles. Hence, the diagonal of the forwarding latency matrix is always all zeros.

This clustering mechanism can be used to implement several features of modern microprocessors. First, traditional clustering is possible, in which it takes multiple additional cycles to forward results between different clusters (for instance, one or more integer clusters and a floating point unit). Second, several issue queues and corresponding issue width limits can be defined within a given virtual cluster, for instance to sort loads, stores and ALU operations into separate issue queues with different policies. This is done by specifying an inter-cluster latency of zero cycles between the relevant pseudo-clusters with separate issue queues. Both of these uses are required to accurately model most modern processors.

There is also an equivalent `intercluster_bandwidth_map` matrix to specify the maximum number of values that can be routed between any two clusters each cycle.

The `IssueQueue` template class is used to declare issue queues; each cluster has its own issue queue. The syntax `IssueQueue<`*size*`> issueq_`*name*`;` is used to declare an issue queue with a specific size. In the current implementation, the size can be from 1 to 64 slots. The macros `foreach_issueq()`, `sched_get_all_issueq_free_slots()` and `issueq_operation_on_cluster_with_result()` macros must be modified if the cluster and issue queue configuration is changed to reflect all available clusters; the modifications required should be obvious from the example code. These macros with switch statements are required instead of a simple array since the issue queues can be of different template types and sizes.

## 19.2 Cluster Selection

The `ReorderBufferEntry::select_cluster()` function is responsible for routing a given uop into a specific cluster at the time it is dispatched; uops do not switch between clusters after this.

Various heuristics are employed to select which cluster a given uop should be routed to. In the reference implementation provided in `ooopipe.cpp`, a weighted score is generated for each possible cluster by scanning through the uop's operands to determine which cluster they will be forwarded from. If a given operand's corresponding producer uop $S$ is currently either dispatched to cluster $C$ but waiting to execute or is still on the bypass network of cluster $C$, then cluster $C$'s score is incremented.

The final cluster is selected as the cluster with the highest score out of the set of clusters which the uop can actually issue on (e.g. a floating point uop cannot issue on a cluster with only integer units). The `ReorderBufferEntry::executable_on_cluster_mask` bitmap can be used to further restrict which clusters a uop can be dispatched to, for instance because certain clusters can only write to certain physical register files. This mechanism is designed to route each uop to the cluster in which the majority of its operands will become available at the earliest time; in practice it works quite well and variants of this technique are often used in real processors.

## 19.3 Issue Queue Structure and Operation

PTLsim implements issue queues in the `IssueQueue` template class using the collapsing priority queue design used in most modern processors.

As each uop is dispatched, it is placed at the end of the issue queue for its cluster and several associative arrays are updated to reflect which operands the uop is still waiting for. In the IssueQueue class, the `insert()` method takes the ROB index of the uop (its *tag* in issue queue terminology), the tags (ROB indices) of its operands, and a map of which of the operands are ready versus waiting. The ROB index is inserted into an associative array, and the ROB index tags of any waiting operands are inserted into corresponding slots in parallel arrays, one array per operand (in the current implementation, up to 4 operands are tracked). If an operand was ready at dispatch time, the slot for that operand in the corresponding array is marked as invalid since there is no need to wake it up later. Notice that the new slot is always at the end of the issue queue array; this is made possible by the collapsing mechanism described below.

The issue queue maintains two bitmaps to track the state of each slot in the queue. The `valid` bitmap indicates which slots are occupied by uops, while the `issued` bitmap indicates which of those uops have been issued. Together, these two bitmaps form the state machine described in Table 19.1.

Table 19.1: Issue Queue State Machine

| Valid | Issued | Meaning |
|---|---|---|
| 0 | 0 | Unused slot |
| 0 | 1 | (invalid) |
| 1 | 0 | Dispatched but waiting for operands |
| 1 | 1 | Issued to a functional unit but not yet completed |

After `insert()` is called, the slot is placed in the dispatched state. As each uop completes, its tag (ROB index) is broadcast using the `broadcast()` method to one or more issue queues accessible in that cycle. Because of clustering, some issue queues will receive the broadcast later than others; this is discussed below. Each slot in each of the four operand arrays is compared against the broadcast value. If the operand tag in that slot is valid and matches the broadcast tag, the slot (in one of the operand arrays only, not the entire issue queue) is invalidated to indicate it is ready and no longer waiting for further broadcasts.

Every cycle, the `clock()` method uses the `valid` and `issued` bitmaps together with the valid bitmaps of each of the operand arrays to compute which issue queue slots in the dispatched state are no longer waiting on any of their operands. This bitmap of ready slots is then latched into the `allready` bitmap.

The `issue()` method simply finds the index of the first set bit in the `allready` bitmap (this is the slot of the oldest ready uop in program order), marks the corresponding slot as issued, and returns the slot. The processor then selects a functional unit for the uop in that slot and executes it via the `ReorderBufferEntry::issue()` method. After the uop has completed execution (i.e. it cannot possibly be replayed), the `release()` method is called to remove the slot from the issue queue, freeing it up for incoming uops in the dispatch stage. The collapsing design of the issue queue means that the slot is not simply marked as invalid - all slots after it are physically shifted left by one, leaving a free slot at the end of the array. This design is relatively simple to implement in hardware and makes determining the oldest ready to issue uop very trivial.

Because of the collapsing mechanism, it is critical to note that the slot index returned by `issue()` will become invalid after the next call to the `remove()` method; hence, it should never be stored anywhere if a slot could be removed from the issue queue in the meantime.

If a uop issues but determines that it cannot actually complete at that time, it must be *replayed*. The `replay()` method clears the issued bit for the uop's issue queue slot, returning it to the dispatched state. The replay mechanism can optionally add additional dependencies such that the uop is only re-issued after those dependencies are resolved. This is important for loads and stores, which may need to add a dependency on a prior store queue entry after finding a matching address in the load or store queues. In rare cases, a replay may also be required when a uop is issued but no applicable functional units are left for it to execute on. The `ReorderBufferEntry::replay()` method is a wrapper around `IssueQueue::replay()` used to collect the operands the uop is still waiting for.

### 19.3.1 Implementation

PTLsim uses a novel method of modeling the issue queue and other associative structures with small tags. Specifically, the `FullyAssociativeArrayTags8bit` template class declared in `logic.h` and used to build the issue queue makes use of the host processor's 128-bit vector (SSE) instructions to do massively parallel associative matching, masking and bit scanning on up to 16 tags every clock cycle. This makes it substantially faster than simulators using the naive approach of scanning the issue queue entries linearly. Similar classes in `logic.h` support O(1) associative searches of both 8-bit and 16-bit tags; tags longer than this are generally more efficient if the generic `FullyAssociativeArrayTags` using standard integer comparisons is used instead.

As a result of this high performance design, each issue queue is limited to 64 entries and the tags to be matched must be between 0 and 255 to fit in 8 bits. The `FullyAssociativeArrayTags16bit` class can be used instead if longer tags are required, at the cost of reduced simulation performance. To enable this, `BIG_ROB` must be defined in `ooocore.h`.

### 19.3.2 Other Designs

It's important to remember that the issue queue design described above is *one* possible implementation out of the many designs currently used in industry and research processors. For instance, in lieu of the collapsing design (used by the Pentium 4 and Power4/5/970), the AMD K8 uses a sequence number tag of the ROB and comparator logic to select the earliest ready instruction. Similarly, the Pentium 4 uses a set of bit vectors (a *dependency matrix*) instead of tag broadcasts to wake up instructions. These other approaches may be implemented by modifying the `IssueQueue` class as appropriate.

## 19.4 Issue

The `issue()` top-level function issues one or more instructions in each cluster from each issue queue every cycle. This function consults the `clusters[`*clusterid*`].issue_width` field defined in `ooocore.h` to determine the maximum number of uops to issue from each cluster. The `issueq_operation_on_cluster_with_res` `iqslot, issue())` macro (Section 19.1) is used to invoke the `issue()` method of the appropriate cluster to select the earliest ready issue queue slot, as described in Section 19.3.

The `ReorderBufferEntry::issue()` method of the corresponding ROB entry is then called to actually execute the uop. This method first makes sure a functional unit is available within the cluster that's capable of executing the uop; if not, the uop is replayed and re-issued again on the next cycle. At this point, the uop's three operands (`ra`, `rb`, `rc`) are read from the physical register file. If any of the operands are invalid, the entire uop is marked as invalid with an `EXCEPTION_Propagate` result and is not further executed. Otherwise, the uop is executed by calling the synthesized execute function for the uop (see Section 17.1).

Loads and stores are handled specially by calling the `issueload()` or `issuestore()` method. Since loads and stores can encounter an mis-speculation (e.g. when a load is erroneously issued before an earlier store to the same addresses), the `issueload()` and `issuestore()` functions can return `ISSUE_MISSPECULATED` to force all uops in program order after the mis-speculated uop to be annulled and sent through the pipeline again. Similarly, if `issueload()` or `issuestore()` return `ISSUE_NEEDS_REPLAY`, issuing from that cluster is aborted since the uop has been replayed in accordance with Section 19.3. It is important to note that loads which miss the cache are considered to complete successfully and do *not* require a replay; their physical register is simply marked as waiting until the load arrives. In both the mis-speculation and replay cases, no further uops from the cluster's issue queue are dispatched until the next cycle.

Branches are handled similar to integer and floating point operations, except that they may cause a mis-speculation in the event of a branch misprediction; this is discussed below.

If the uop caused an exception, we force it directly to the commit stage and not through writeback; this keeps dependencies waiting until they can be properly annulled by the speculation recovery logic. The commit stage will detect the exception and take appropriate action. If the exceptional uop was speculatively executed beyond a branch, it will never reach commit anyway since the bogus branch would have to commit before the exception would even become visible.

***NOTE:*** In PTLsim, all issued uops put their result in the uop's assigned physical register at the time of issue, even though the data technically does not appear there until writeback (i.e. the physical register enters the *written* state). This is done to simplify the simulator implementation; it is assumed that any data "read" from physical registers before writeback is in fact being read from the bypass network instead.

# Chapter 20

# Speculation and Recovery

## 20.1 Misspeculation Cases

PTLsim supports three speculative execution recovery mechanisms to handle various types of speculation failures:

- **Replay** is for scheduling and dependency mis-predictions only. Replayed uops remain in the issue queue so replay is very fast but limited in scope. Replay is described extensively in Section 19.
- **Redispatch** finds the slice of uops in the ROB dependent on a mis-speculated uop and sends only those dependent uops back to the *ready-to-dispatch* state. It is used for load-store aliasing recovery, value mispredictions and other cases where the fetched uops themselves are still valid, but their outputs are invalid.
- **Annulment** removes any uops in program order after (or optionally including) a given uop. It is used for branch mispredictions and misalignment recovery.

## 20.2 Redispatch

### 20.2.1 Redispatch Process

Many types of mis-speculations do not require refetching a different set of uops; instead, any uops dependent on a mis-speculated uop can simply be recirculated through the pipeline so they can re-execute and produce correct values. This process is known as *redispatch*; in the baseline out of order core, it is used to recover from load-store aliasing (Section 22.2.1).

When a mis-speculated ROB is detected, `ROB.redispatch_dependents()` is called. This function identifies the slice of uops that consumed values (directly or indirectly) from the mis-speculated uop, using dependency bitmaps similar to those used in real processors. `ROB.redispatch_dependents(bool` `inclusive`) has an *inclusive* parameter: if false, only the dependent uops are redispatched, not including the mis-speculated uop. This is most useful for value prediction, where the correct value can be directly reinjected into the mis-speculated uop's physical register without re-executing it.

In `ROB.redispatch()`, each affected uop is placed back into the `rob_ready_to_dispatch` state, lways in program order. This helps to avoid deadlocks, since the redispatched slice is given priority for insertion back into the issue queue. The resources associated with each uop (physical register, LDQ/STQ slot, IQ slot, etc.) are also restored to the state they were in immediately after renaming, so they can be properly recirculated through the pipeline as if the uop never issued. Various other issues must also be handled, such as making sure known store-to-load aliasing constraints are preserved across the redispatch so as to avoid infinite replay loops, and branch directions must be corrected if a mispredict caused a fetch unit redirection but that mispredict was in fact based on mis-speculated data.

### 20.2.2 Deadlock Recovery

Redispatch can create deadlocks in cases where other unrelated uops occupy all the issue queue slots needed by the redispatched uops to make forward progress, and there is a circular dependency loop (e.g. on loads and stores not known at the time of the redispatch) that creates a chicken-and-egg problem, thus blocking forward progress.

To recover from this situation, we detect the case where no uops have been dispatched for 64 cycles, yet the `ready_to_dispatch` queue still has valid uops. This situation very rarely happens in practice unless there is a true deadlock. To break up the deadlock, ideally we should only need to redispatch all uops occupying issue queue slots or those already waiting for dispatch - all others have produced a result and cannot block the issue queues again. However, this does not always work in pathological cases, and can sometime lead to repeated deadlocks. Since deadlocks are very infrequent, they can be resolved by just flushing the entire pipeline. This has a negligible impact on performance.

### 20.2.3 Statistical Counters

Several statistical counters are maintained in the PTLsim statistics tree to measure redispatch overhead, in the `ooocore.dispatch.redispatch` node:

- `deadlock-flushes` measures how many times the pipeline must be flushed to resolve a deadlock.
- `trigger-uops` measures how many uops triggered redispatching because of a misspeculation. This number does not count towards the statistics below.
- `dependent-uops` is a histogram of how many uops depended on each trigger uop, not including the trigger uop itself.

## 20.3 Annulment

### 20.3.1 Branch Mispredictions

Branch mispredictions form the bulk of all mis-speculated operations. Whenever the actual RIP returned by a branch uop differs from the `riptaken` field of the uop, the branch has been mispredicted. This means all uops after (but *not* including) the branch must be annulled and removed from all processor structures. The fetch queue (Section 17.1) is then reset and fetching is redirected to the correct branch target. However, all uops in program order before the branch are still correct and may continue executing.

Note that we do *not* just reissue the branch: this would be pointless, as we already know the correct RIP since the branch uop itself has already executed once. Instead, we let it writeback and commit as if it were predicted correctly.

### 20.3.2 Annulment Process

In PTLsim, the `ReorderBufferEntry::annul()` method removes any and all ROBs that entered the pipeline after and optionally including the misspeculated uop (depending on the `keep_misspec_uop` argument). Because this method moves all affected ROBs to the free state, they are instantly taken out of consideration for future pipeline stages and will be dropped on the next cycle.

We must be extremely careful to annul all uops in an x86 macro-op; otherwise half the x86 instruction could be executed twice once refetched. Therefore, if the first uop to annul is not also the first uop in the x86 macro-op, we may have to scan backwards in the ROB until we find the first uop of the macro-op. In this way, we ensure that we can annul the entire macro-op. All uops comprising the macro-op are guaranteed to still be in the ROB since none of the uops can commit until the entire macro-op can commit. Note that this does not apply if the final uop in the macro-op is a branch and that branch uop itself is being retained as occurs with mispredicted branches.

The first uop to annul is determined in the `annul()` method by scanning backwards in time from the excepting uop until a uop with its SOM (start of macro-op) bit is set, as described in Section 5.1. This SOM uop represents the boundary between x86 instructions, and is where we start annulment. The end of the range of uops to annul is at the tail of the reorder buffer.

We have to reconstruct the speculative RRT as it existed just before the first uop to be annulled was renamed. This is done by calling the `pseudocommit()` method of each annulled uop to implement the "fast flush with pseudo-commit" algorithm as follows. First, we overwrite the speculative RRT with the committed RRT. We then *simulate* the commitment of all non-speculative ROBs up to the first uop to be annulled by updating the speculative RRT as if it were the commit RRT. This brings the speculative RRT to the same state as if all in flight nonspeculative operations before the first uop to be annulled had actually committed. Fetching is then resumed at the correct RIP, where new uops are renamed using the recovered speculative RRT.

Other methods of RRT reconstruction (like backwards walk with saved checkpoint values) are difficult to impossible because of the requirement that flag rename tables be restored even if some of the required physical registers with attached flags have since been freed. Technically RRT checkpointing could be used but due to the load/store replay mechanism in use, this would require a checkpoint at every load and store as well as branches. Hence, the forward walk method seems to offer the best performance in practice and is quite simple. The Pentium 4 is believed to use a similar method of recovering from some types of mis-speculations.

After reconstructing the RRT, for each ROB to annul, we broadcast the ROB index to the appropriate cluster's issue queue, allowing the issue queue to purge the slot of the ROB being annulled. Finally, for each annulled uop, we free any resources allocated to it (i.e., the ROB itself, the destination physical register, the load/store queue entry (if any) and so on. Any updates to the branch predictor and return address stack made during the speculative execution of branches are also rolled back.

Finally, the fetch unit is restarted at the correct RIP and uops enter the pipeline and are renamed according to the recovered rename tables and allocated resource maps.

# Chapter 21

# Load Issue

## 21.1 Address Generation

Loads and stores both have their physical addresses computed using the `ReorderBufferEntry::addrgen()` method, by adding the `ra` and `rb` operands. If the load or store is one of the special unaligned fixup forms (`ld.lo`, `ld.hi`, `st.lo`, `st.hi`) described in Section 5.6, the address is re-aligned according to the type of instruction.

At this point, the `check_and_translate()` method is used to translate the virtual address into a mapped physical address using the page tables and TLB. The function of this method varies significantly between userspace-only PTLsim and full system PTLsim/X. In userspace-only PTLsim, the shadow page access tables (Section 11.3) are used to do access checks; the same virtual address is then returned to use as a physical address. In full system PTLsim/X, the real x86 page tables are used to produce the physical address, significantly more involved checks are done, and finally a pointer into PTLsim's mapping of all physical pages is returned (see Section 14.3.1).

If the virtual address is invalid or not present for the specified access type, `check_and_translate()` will return a null pointer. At this point, `handle_common_load_store_exceptions()` is called to take action as follows.

If a given load or store accesses an unaligned address but is not one of the special `ld.lo`/`ld.hi`/`st.lo`/`st.hi` uops described in Section 5.6, the processor responds by first setting the "`unaligned`" bit in the original `TransOp` in the basic block cache, then it annuls all uops after and including the problem load, and finally restarts the fetch unit at the RIP address of the load or store itself. When the load or store uop is refetched, it is transformed into a pair of `ld.lo`/`ld.hi` or `st.lo`/`st.hi` uops in accordance with Section 5.6. This refetch approach is required rather than a simple replay operation since a replay would require allocating two entries in the issue queue and potentially two ROBs, which is not possible with the PTLsim design once uops have been renamed.

If a load or store would cause a page fault for any reason, the `check_and_translate()` function will fill in the `exception` and `pfec` (Page Fault Error Code) variables. These two variables are then placed into the low and high 32 bits, respectively, of the 64-bit result in the destination physical register or store buffer, in place of the actual data. The load or store is then aborted and execution returns to the `ReorderBufferEntry::issue()` method, causing the result to be marked with an exception (`EXCEPTION_PageFaultOnRead` or `EXCEPTION_PageFaultOnWrite`).

One x86-specific complication arises at this point. If a load (or store) uop is the high part (`ld.hi` or `st.hi`) of an unaligned load or store pair, but the actual user address did not overlap any of the high 64 bits accessed by the `ld.hi` or `st.hi` uop, the load or store should be completely ignored, even if the high part overlapped onto an invalid page. This is because it is perfectly legal to do an unaligned load or store at the very end of a page such that the next 64 bit chunk is not mapped to a valid page; the x86 architecture mandates that the load or store execute correctly as far as the user program is concerned.

## 21.2 Store Queue Check and Store Dependencies

After doing these exception checks, the load/store queue (LSQ) is scanned backwards in time from the current load's entry to the LSQ's head. If a given LSQ entry corresponds to a store, the store's address has been resolved and the memory range needed by the load overlaps the memory range touched by the store, the load effectively has a dependency on the earlier store that must be resolved before the load can issue. The meaning of "overlapping memory range" is defined more specifically in Section 22.1.

In some cases, the addresses of one or more prior stores that a load may depend on may not have been resolved by the time the load issues. Some processors will stall the load uop until *all* prior store addresses are known, but this can decrease performance by incorrectly preventing independent loads from starting as soon as their address is available. For this reason, the PTLsim processor model aggressively issues loads as soon as possible unless the load is predicted to frequently alias another store currently in the pipeline. This load/store aliasing prediction technique is described in Section 22.2.1.

In either of the cases above, in which an overlapping store is identified by address but that store's data is not yet available for forwarding to the load, or where a prior store's address has not been resolved but is *predicted* to overlap the load, the load effectively has a data flow dependency on the earlier store. This dependency is represented by setting the load's fourth `rs` operand (`operands[RS]` in the `ReorderBufferEntry`) to the store the load is waiting on. After adding this dependency, the `replay()` method is used to force the load back to the dispatched state, where it waits until the prior store is resolved. After the load is re-issued for a second time, the store queue is scanned again to make sure no intervening stores arrived in the meantime. If a different match is found this time, the load is replayed a third time. In practice, loads are rarely replayed more than once.

## 21.3 Data Extraction

Once the prior store a load depends on (if any) is ready and all the exception checks above have passed, it is time to actually obtain the load's data. This process can be complicated since some bytes in the region accessed by the load could come from the data cache while other bytes may be forwarded from a prior store. If one or more bytes need to be obtained from the data cache, the

L1 cache is probed (via the `caches.probe_cache_and_sfr()` function) to see if the required line is present. If so, and the combination of the forwarded store (if any) and the L1 line fills in all bytes required by the load, the final data can be extracted.

To extract the data, the load unit creates a 64-bit temporary buffer by overlaying the bytes touched by the prior store (if any) on top of the bytes obtained from the cache (i.e., the bytes at the mapped address returned by the `addrgen()` function). The correct word is then extracted and sign extended (if required) from this buffer to form the result of the load. Unaligned loads (described in Section 5.6) are somewhat more complex in that both the low and high 64 bit chunks from the `ld.lo` and `ld.hi` uops, respectively, are placed into a 128-bit buffer from which the final result is extracted.

For simulation purposes only, the data to load is immediately accessed and recorded by `issueload()` regardless of whether or not there is a cache miss. This makes the loaded data significantly easier to track. In a real processor, the data extraction process obviously only happens after the missing line actually arrives, however our implementation in no way affects performance.

## 21.4 Cache Miss Handling

If no combination of the prior store's forwarded bytes and data present in the L1 cache can fulfill a load, this is miss and lower cache levels must be accessed. This process is described in Sections 25.2 and 25.3. As far as the core is concerned, the load is completed at this point even if the data has not yet arrived. The issue queue entry for the load can be released since the load is now officially in progress and cannot be replayed. Once the loaded data has arrived, the cache subsystem calls the `OutOfOrderCoreCacheCallbacks::dcache_wakeup()` function, which marks both the physical register and LSQ entry of the load as ready, and places the load's ROB into the *completed* state. This allows the processor to wake up dependents of the load on the next cycle.

# Chapter 22

# Stores

## 22.1 Store to Store Forwarding and Merging

In the PTLsim out of order model, a given store may merge its data with that of a previous store in program order. This ensures that loads which may need to forward data from a store always reference exactly one store queue entry, rather than having to merge data from multiple smaller prior stores to cover the entire byte range being loaded. In this model, physical memory is divided up into 8 byte (64 bit) chunks. As each store issues, it scans the store queue backwards in program order to find the most recent prior store to the same 8 byte aligned physical address. If there is a match, the current store depends on the matching prior store, and cannot complete and forward its data to other consuming loads and stores until the prior store in question also completes. This ensures that the current store's data can be composited on top of the older store's data to form a single up to date 8-byte chunk. As described in Section 18.4, each store queue entry contains a byte mask to indicate which of the 8 bytes in each chunk are currently modified by stores in flight versus those bytes which must come from the data cache.

Technically there are more efficient approaches, such as allowing stores to issue in any order so long as they do not overlap on the basis of individual bytes. However, no modern processor allows such arbitrary forwarding since the circuit complexity involved with scanning the store queue for partial address matches would be prohibitive and slow. Instead, most processors only support store to load forwarding when a single larger prior store covers the entire byte range accessed by a smaller or same sized load; all other combinations stall the load until the overlapping prior stores commit to the data cache.

The store inheritance scheme used by PTLsim (described first) is an improvement to the more common "stall on size mismatch" scheme above, but may incur more store dependency replays (since stores now depend on other stores when they target the same 8-byte chunk) compared to a stall on size mismatch scheme. As a case study, the Pentium 4 processor (Prescott core) implements a combination of these approaches.

## 22.2 Split Phase Stores

The `ReorderBufferEntry::issuestore()` function is responsible for issuing all store uops. Stores are unusual in that they can issue even if their `rc` operand (the value to store) is not ready at the same time as the `ra` and `rb` operands forming the effective address. This property is useful since it allows a store to establish an entry in the store queue as soon as the effective address can be generated, even if the data to store is not ready. By establishing addresses in the store queue as soon as possible, we can avoid performance losses associated with the unnecessary replay of loads that may depend on a store whose address is unavailable at the time the load issues. In effect, this means that each store uop may actually issue twice.

In the first phase issue, which occurs as soon as the `ra` and `rb` operands become ready, the store uop computes its effective physical address, checks that address for all exceptions (such as alignment problems and page faults) and writes the address into the corresponding `LoadStoreQueueEntry` structure before setting its the `addrvalid` bit as described in Section 18.4. If an exception is detected at this point, the `invalid` bit in the store queue entry is set and the destination physical register's `FLAG_inv` flag is set so any attempt to commit the store will fail.

### 22.2.1 Load Queue Search (Alias Check)

The load queue is then searched to find any loads after the current store in program order which have already issued but have done so without forwarding data from the current store. These loads erroneously issued before the current store (now known to overlap the load's address) was able to forward the correct data to the offending load(s). This situation is known as *aliasing*, and is effectively a mis-speculation requiring us to reissue any uops depending on the store. The redispatch method (Section 20.2) is used to re-execute only those uops dependent (either directly or indirectly) on the store.

Since the redispatch process required to correct aliasing violations is expensive and may result in infinite loops, it is desirable to predict in advance which loads and stores are likely to alias each other such that loads predicted to alias are never issued when prior stores in the store queue still have unknown addresses. This works because in most out of order processors, statistically speaking, very few loads alias stores compared to normal loads from the cache. When an aliasing mis-speculation occurs, an entry is added to a small fully associative structure (typically $\leq 16$ entries) called the Load Store Alias Predictor (LSAP). This structure is indexed by a portion of the address of the load instruction that aliased. This allows the load unit to avoid issuing any load uop that matches any address in the LSAP if any prior store addresses are still unresolved; if this is the case, a dependency is created on the first unresolved store such that the load is replayed (and the load and store queues are again scanned) once that store resolves. Similar methods of aliasing prediction are used by the Pentium 4 (Prescott core only) and Alpha 21264.

### 22.2.2 Store Queue Search (Merge Check)

At this point the store queue is searched for prior stores to the same 8-byte block as described above in Section 22.1; if the store depends on a prior store, the scheduler structures are updated to add an additional dependency (in `operands[RS]`) on this prior store before the store is replayed in accordance with Section 19.3 to wait for the prior store to complete. If no prior store is found, or the prior store is ready, the current store is marked as a second phase store by setting the `load_store_second_phase` flag in its ROB entry. Finally, the store is replayed in accordance with Section 19.3.

In the second phase of store uop scheduling, the store uop is only re-issued when all four operands (`ra` + `rb` address, `rc` data and `rs` source store queue entry) are valid. The second phase repeats the scan of the load and store queues described above to catch any loads and stores that may have issued between the first and second phase issues; the store is replayed a third time if necessary. Otherwise, the `rc` operand data is merged with the data from the prior store (if any) store queue entry, and the combined data and bytemask is written into the current store's store queue entry. Finally, the entry's `dataready` bit is set to make the entry available for forwarding to other waiting loads and stores.

The first and second phases may be combined into a single issue without replay if both the address and data operands of the store are all ready at the same time and the prior store (if any) the current store inherits from has already successfully issued.

# Chapter 23

# Forwarding, Wakeup and Writeback

## 23.1 Forwarding and the Clustered Bypass Network

Immediately after each uop is issued and the `ReorderBufferEntry::issue()` method actually generates its result, the `cycles_left` field of the ROB is set to the expected latency of the uop (e.g. between 1 and 5 cycles). The uop is then moved to the *issued* state and placed on the `rob_issued_list`. Every cycle, the `complete()` method iterates through each ROB in issued state and decrements its `cycles_left` field. If `cycles_left` becomes zero, the corresponding uop has completed execution. The ROB is moved to the *completed* state (on `rob_completed_list`) and its physical register or store queue entry is moved to the `bypass` state so newly dispatched uops do not try to wait for it.

The `transfer()` function is also called every cycle. This function examines the list of ROBs in the *completed* state and is responsible for broadcasting the completed ROB's tag (ROB index) to the issue queues. Because of clustering (Section 19.1), some issue queues will receive the broadcast later than others. Specifically, the ROB's `forward_cycle` field determines which issue queues and remote clusters are visible `forward_cycle` cycles after the uop completed. The `forward()` method, called by `transfer()` for each uop in the *completed* state, indexes into a lookup table `forward_at_cycle_lut[`*cluster*`][`*forward_cycle*`]` to get a bitmap of which remote clusters are accessible `forward_cycle` cycles after he uop completed, relative to the original cluster.the uop issued in. The `IssueQueue::broadcast()` method (Section 19.3) is then called for each applicable cluster to wake up any operands of uops in that cluster waiting on the newly completed uop.

The `MAX_FORWARDING_LATENCY` constant (in `ooocore.h`) specifies the maximum number of cycles between any two clusters. After the ROB has progressed through `MAX_FORWARDING_LATENCY` cycles in the *completed* state, it is moved to the `ready-to-writeback` state, effectively meaning the result has arrived at the physical register file and is eligible for writeback in the next cycle.

## 23.2 Writeback

Every cycle, the `writeback()` function scans the list of ROBs in the *ready-to-writeback* state and selects at most `WRITEBACK_WIDTH` results to write to the physical register file. The `forward()` method is first called one final time to catch the corner case in which a dependent uop was dispatched while producer uop was waiting in the *ready-to-writeback* state.

As mentioned in Section 19.4, for simulation purposes only, each uop puts its result directly into its assigned physical register at the time of issue, even though the data technically does not appear there until writeback. This is done to simplify the simulator implementation; it is assumed that any data "read" from physical registers before writeback is in fact being read from the bypass network instead. Therefore, no actual data movement occurs in the `writeback()` function; its sole purpose is to place the uop's physical register into the written state (via the `PhysicalRegister::writeback()` method) and to move the ROB into its terminal state, *ready-to-commit*.

# Chapter 24

# Commitment

## 24.1 Introduction

The commit stage examines uops from the head of the ROB, blocks until all uops comprising a given x86 instruction are ready to commit, commits the results of those uops to the architectural state and finally frees the resources associated with each uop.

## 24.2 Atomicity of x86 instructions

The x86 architecture specifies *atomic execution* for all distinct x86 instructions. This means that since each x86 instruction may be comprised of multiple uops; none of these uops may commit until *all* uops in the instruction are ready to commit. In PTLsim, this is accomplished by checking if the uop at the head of the ROB (next to commit) has its SOM (start of macro-op) bit set. If so, the ROB is scanned forwards from the SOM uop to the next uop in program order with its EOM (end of macro-op) bit set. If all uops in this range are ready to commit and exception-free, the SOM uop is allowed to commit, effectively unlocking the ROB head pointer until the next uop with a SOM bit set is encountered. However, any exception in any uop comprising the x86 instruction at the head of the ROB causes the pipeline to be flushed and an exception to be taken. Similarly, external interrupts are only acknowledged at the boundary between x86 instructions (i.e. after the EOM uop of each instruction).

## 24.3 Commitment

As each uop commits, it may update several components of the architectural state.

Integer ALU and floating point uops obviously update their destination architectural register (*rd*). In PTLsim, this is done by simply updating the committed register rename table (`commitrrt`) rather than actually copying register values. However, the old physical register mapped to architectural register *rd* will normally become inaccessible after the Commit RRT mapping for *rd* is overwritten with the committing uop's physical register index. The old physical register previously mapped to *rd* can then be freed. Technically physical registers allocated to intermediate uops (such as those used to hold temporary values) can be immediately freed without updating any Commit RRT entries, but for consistency we do not do this.

In PTLsim, a physical register is freed by moving it to the `PHYSREG_FREE` state. Unfortunately for various reasons related to long pipelines and the renaming of x86 flags, register reclamation is not so simple, but this will be discussed below in Section 24.5.

Some uops may also commit to a subset of the x86 flags, as specified in the uop encoding. For these uops, in theory no rename tables need updating, since the flags can be directly masked into the `REG_flags` architectural pseudo-register. Should the pipeline be flushed, the rename table entries for the ZAPS, CF, OF flag sets will all be reset to point to the `REG_flags` pseudo-register anyway. However, for the speculation recovery scheme described in Section 20.3.2, the `REG_zf`, `REG_cf`, and `REG_of` commit RRT entries are updated as well to match the updates done to the speculative RRT.

Branches and jumps update the `REG_rip` pseudo architectural register, while all other uops simply increment `REG_rip` by the number of bytes in the x86 instruction being committed. The number of bytes (1-15) is stored in a 4-bit `bytes` field of each uop in each x86 instruction.

Stores commit to the architectural state by writing directly to the data cache, which in PTLsim is equivalent to writing into real physical memory. Remember that a series of stores into a given 64-bit chunk of memory are merged within the store queue to the store uop's corresponding STQ entry as the store uop issues, so the commit unit always writes 64 bits to the cache at a time. The byte mask associated with the STQ entry of the store uop is used to only update the modified bytes in each chunk of memory in program order.

## 24.4 Additional Commit Actions for Full System Use

In full system PTLsim/X, several additional actions must be taken at commit time:

- Self modifying code checks must be done using `smc_isdirty(mfn)`, as described in Section 6.4.
- Stores must set the dirty bit on the target physical page, using the `smc_setdirty(mfn)` function (so as to properly notify subsequent instructions of self modifying code).
- The x86 page table accessed and dirty bits must be updated whenever a load or store commits, using the `Context.update_pte_acc_dirty()` function.
- If an interrupt is pending, and we have just committed the last uop in an atomic x86 instruction, we can now safely service it.

## 24.5 Physical Register Recycling Complications

### 24.5.1 Problem Scenarios

In some processor designs, it is not always possible to immediately free the physical register mapped to a given architectural register when that old architectural register mapping is overwritten during commit as described above. Out of order x86 processors must maintain three separate rename table entries for the ZAPS, CF, OF flags in addition to the register rename table entry, any or all of which may be updated when uops rename and retire, depending on the uop's flag renaming semantics (see Section 5.4), For this reason, even though a given physical register value may become inaccessible and hence dead at commit time, the flags associated with that physical register are frequently still referenced within the pipeline, so the physical register itself must remain allocated.

Consider the following specific example, with uops listed in program order:

- `sub rax = rax,rbx`
  Assign RRT[`rax`] = phys reg r0
  Assign RRT[`flags`] = *r0* (since SUB all updates flags)

- `mov rax = rcx`
  Assign RRT[`rax`] = phys reg r1
  *No flags renamed:* MOV never updates flags, so RRT[`flags`] is still *r0*.

- `br.e target`
  Depends on flags attached to *r0*, even though actual architectural register (`rax`) for *r0* has already been overwritten in the commit RRT by the MOV's commit. We cannot free *r0* since the BR uop might not have issued yet.

This situation only happens with instruction sets like x86 (and SPARC or even PowerPC to some extent) which support writing flags (particularly multiple independent flags) and data in a single instruction.

### 24.5.2 Reference Counting

For these reasons, we need to prevent U2's register from being freed if it is still referenced by anything still in the pipeline; the normal reorder buffer mechanism cannot always handle this situation in a very long pipeline.

One solution (the one used by PTLsim) is to give each physical register a reference counter. Physical registers can be referenced from three structures: as operands to ROBs, from the speculative RRT, and from the committed RRT. As each uop operand is renamed, the counter for the corresponding physical register is incremented by calling the `PhysicalRegister::addref()` method. As each uop commits, the counter for each of its operands is decremented via the `PhysicalRegister::unref()` method. Similarly, `unref()` and `addref()` are used whenever an entry in the speculative RRT or commit RRT is updated. During mis-speculation recovery (see Section 20.3.2), `unref()` is also used to unlock the operands of uops slated for annulment. Finally, `unref()` and `addref()` are used when loads and stores need to add a new dependency on a waiting store queue entry (see Sections 21 and 22.2).

As we update the committed RRT during the commit stage, the old register R mapped to the destination architectural register A of the uop being committed is examined. The register R is only moved to the *free* state iff its reference counter is zero. Otherwise, it is moved to the *pendingfree* state. The hardware examines the counters of *pendingfree* physical registers every cycle and moves physical registers to the *free* state only when their counters become zero and they are in the *pendingfree* state.

### 24.5.3 Hardware Implementation

The hardware implementation of this scheme is straightforward and low complexity. The counters can have a very small number of bits since it is very unlikely a given physical register would be referenced by all 100+ uops in the ROB; 3 bits should be enough to handle the typical maximum of < 8 uops sharing a given operand. Counter overflows can simply stall renaming or flush the pipeline since they are so rare.

The counter table can be updated in bulk each cycle by adding/subtracting the appropriate sum or just adding zero if the corresponding register wasn't used. Since there are several stages between renaming and commit, the same counter is never both incremented and decremented in the same cycle, so race conditions are not an issue.

In real processors, the Pentium 4 uses a scheme similar to this one but uses bit vectors instead. For smaller physical register files, this may be a better solution. Each physical register has a bit vector with one bit per ROB entry. If a given physical register P is still used by ROB entry E in the pipeline, P's bit vector bit R is set. Register P cannot be freed until all bits in its vector are zero.

## 24.6 Pipeline Flushes and Barriers

In some cases, the entire pipeline must be empty after a given uop commits. For instance, a *barrier* uop, represented by any `br.p` (branch private) uop, will stall the frontend when first renamed, and when committed (at which point it is the only uop in the pipeline), it will call `flush_pipeline()` to restart fetching at the appropriate RIP. Exceptions have a similar effect when they reach the commit stage. After doing this, the current architectural registers must be copied into the externally visible `ctx.commitarf[]` array, since normally the architectural registers are scattered throughout the physical register file. Fortunately, the commit stage also updates `ctx.commitarf[]` in parallel with the commit RRT, even though the `commitarf` array is never actually read by the out of order core. Interrupts are a special case of barriers, the difference being they can be serviced after *any* x86 instruction commits its last uop.

At this point, the `handle_barrier()`, `handle_exception()` or `handle_interrupt()` function is called to actually communicate with the world outside the out of order core. In the case of `handle_barrier()`, generally this involves executing native code inside PTLsim to redirect execution into or out of the kernel, or to service a very complex x86 instruction (e.g. `cpuid`, floating point save or restore, etc). For `handle_exception()`, on userspace-only PTLsim, the simulation is stopped and the user is notified that a genuine user visible (non-speculative) exception reached the commit stage. In contrast, on full system PTLsim/X, exceptions are little more than jumps into kernel space; this is described in detail in Chapter 14.

If execution can continue after handling the barrier or exception, the `external_to_core_state()` function is called to completely reset the out of order core using the state stored in `ctx.commitarf[]`. This involves allocating a fixed physical register for each of the 64 architectural registers in `ctx.commitarf[]`, setting the speculative and committed rename tables to their proper cold start values, and resetting all reference counts on physical registers as appropriate. If the processor is configured with multiple physical register files (Section 18.3), the initial physical register for each architectural register is allocated in the first physical register file only (this is configurable by modifying `external_to_core_state()`). At this point, the main simulation loop can resume as if the processor had just restarted from scratch.

# Chapter 25

# Cache Hierarchy

The PTLsim cache hierarchy model is highly flexible and can be used to model a wide variety of contemporary cache structures. The cache subsystem (defined in `dcache.h` and implemented by `dcache.cpp`) by default consists of four levels:

- **L1 data cache** is directly probed by all loads and stores
- **L1 instruction cache** services all instruction fetches
- **L2 cache** is shared between data and instructions, with data paths to both L1 caches
- **L3 cache** is also shared and is optionally present
- **Main memory** is considered infinite in size but still has configurable characteristics

These cache levels are listed in order from highest level (closer to the core) to lowest level (far away). The cache hierarchy is assumed to be *inclusive*, i.e. any data in higher levels is assumed to always be present in lower levels. Additionally, the cache levels are generally *write-through*, meaning that every store updates all cache levels, rather than waiting for a dirty line to be evicted. PTLsim supports a 48-bit virtual address space and 40-bit physical addresses (full system PTL-sim/X only) in accordance with the x86-64 standard.

## 25.1 General Configurable Parameters

All caches support configuration of:

- Line size in bytes. Any power of two size is acceptable, however the line size of a lower cache level must be the same or larger than any line size of a higher level cache. For example, it is illegal to have 128 byte L1 lines with 64 byte L2 lines.
- Set count may be any power of two number. The total cache size in bytes is of course (line size) $\times$ (set count)$\times$ (way count)

- Way count (associativity) may be any number from 1 (direct mapped) up to the set count (fully associative). Note that simulation performance (and clock speed in a real processor) will suffer if the associativity is too great, particularly for L1 caches.
- Latency in cycles from a load request to the arrival of the data.

In `dcache.h`, the two base classes `CacheLine` and `CacheLineWithValidMask` are interchangeable, depending on the model being used. The `CacheLine` class is a standard cache line with no actual data (since the bytes in each line are simply held in memory for simulation purposes).

The `CacheLineWithValidMask` class adds a bitmask specifying which bytes within the cache line contain valid data and which are unknown. This is useful for implementing "no stall on store" semantics, in which stores simply allocate a new way in the appropriate set but only set the valid bits for those bytes actually modified by the store. The rest of the cache line not touched by the store can be brought in later without stalling the processor (unless a load tries to access them); this is PTLsim's default model. Additionally, this technique may be used to implement sectored cache lines, in which the line fill bus is smaller than the cache line size. This means that groups of bytes within the line may be filled over subsequent cycles rather than all at once.

The `AssociativeArray` template class in `logic.h` forms the basis of all caches in PTLsim. To construct a cache in which specific lines can be locked into place, the `LockableAssociativeArray` template class may be used instead. Finally, the `CommitRollbackCache` template class is useful for creating versions of PTLsim with cache level commit/rollback support for out of order commit, fault recovery and advanced speculation techniques.

The various caches are defined in `dcache.h` by specializations of these template classes. The classes are `L1Cache`, `L1ICache`, `L2Cache` and `L3Cache`.

## 25.2 Initiating a Cache Miss

As described in Section 21, in the out of order core model, the `issueload()` function determines if some combination of a prior store's forwarded bytes (if any) and data present in the L1 cache can fulfill a load. If not, this is a miss and lower cache levels must be accessed. In this case, a `LoadStoreInfo` structure (defined in `dcache.h`) is prepared with various metadata about the load, including which ROB entry and physical register to wake up when the load arrives, its size, alignment, sign extension properties, prefetch properties and so on. The `issueload_slowpath()` function (defined in `dcache.cpp`) is then called with this information, the physical address to load and any data inherited from a prior store still in the pipeline. The `issueload_slowpath()` function moves the load request out of the core pipeline and into the cache hierarchy.

The *Load Fill Request Queue* (LFRQ) is a structure used to hold information about any outstanding loads that have missed any cache level. The LFRQ allows a configurable number of loads to be outstanding at any time and provides a central control point between cache lines arriving from the L2 cache or lower levels and the movement of the requested load data into the processor core to dependent instructions. The `LoadFillReq` structure, prepared by `issueload_slowpath()`, contains all the data needed to return a filled load to the core: the physical address of the load, the data and bytemask already known so far (e.g. forwarded from a prior store) and the LoadStoreInfo metadata described above.

The *Miss Buffer* (MB) tracks all outstanding cache lines, rather than individual loads. Each MB slot uses a bitmap to track one or more LFRQ entries that need to be awakened when the missing cache line arrives. After adding the newly created LoadFillReq entry to the LFRQ, the MissBuffer::initiate_miss() method uses the missing line's physical address to allocate a new slot in the miss buffer array (or simply uses an existing slot if a miss was already in progress on a given line). In any case, the MB's wakeup bitmap is updated to reflect the new LFRQ entry referring to that line. Each MB entry contains a cycles field, indicating the number of cycles remaining for that miss buffer before it can be moved up the cache hierarchy until it reaches the core. Each entry also contains two bits (icache and dcache) indicating which L1 caches to which the line should eventually be delivered; this is required because a single L2 line (and corresponding miss buffer) may be referenced by both the L1 data and instruction caches.

In initiate_miss(), the L2 and L3 caches are probed to see if they contain the required line. If the L2 has the line, the miss buffer is placed into the STATE_DELIVER_TO_L1 state, indicating that the line is now in progress to the L1 cache. Similarly, an L2 miss but L3 hit results in the STATE_DELIVER_TO_L2 state, and a miss all the way to main memory results in STATE_DELIVER_TO_L3.

In the very unlikely event that either the LFRQ slot or miss buffer are full, an exception is returned to out of order core, which typically replays the affected load until space in these structures becomes available. For prefetch requests, only a miss buffer is allocated; no LFRQ slot is needed.

## 25.3 Filling a Cache Miss

The MissBuffer::clock() method implements all synchronous state transitions. For each active miss buffer, the cycles counter is decremented, and if it becomes zero, the MB's current state is examined. If a given miss buffer was in the STATE_DELIVER_TO_L3 state (i.e. in progress from main memory) and the cycle counter just became zero, a line in the L3 cache is validated with the incoming data (this may involve evicting another line in the same set to make room). The MB is then moved to the next state up the cache hierarchy (i.e. STATE_DELIVER_TO_L2 in this example) and its cycles field is updated with the latency of the cache level it is now leaving (e.g. L3_LATENCY in this example).

This process continues with successive levels until the MB is in the STATE_DELIVER_TO_L1 state and its cycles field has been decremented to zero. If the MB's dcache bit is set, the L1 corresponding line is validated and the lfrq.wakeup() method is called to invoke a new state machine to wake up any loads waiting on the recently filled line (as known from the MB's lfrqmap bitmap). If the MB's icache bit was set, the line is validated in the L1 instruction cache, and the PerCoreCacheCallbacks::icache_wakeup() callback is used to notify the out of order core's fetch stage that it may probe the cache for the missing line again. In any case, the miss buffer is then returned to the unused state.

Each LFRQ slot can be in one of three states: *free*, *waiting* and *ready*. LFRQ slots remain in the *waiting* state as long as they are referenced by a miss buffer; once the lfrq.wakeup()

method is called, all slots affiliated with that miss buffer are moved to the *ready* state. The `LoadFillRequestQueue::clock()` method finds up to `MAX_WAKEUPS_PER_CYCLE` LFRQ slots in the *ready* state and wakes them up by calling the `PerCoreCacheCallbacks::dcache_wakeup()` callback with the saved `LoadStoreInfo` metadata. The out of order core handles this callback as described in Section 21.4.

For simulation purposes only, the value to be loaded is immediately recorded as soon as the load issues, independent of the cache hit or miss status. In real hardware, the LFRQ entry data would be used to extract the correct bytes from the newly arrived line and perform sign extension and alignment. If the original load required bytes from a mixture of its source store buffer and the data cache, the SFR data and mask fields in the LFRQ entry would be used to perform this merging operation. The data would then be written into the physical register specified by the `LoadStoreInfo` metadata and that register would be marked as ready before sending a signal to the issue queues to wake up dependent operations.

In some cases, the out of order core may need to annul speculatively executed loads. The cache subsystem is notified of this through the `annul_lfrq_slot()` function called by the core. This function clears the specified LFRQ slot in each miss buffer's lfrqmap entry (since that slot should no longer be awakened now that it has been annulled), and frees the LFRQ entry itself.

## 25.4 Translation Lookaside Buffers

The following section applies to full system PTLsim/X only. The userspace version of PTLsim does not model TLBs since doing so would be inaccurate: it is physically impossible to model TLB miss delays without actually walking real page tables and encountering the associated cache misses. For more information, please see Section 14.3.1 concerning page translation in PTLsim/X.

# Chapter 26

# Branch Prediction

## 26.1 Introduction

PTLsim provides a variety of branch predictors in `branchpred.cpp`. The branch prediction subsystem is relatively independent of the core simulator and can be treated as a black box, so long as it implements the interfaces in `branchpred.h`.

The branch prediction subsystem always contains at least three distinct predictors for the three main classes of branches:

- *Conditional Branch Predictor* returns a boolean (taken or not taken) for each conditional branch (`br.cc` uop)
- *Branch Target Buffer* (BTB) predicts indirect branch (`jmp` uop) targets
- *Return Address Stack* (RAS) predicts return instructions (i.e. specially marked indirect `jmp` uops) based on prior calls
- Unconditional branches (`bru`) are never predicted since their destination is explicitly encoded.

All these predictors are accessed by the core through the `BranchPredictorInterface` object. Based on the opcode and other uop information, the core determines the type flags of each branch uop:

- `BRANCH_HINT_UNCOND` for unconditional branches. These are never predicted since the destination is implied.
- `BRANCH_HINT_COND` for conditional branches.
- `BRANCH_HINT_INDIRECT` for indirect branches, including returns.
- `BRANCH_HINT_CALL` for calls (both direct and indirect). This implies that the return address of the call should be a should be pushed on the RAS.

- `BRANCH_HINT_RET` for returns (indirect branches). This implies that the return address should be taken from the top RAS stack entry, not the BTB.

Multiple flags may be present for each uop (for instance, `BRANCH_HINT_RET` and `BRANCH_HINT_INDIRECT` are both used for the `jmp` uop terminating an x86 `ret` instruction).

To make a prediction at fetch time, the core calls the `BranchPredictorInterface::predict()` method, passing it a `PredictorUpdate` structure. This structure is carried along with each uop until it retires, and contains all the information needed to eventually update the branch predictor at the end of the pipeline. The contents will vary depending on the predictor chosen, but in general this structure contains pointers into internal predictor counter tables and various flags. The `predict()` method fills in this structure.

As each uop commits, the `BranchPredictorInterface::update()` method is passed the uop's saved `PredictorUpdate` structure and the branch outcome (expected target RIP versus real target RIP) so the branch predictor can be updated. In PTLsim, predictor updates only occur at retirement to avoid corruption caused by speculative instructions.

## 26.2 Conditional Branch Predictor

The PTLsim conditional branch predictor is the most flexible predictor, since it can be easily replaced. The default predictor implemented in `branchpred.cpp` is a selection based predictor. In essence, two separate predictors are maintained. The *history predictor* hashes a shift register of previously predicted branches into a table slot; this slot returns whether or not the branch with that history is predicted as taken. PTLsim supports various combinations of the history and branch address to provide *gshare* based semantics. The *bimodal predictor* is simpler; it uses 2-bit saturating counters to predict if a given branch is likely to be taken. Finally, a *selection predictor* specifies which of the two predictors is more accurate and should be used for future predictions. This style of predictor, sometimes called a *McFarling predictor*, has been described extensively in the literature and variations are used by most modern processors.

Through the `CombinedPredictor` template class, the user can specify the sizes of all the tables (history, bimodal, selector), the history depth, the method in which the global history and branch address are combined and so on. Alternatively, the conditional branch predictor can be replaced with something entirely different if desired.

## 26.3 Branch Target Buffer

The Branch Target Buffer (BTB) is essentially a small cache that maps indirect branch RIP addresses (i.e., `jmp` uops) into predicted target RIP addresses. It is set associative, with a user configurable number of sets and ways. In PTLsim, the BTB does not take into account any indirect branch history information. The BTB is a nearly universal structure in branch prediction; see the literature for more information.

## 26.4 Return Address Stack

The Return Address Stack (RAS) predicts the target address of indirect jumps marked with the `BRANCH_HINT_RET` flag. Whenever the `BRANCH_HINT_RET` flag is passed to the predict() method, the top RAS stack entry is returned as the predicted target, overriding anything in the BTB.

Unlike the conditional branch predictor and BTB, the RAS updated speculatively in the frontend pipeline, before the outcome of calls and returns are known. This allows better performance when closely spaced calls and returns must be predicted as they are fetched, before either the call or corresponding return have actually executed. However, when called with the `BRANCH_HINT_RET` flag, the `predict()` method only returns the RIP at the top of the RAS, but does not push or pop the RAS. This must be done after the corresponding `bru` or `jmp` (for direct and or indirect calls, respectively) or `jmp` (for returns) uop is actually allocated in the ROB.

This approach is required since the RAS is speculatively updated: if uops must be annulled (because of branch mispredictions or mis-speculations), the annulment occurs by walking backwards in the ROB until the excepting uop is encountered. However, if the RAS were updated during the fetch stage, some uops may not be in the ROB yet and hence the rollback logic cannot undo speculative changes made to the RAS by these uops. This causes the RAS to get out of alignment and performance suffers.

To solve this problem, the RAS is only updated in the allocate stage immediately after fetch. In the out of order core's `rename()` function, the `BranchPredictorInterface::updateras()` method is called to either push or pop an entry from the RAS (calls push entries, returns pop entries). Unlike the conditional branch predictor and BTB, this is the only place the RAS is updated, rather than performing updates at commit time.

If uops must be annulled, the `ReorderBufferEntry::annul()` method calls the `BranchPredictorInterface::annu` method with the `PredictorUpdate` structure for each uop it encounters in reverse program order. This method effectively undoes whatever change was made to the RAS when the `updateras()` method was called with the same `PredictorUpdate` information during renaming and allocation. This is possible because `updateras()` saves checkpoint information (namely, the old RAS top of stack and the value at that stack slot) before updating the RAS; this allows the RAS state to be rolled backwards in time as uops are annulled in reverse program order. At the end of the annulment process when fetching is restarted at the correct RIP, the RAS state should be identical to the state that existed before the last uop to be annulled was originally fetched.

# Part V

# Appendices

# Chapter 27

# PTLsim uop Reference

The following sections document the semantics and encoding of each micro-operation (uop) supported by the PTLsim processor core. The `opinfo[]` table in `ptlhwdef.cpp` and constants in `ptlhwdef.h` give actual numerical values for the opcodes and other fields described below.

## Merging Rules

| Mnemonic | Syntax | Operation |
|---|---|---|
| `op` | `rd = ra,rb` | rd = ra ← (ra *op* rb) |

**Merging Rules:**

The x86 compatible ALUs implement operations on 1, 2, 4 or 8 byte quantities. Unless otherwise indicated, all operations take a 2-bit size shift field (`sz`) used to determine the effective size in bytes of the operation as follows:

- **sz = 0:** Low byte of *rd* is set to the 8-bit result; high 7 bytes of *rd* are set to corresponding bytes of *ra*.
- **sz = 1:** Low two bytes of *rd* is set to the 16-bit result; high 6 bytes of *rd* are set to corresponding bytes of *ra*.
- **sz = 2:** Low four bytes of *rd* is set to the 32-bit result; high 4 bytes of *rd* are cleared to zero in accordance with x86-64 zero extension semantics. The *ra* operand is unused and should be `REG_zero`.
- **sz = 3:** All 8 bytes of *rd* are set to the 64-bit result. *ra* is unused and should be `REG_zero`.

Flags are calculated based on the *sz*-byte value produced by the ALU, not the final 64-bit result in *rd*.

## Other Pseudo-Operators

The descriptions in this reference use various pseudo-operators to describe the semantics of each uop. These operators are described below.

**EvalFlags(*ra*)**

The *EvalFlags* pseudo-operator evaluates the ZAPS, CF, OF flags attached to the source operand *ra* in accordance with the type of condition code evaluation specified by the uop. The operator returns 1 if the evaluation is true; otherwise 0 is returned.

**SignExt(*ra*, N)**

The *SignExt* operator sign extends the ra operand by the number of bits specified by N. Specifically, bit *ra*[N] is copied to all high order bits from bit 63 down to bit *N*. If N is not specified, it is assumed to mean the number of bits in the effective size of the uop's result (as described under Merging Rules).

**MergeWithSFR(mem, sfr)**

The *MergeWithSFR* pseudo-operator is described in the reference page for load uops.

**MergeAlign(mem, sfr)**

The *MergeAlign* pseudo-operator is described in the reference page for load uops.

`mov and or xor andnot ornot nand nor eqv`

## Logical Operations

| **Mnemonic** | **Syntax** | **Operation** |
|---|---|---|
| `mov` | `rd = ra,rb` | rd = ra ← rb |
| `and` | `rd = ra,rb` | rd = ra ← ra & rb |
| `or` | `rd = ra,rb` | rd = ra ← ra \| rb |
| `xor` | `rd = ra,rb` | rd = ra ← ra ^ rb |
| `andnot` | `rd = ra,rb` | rd = ra ← (~ra) & rb |
| `ornot` | `rd = ra,rb` | rd = ra ← (~ra) \| rb |
| `nand` | `rd = ra,rb` | rd = ra ← ~(ra & rb) |
| `nor` | `rd = ra,rb` | rd = ra ← ~(ra \| rb) |
| `eqv` | `rd = ra,rb` | rd = ra ← ~(ra ^ rb) |

**Notes:**

- All operations merge the ALU result with *ra* and generate flags in accordance with the standard x86 merging rules described previously.

`add sub addadd addsub subadd subsub addm subm addc subc`

## Add and Subtract

| **Mnemonic** | **Syntax** | **Operation** |
|---|---|---|
| `add` | `rd = ra,rb` | rd = ra ← ra + rb |
| `sub` | `rd = ra,rb` | rd = ra ← ra - rb |
| `adda` | `rd = ra,rb,rc*S` | rd = ra ← ra + rb + (rc << S) |
| `adds` | `rd = ra,rb,rc*S` | rd = ra ← ra - rb + (rc << S) |
| `addm` | `rd = ra,rb,rc` | rd = ra ← (ra + rb) & rc |
| `subm` | `rd = ra,rb,rc` | rd = ra ← (ra - rb) & rc |
| `addc` | `rd = ra,rb,rc` | rd = ra ← (ra + rb) + rc.cf |
| `subc` | `rd = ra,rb,rc` | rd = ra ← (ra - rb) - rc.cf |

**Notes:**

- All operations merge the ALU result with *ra* and generate flags in accordance with the standard x86 merging rules described previously.
- The `adda` and `adds` uops are useful for small shifts and x86 three-operand `LEA`-style address generation.
- The `addc` and `subc` uops use only the carry flag field of their rc operand; the value is unused.
- The `addm` and `subm` uops mask the result by the immediate in *rc*. They are used in microcode for modular stack arithmetic.

`sel`

## Conditional Select

| Mnemonic | Syntax | Operation |
|---|---|---|
| `sel.`*`cc`* | `rd = ra,rb,(rc)` | rd = ra ← (EvalFlags(rc)) ? rb : ra |

**Notes:**

- ***cc*** is any valid condition code flag evaluation
- The `sel` uop merges the selected operand with *ra* in accordance with the standard x86 merging rules described previously
- The 64-bit result and all flags are treated as a single value for selection purposes, i.e. the flags attached to the selected input are passed to the output
- If one of the (ra, rb) operands is not valid (has `FLAG_INV` set) but the selected operand is valid, the result is valid. This is an exception to the invalid bit propagation rule only when the selected input is valid. If the *rc* operand is invalid, the result is always invalid.
- If any of the inputs are waiting (`FLAG_WAIT` is set), the uop does not issue, even if the selected input was ready. This is a pipeline simplification.
- set rd = (a),b
- sel rd = b,0,1,c

`set`

## Conditional Set

| Mnemonic | Syntax | Operation |
|---|---|---|
| `set.`*`cc`* | `rd = ra,rb,(rc)` | rd = ra ← EvalFlags(rc) ? rb : 0 |

**Notes:**

- ***cc*** is any valid condition code flag evaluation
- The value 0 or 1 is zero extended to the operation size and merged with *rb* in accordance with the standard x86 merging rules described previously (except that `set` uses *rb* as the merge target instead of *ra*)
- Flags attached to *ra* (condition code) are passed through to the output

`set.sub set.and`

## Conditional Compare and Set

| Mnemonic | Syntax | Operation |
|---|---|---|
| `set.sub.`*cc* | `rd = ra,rb,rc` | rd = rc ← EvalFlags(ra - rb) ? 1 : 0 |
| `set.and.`*cc* | `rd = ra,rb,rc` | rd = rc ← EvalFlags(ra & rb) ? 1 : 0 |

**Notes:**

- The `set.sub` and `set.and` uops take the place of a `sub` or `and` uop immediately consumed by a `set` uop; this is intended to shorten the critical path if uop merging is performed by the processor
- ***cc*** is any valid condition code flag evaluation
- The value 0 or 1 is zero extended to the operation size and then merged with *rc* in accordance with the standard x86 merging rules described previously (except that `set.sub` and `set.and` use *rc* as the merge target instead of *ra*)
- Flags generated as the result of the comparison are passed through with the result

`br`

## Conditional Branch

| Mnemonic | Syntax | Operation |
|---|---|---|
| `br.`*`cc`* | `rip = (ra,rb),riptaken,ripseq` | rip = EvalFlags(ra) ? riptaken : ripseq |

**Notes:**

- ***cc*** is any valid condition code flag evaluation
- The `rip` (user-visible instruction pointer register) is reset to one of two immediates. If the flags evaluation is true, the *riptaken* immediate is selected; otherwise the *ripseq* immediate is selected.
- If the flag evaluation is false (i.e., ripseq is selected), the `BranchMispredict` internal exception is raised. The processor should annul all uops after the branch and restart fetching at the RIP specified by the result (in this case, *ripseq*).
- Branches are always assumed to be taken. If the branch is predicted as not taken (i.e. future uops come from the next sequential RIP after the branch), it is the responsibility of the decoder or frontend to swap the *riptaken* and *ripseq* immediates and invert the condition of the branch. All condition encodings can be inverted by inverting bit 0 of the 4-bit condition specifier.
- The destination register should always be `REG_rip`; otherwise this uop is undefined.
- If the target RIP falls within an unmapped page, not present page or a page marked as no-execute (NX), the `PageFaultOnExec` exception is taken.
- No flags are generated by this uop

`br.sub br.and`

## Compare and Conditional Branch

| **Mnemonic** | **Syntax** | **Operation** |
|---|---|---|
| `br.`*cc* | `rip = ra,rb,riptaken,ripseq` | rip = EvalFlags(ra - rb) ? riptaken : ripseq |
| `br.`*cc* | `rip = ra,rb,riptaken,ripseq` | rip = EvalFlags(ra & rb) ? riptaken : ripseq |

**Notes:**

- The `br.sub` and `br.and` uops take the place of a `sub` or `and` uop immediately consumed by a `br` uop; this is intended to shorten the critical path if uop merging is performed by the processor
- ***cc*** is any valid condition code flag evaluation
- The `rip` (user-visible instruction pointer register) is reset to one of two immediates. If the flags evaluation is true, the *riptaken* immediate is selected; otherwise the *ripseq* immediate is selected
- If the flag evaluation is false (i.e., ripseq is selected), the `BranchMispredict` internal exception is raised. The processor should annul all uops after the branch and restart fetching at the RIP specified by the result (in this case, *ripseq*)
- Branches are always assumed to be taken. If the branch is predicted as not taken (i.e. future uops come from the next sequential RIP after the branch), it is the responsibility of the decoder or frontend to swap the *riptaken* and *ripseq* immediates and invert the condition of the branch. All condition encodings can be inverted by inverting bit 0 of the 4-bit condition specifier.
- The destination register should always be `REG_rip`; otherwise this uop is undefined
- If the target RIP falls within an unmapped page, not present page or a page marked as no-execute (NX), the `PageFaultOnExec` exception is taken.
- Flags generated as the result of the comparison are passed through with the result

`jmp`

## Indirect Jump

| Mnemonic | Syntax | Operation |
|---|---|---|
| `jmp` | `rip = ra,riptaken` | rip = ra |

**Notes:**

- The `rip` (user-visible instruction pointer register) is reset to the target address specified by *ra*
- If the *ra* operand does not match the *riptaken* immediate, the `BranchMispredict` internal exception is raised. The processor should annul all uops after the branch and restart fetching at the RIP specified by the result (in this case, *ra*)
- Indirect jumps are always assumed to match the predicted target in *riptaken*. If some other target is predicted, it is the responsibility of the decoder or frontend to set the *riptaken* immediate to that predicted target
- The destination register should always be `REG_rip`; otherwise this uop is undefined
- If the target RIP falls within an unmapped page, not present page or a marked as no-execute (NX), the `PageFaultOnExec` exception is taken.
- No flags are generated by this uop

`jmpp`

## Indirect Jump Within Microcode

| Mnemonic | Syntax | Operation |
|---|---|---|
| `jmpp` | `null = ra,riptaken` | internalrip = ra |

**Notes:**

- The `jmpp` uop redirects uop fetching into microcode not accessible as x86 instructions. The target address (inside PTLsim, not x86 space) is specified by *ra*
- If the *ra* operand does not match the *riptaken* immediate, the `BranchMispredict` internal exception is raised. The processor should annul all uops after the branch and restart fetching at the RIP specified by the result (in this case, *ra*)
- Indirect jumps are always assumed to match the predicted target in *riptaken*. If some other target is predicted, it is the responsibility of the decoder or frontend to set the *riptaken* immediate to that predicted target
- The destination register should always be `REG_rip`; otherwise this uop is undefined
- The user visible rip register is not updated after this uop issues; otherwise it would point into PTLsim space not accessible to x86 code. Updating is resumed after a normal `jmp` issues to return to user code. It is the responsibility of the decoder to move the user address to return to into some temporary register (traditionally `REG_sr2` but this is not required).
- No flags are generated by this uop

`bru`

## Unconditional Branch

| **Mnemonic** | **Syntax** | **Operation** |
|---|---|---|
| `bru` | `rip = riptaken` | rip = riptaken |

**Notes:**

- The `rip` (user-visible instruction pointer register) is reset to the specified immediate. The processor may redirect fetching from the new RIP
- No exceptions are possible with unconditional branches
- If the target RIP falls within an unmapped page, not present page or a marked as no-execute (NX), the `PageFaultOnExec` exception is taken.
- No flags are generated by this uop

`brp`

## Unconditional Branch Within Microcode

| **Mnemonic** | **Syntax** | **Operation** |
|---|---|---|
| `bru` | `null = riptaken` | internalrip = riptaken |

**Notes:**

- The `brp` uop redirects uop fetching into microcode not accessible as x86 instructions. The target address (inside PTLsim, not x86 space) is specified by the *riptaken* immediate
- The `rip` (user-visible instruction pointer register) is reset to the specified *riptaken* immediate. The processor may redirect fetching from the new RIP
- No exceptions are possible with unconditional branches
- The user visible rip register is not updated after this uop issues; otherwise it would point into PTLsim space not accessible to x86 code. Updating is resumed after a normal `jmp` uop issues to return to user code. It is the responsibility of the decoder to move the user address to return to into some temporary register (traditionally `REG_sr2` but this is not required).
- No flags are generated by this uop

`chk`

## Check Speculation

| Mnemonic | Syntax | Operation |
|---|---|---|
| `chk.`*`cc`* | `rd = ra,recrip,extype` | rd = EvalCheck(ra) ? 0 : recrip |

**Notes:**

- The `chk` uop verifies *certain* properties about ra. If this verification check passes, no action is taken. If the check fails, `chk` signals an exception of the user specified type in the *rc* immediate. The result of the `chk` uop in this case is the user specified RIP to recover at after the check failure is handled in microcode. This recovery RIP is saved in the `recoveryrip` internal register.
- This mechanism is intended to allow simple inlined uop sequences to branch into microcode if certain conditions fail, since normally inlined uop sequences cannot contain embedded branches. One example use is in the `REP` series of instructions to ensure that the count is not zero on entry (a special corner case).
- Unlike most conditional uops, the `chk` uop directly checks the numerical value of *ra* against zero, and ignores any attached flags. Therefore, the ***cc*** condition code flag evaluation type is restricted to the subset (e, ne, be, nbe, l, nl, le, nle).
- No flags are generated by this uop

`ld ld.lo ld.hi ldx ldx.lo ldx.hi`

## Load

| Mnemonic | Syntax | Operation |
|---|---|---|
| `ld` | `rd = [ra,rb],sfra` | rd = MergeWithSFR(mem[ra + rb], sfra) |
| `ld.lo` | `rd = [ra+rb],sfra` | rd = MergeWithSFR(mem[floor(ra + rb), 8], sfra) |
| `ld.hi` | `rd = [ra+rb],rc,sfra` | rd = MergeAlign( MergeWithSFR(mem[(floor(ra + rb), 8) + 8], sfra), rc) |

**Notes:**

- *The PTLsim load unit model is described in substantial detail in Section 21; this section only gives an overview of the load uop semantics.*
- The `ld` family of uops loads values from the virtual address specified by the sum *ra* + *rb*. The `ld` form zero extends the loaded value, while the `ldx` form sign extends the loaded value to 64 bits.
- All values are zero or sign extended to 64 bits; no subword merging takes place as with ALU uops. The decoder is responsible for following the load with an explicit `mov` uop to merge 8-bit and 16-bit loads with their old destination register.
- The *sfra* operand specifies the store forwarding register (a.k.a. store buffer) to merge with data from the cache to form the final result. The inherited SFR may be determined dynamically by querying a store queue or can be predicted statically.
- If the load misses the cache, the `FLAG_WAIT` flag of the result is set.
- Load uops do not generate any other condition code flags

**Unaligned Load Support:**

- The processor supports unaligned loads via a pair of `ld.lo` and `ld.hi` uops; an overview can be found in Section 5.6. The alignment type of the load is stored in the uop's cond field (0 = `ld`, 1 = `ld.lo`, 2 = `ld.hi`).
- The `ld.lo` uop rounds down its effective address $\lfloor ra+rb \rfloor$ to the nearest 64-bit boundary and performs the load. The `ld.hi` uop rounds $\lceil ra+rb+8 \rceil$ up to the next 64-bit boundary, performs a load at that address, then takes as its third rc operand the first (`ld.lo`) load's result. The two loads are concatenated into a 128-bit word and the final unaligned data is extracted (and sign extended if the `ldx` form was used).

- Special corner case for when the actual user address (*ra* + *rb*) did not actually require any bytes in the 8-byte range loaded by the `ld.hi` uop (i.e. the load was contained entirely within the low 64-bit aligned chunk). Since it is perfectly legal to do an unaligned load to the very end of the page such that the next 64 bit chunk is not mapped to a valid page, the `ld.hi` uop does not actually access memory; the entire result is extracted from the prior `ld.lo` result in the *rc* operand.

**Exceptions:**

- `UnalignedAccess` if the address (*ra* + *rb*) is not aligned to an integral multiple of the size in bytes of the load. Unaligned loads (`ld.lo` and `ld.hi`) do not generate this exception. Since x86 automatically corrects alignment problems, microcode must handle this exception as described in Section 5.6.
- `PageFaultOnRead` if the virtual address (*ra* + *rb*) falls on a page not accessible to the caller in the current operating mode, or a page marked as not present.
- Various other exceptions and replay conditions may exist depending on the specific processor core model.

`st`

## Store

| Mnemonic | Syntax | Operation |
|---|---|---|
| `st` | `sfrd = [ra,rb],rc,sfra` | sfrd = MergeWithSFR((ra + rb), sfra, rc) |
| `st.lo` | `sfrd = [ra+rb],rc,sfra` | sfrd = MergeWithSFR(floor(ra + rb, 8), sfra, rc) |
| `st.hi` | `sfrd = [ra+rb],rc,sfra` | sfrd = MergeWithSFR(floor(ra + rb, 8) + 8, sfra, rc) |

**Notes:**

- *The PTLsim store unit model is described in substantial detail in Section 22.1; this section only gives an overview of the store uop semantics.*
- The `st` family of uops prepares values to be stored to the virtual address specified by the sum *ra* + *rb*.
- The *sfra* operand specifies the store forwarding register (a.k.a. store buffer) to merge the data to be stored (the *rc* operand) into. The inherited SFR may be determined dynamically by querying a store queue or can be predicted statically, as described in *22.1.*
- Store uops only generate the SFR for tracking purposes; the cache is only written when the SFR is committed.
- The store uop may issue as soon as the *ra* and *rb* operands are ready, even if the *rc* and *sfra* operands are not known. The store must be replayed once these operands become known, in accordance with Section 22.2.
- Store uops do not generate any other condition code flags

**Unaligned Store Support:**

- The processor supports unaligned stores via a pair of `st.lo` and `st.hi` uops; an overview can be found in Section 5.6. The alignment type of the load is stored in the uop's cond field (0 = `st`, 1 = `st.lo`, 2 = `st.hi`).
- Stores are handled in a similar manner, with `st.lo` and `st.hi` rounding down and up to store parts of the unaligned value in adjacent 64-bit blocks.
- The `st.lo` uop rounds down its effective address $\lfloor ra + rb \rfloor$ to the nearest 64-bit boundary and stores the appropriately aligned portion of the `rc` operand that actually falls within that range of 8 bytes. The `ld.hi` uop rounds $\lceil ra + rb + 8 \rceil$ up to the next 64-bit boundary and similarly stores the appropriately aligned portion of the `rc` operand that actually falls within that high range of 8 bytes.

- Special corner case for when the actual user address (*ra* + *rb*) did not actually touch any bytes in the 8-byte range normally written by the `st.hi` uop (i.e. the store was contained entirely within the low 64-bit aligned chunk). Since it is perfectly legal to do an unaligned store to the very end of the page such that the next 64 bit chunk is not mapped to a valid page, the `st.hi` uop does not actually do anything in this case (the bytemask of the generated SFR is set to zero and no exceptions are checked).

**Exceptions:**

- `UnalignedAccess` if the address (*ra* + *rb*) is not aligned to an integral multiple of the size in bytes of the store. Unaligned stores (`st.lo` and `st.hi`) do not generate this exception. Since x86 automatically corrects alignment problems, microcode must handle this exception as described in Section 5.6.
- `PageFaultOnWrite` if the virtual address (*ra* + *rb*) falls on a write protected page, a page not accessible to the caller in the current operating mode, or a page marked as not present.
- `LoadStoreAliasing` if a prior load is found to alias the store (see Section 22.2.1).
- Various other exceptions and replay conditions may exist depending on the specific processor core model.

`ldp ldxp`

## Load from Internal Microcode Space

| Mnemonic | Syntax | Operation |
|---|---|---|
| `ldp` | `rd = [ra,rb]` | rd = MSR[ra+rb] |
| `ldxp` | `rd = [ra+rb]` | rd = SignExt(MSR[ra+rb]) |

**Notes:**

- The `ldp` and `ldxp` uops load values from the internal PTLsim address space not accessible to x86 code. Typically this address space is mapped to internal machine state registers (MSRs) and microcode scratch space. The internal address to access is specified by the sum *ra* + *rb*. The `ldp` form zero extends the loaded value, while the `ldxp` form sign extends the loaded value to 64 bits.
- Load uops do not generate any other condition code flags
- Internal loads may not be unaligned, and never stall or generate exceptions.

`stp`

## Store to Internal Microcode Space

| Mnemonic | Syntax | Operation |
|---|---|---|
| `stp` | `null = [ra,rb],rc` | MSR[ra+rb] = rc |

**Notes:**

- The `stp` uop stores a value to the internal PTLsim address space not accessible to x86 code. Typically this address space is mapped to internal machine state registers (MSRs) and microcode scratch space. The internal address to store is specified by the sum *ra* + *rb* and the value to store is specified by *rc*.
- Store uops do not generate any other condition code flags
- Internal stores may not be unaligned, and never stall or generate exceptions.

`shl shr sar rotl rotr rotcl rotcr`

## Shifts and Rotates

| Mnemonic | Syntax | Operation |
|---|---|---|
| `shl` | `rd = ra,rb,rc` | rd = ra ← (ra << rb) |
| `shr` | `rd = ra,rb,rc` | rd = ra ← (ra >> rb) |
| `sar` | `rd = ra,rb,rc` | rd = ra ← SignExt(ra >> rb) |
| `rotl` | `rd = ra,rb,rc` | rd = ra ← (ra *rotateleft* rb) |
| `rotr` | `rd = ra,rb,rc` | rd = ra ← (ra *rotateright* rb) |
| `rotcl` | `rd = ra,rb,rc` | rd = ra ← ({rc.cf, ra} *rotateleft* rb) |
| `rotcr` | `rd = ra,rb,rc` | rd = ra ← ({rc.cf, ra} *rotateright* rb) |

**Notes:**

- The shift and rotate instructions have some of the most bizarre semantics in the entire x86 instruction set: they may or may not modify flags depending on the rotation count operand, which we may not even know until the instruction issues. This is introduced in Section 5.9.
- The specific rules are as follows:
  - If the count $rb = 0$ is zero, no flags are modified
  - If the count $rb = 1$, both OF and CF are modified, but ZAPS is preserved
  - If the count $rb > 1$, only the CF is modified. (Technically the value in OF is undefined, but on K8 and P4, it retains the old value, so we try to be compatible).
  - Shifts also alter the ZAPS flags while rotates do not.
- For constant counts (immediate *rb* values), the semantics are easy to determine in advance.
- For variable counts (*rb* comes from register), things are more complex. Since the shift needs to determine its output flags at runtime based on both the shift count and the input flags (CF, OF, ZAPS), we need to specify the latest versions in program order of all the existing flags. However, this would require three operands to the shift uop not even counting the value and count operands. Therefore, we use a `collcc` (collect condition code flags, see Section 5.4) uop to get all the most up to date flags into one result, using three operands for ZAPS, CF, OF. This forms a zero word with all the correct flags attached, which is then forwarded as the *rc* operand to the shift. This may add additional scheduling constraints in the case that one of the operands to the shift itself sets the flags, but this is fairly rare. Conveniently, this also lets us directly implement the 65-bit `rotcl`/`rotcr` uops in hardware with little additional complexity.

- All operations merge the ALU result with *ra* and generate flags in accordance with the standard x86 merging rules described previously.
- The specific flags attached to the result depend on the input conditions described above. The user should always assume these uops always produce the latest version of each of the ZAPS, CF, OF flag sets.

`mask`

## Masking, Insertion and Extraction

| Mnemonic | Syntax | Operation |
|---|---|---|
| `mask.`*x*/*z* | `rd = ra,rb,[ms,mc,ds]` | See semantics below |

**Notes:**

- The `mask` uop and its variants are used for generalized bit field extraction, insertion, sign and zero extension using the 18-bit control field in the immediate
- These uops are used extensively within PTLsim microcode, but are also useful if the processor supports dynamically merging a chain of `shr`, `and`, `or` uops.
- The condition code flags (ZAPS, CF, OF) are the flags logically generated by the final AND operation.

### Control Field Format

The 18-bit *rc* immediate has the following three 6-bit fields:

| DS | MC | MS |
|---|---|---|
| 12 | 6 | 0 |

- The `mask` uop and its variants are used for generalized bit field extraction, insertion, sign and zero extension using the 18-bit control field in the immediate

### Operation:

```
M = 1'[(ms+mc-1):ms]
T = (ra & ~M) | ((rb >>> ds) & M)
if (Z) {
  # Zero extend
  rd = ra ← (T & 1'[(ms+mc-1):0])
else if (X) {
  # Sign extend
  rd = ra ← (T[ms+mc-1]) ? (T | 1'[63:(ms+mc)]) : (T & 1'[(ms+mc-1):0])
} else {
  rd = ra ← T
}
```

`bswap`

## Byte Swap

| Mnemonic | Syntax | Operation |
|---|---|---|
| `bswap` | `rd = ra` | rd = ra $\leftarrow$ ByteSwap(rb) |

**Notes:**

- The `bswap` uop reverses the endianness of the *rb* operand. The uop's effective result size determines the range of bytes which are reversed.
- This uop's semantics are identical to the x86 `bswap` instruction.
- This uop does not generate any condition code flags.

`collcc`

## Collect Condition Codes

| Mnemonic | Syntax | Operation |
|---|---|---|
| `collcc` | `rd = ra,rb,rc` | rd.zaps = ra.zaps<br>rd.cf = rb.cf<br>rd.of = rc.of<br>rd = rd.flags |

**Notes:**

- The `collcc` uop collects the condition code flags from three potentially distinct source operands into a single output with the combined condition code flags in both its appended flags and data.
- This uop is useful for collecting all flags before passing them as input to another uop which only supports one source of flags (for instance, the shift and rotate uops).

`movccr movrcc`

## Move Condition Code Flags Between Register Value and Flag Parts

| Mnemonic | Syntax | Operation |
|---|---|---|
| `movccr` | `rd = ra` | rd = ra.flags<br>rd.flags = 0 |
| `movrcc` | `rd = ra` | rd.flags = ra<br>rd = ra |

**Notes:**

- The `movccr` uop takes the condition code flag bits attached to *ra* and copies them into the 64-bit register part of the result.
- The `movrcc` uop takes the low bits of the *ra* operand and moves those bits into the condition code flag bits attached to the result.
- The bits moved consist of the ZF, PF, SF, CF, OF flags
- The WAIT and INV flags of the result are always cleared since the uop would not even issue if these were set in *ra*.

`andcc orcc ornotcc xorcc`

## Logical Operations on Condition Codes

| Mnemonic | Syntax | Operation |
|---|---|---|
| `andcc` | `rd = ra,rb` | rd.flags = ra.flags & rb.flags |
| `orcc` | `rd = ra,rb` | rd.flags = ra.flags \| rb.flags |
| `ornotcc` | `rd = ra,rb` | rd.flags = ra.flags \| (~rb.flags) |
| `xorcc` | `rd = ra,rb` | rd.flags = ra.flags ^ rb.flags |

**Notes:**

- These uops are used to perform logical operations on the condition code flags attached to *ra* and *rb*.
- If the *rb* operand is an immediate, the immediate data is used instead of the flags normally attached to a register operand.
- The 64-bit value of the output is always set to zero.

`mull mulh`

## Integer Multiplication

| Mnemonic | Syntax | Operation |
|---|---|---|
| `mull` | `rd = ra,rb` | rd = ra $\leftarrow$ lowbits(ra $\times$ rb) |
| `mulh` | `rd = ra,rb` | rd = ra $\leftarrow$ highbits(ra $\times$ rb) |

**Notes:**

- These uops multiply *ra* and *rb*, then retain only the low *N* bits or high *N* bits of the result (where N is the uop's effective result size in bits). This result is then merged into *ra*.
- The condition code flags generated by these uops correspond to the normal x86 semantics for integer multiplication (`imul`); the flags are calculated relative to the effective result size.
- The *rb* operand may be an immediate

`bt bts btr btc`

## Bit Testing and Manipulation

| Mnemonic | Syntax | Operation |
|---|---|---|
| `bt` | `rd = ra,rb` | rd.cf = ra[rb] |
| | | rd = ra ← (rd.cf) ? -1 : +1 |
| `bts` | `rd = ra,rb` | rd.cf = ra[rb] |
| | | rd = ra ← ra \| (1 << rb) |
| `btr` | `rd = ra,rb` | rd.cf = ra[rb] |
| | | rd = ra ← ra & (~(1 << rb)) |
| `btc` | `rd = ra,rb` | rd.cf = ra[rb] |
| | | rd = ra ← ra ^ (1 << rb) |

**Notes:**

- These uops test a given bit in *ra* and then atomically modify (set, reset or complement) that bit in the result.
- The CF flag of the output is set to the original value in bit position *rb* of *ra*. Other condition code flag bits in the output are undefined.
- The `bt` (bit test) uop is special: it generates a value of -1 or +1 if the tested bit is 1 or 0, respectively. This is used in microcode for setting up an increment for the `rep` x86 instructions.

`ctz clz`

## Count Trailing or Leading Zeros

| Mnemonic | Syntax | Operation |
|---|---|---|
| `ctz` | `rd = ra` | rd.zf = (rb == 0) |
| | | rd = ra ← (rb) ? LSBIndex(rb) : 0 |
| `clz` | `rd = ra` | rd.zf = (rb == 0) |
| | | rd = ra ← (rb) ? MSBIndex(rb) : 0 |

**Notes:**

- These uops find the bit index of the first '1' bit in *rb*, starting from the lowest bit 0 (for `ctz`) or the highest bit of the data type (for `clz`).
- The result is zero (technically, undefined) if ra is zero.
- The ZF flag of the result is 1 if *rb* was zero, or 0 if *rb* was nonzero. Other condition code flags are undefined.

`ctpop`

## Count Population of ’1’ Bits

| Mnemonic | Syntax | Operation |
|---|---|---|
| `ctpop` | `rd = ra` | rd.zf = (ra == 0)<br>rd = PopulationCount(ra) |

**Notes:**

- The `ctpop` uop counts the number of ’1’ bits in the *ra* operand.
- The ZF flag of the result is 1 if *ra* was zero, or 0 if *ra* was nonzero. Other condition code flags are undefined.

# Floating Point Format and Merging

All floating point uops use the same encoding to specify the precision and vector format of the operands. The uop's *size* field is encoded as follows:

- `00`: Single precision scalar floating point (*op*`fp` mnemonic). The operation is only performed on the low 32 bits (in IEEE single precision format) of the 64-bit inputs; the high 32 bits of the ra operand are copied to the high 32 bits of the output.
- `01`: Single precision vector floating point (*op*`fv` mnemonic). The operation is performed on both 32 bit halves (in IEEE single precision format) of the 64-bit inputs in parallel
- `1x`: Double precision scalar floating point (*op*`fd` mnemonic). The operation is performed on the full 64 bit inputs (in IEEE double precision format)

Most floating point operations merge the result with the *ra* operand to prepare the destination. Since a full 64-bit result is generated with the vector and double formats, the *ra* operand is not needed and may be specified as zero to reduce dependencies.

Exceptions to this encoding are listed where appropriate.

Unless otherwise noted, all operations update the internal floating point status register (FPSR, equivalent to the MXCSR register in x86 code) by ORing in any exceptions that occur. If the uop is encoded to generate an actual exception on excepting conditions, the `FLAG_INV` flag is attached to the output to cause an exception at commit time.

No condition code flags are generated by floating point uops unless otherwise noted.

`addf subf mulf divf minf maxf`

## Floating Point Arithmetic

| Mnemonic | Syntax | Operation |
|---|---|---|
| `addf` | `rd = ra,rb` | rd = ra $\leftarrow$ ra + rb |
| `subf` | `rd = ra,rb` | rd = ra $\leftarrow$ ra - rb |
| `mulf` | `rd = ra,rb` | rd = ra $\leftarrow$ ra $\times$ rb |
| `divf` | `rd = ra,rb` | rd = ra $\leftarrow$ ra / rb |
| `minf` | `rd = ra,rb` | rd = ra $\leftarrow$ (ra < rb) ? ra : rb |
| `maxf` | `rd = ra,rb` | rd = ra $\leftarrow$ (ra >= rb) ? ra : rb |

**Notes:**

- These uops do arithmetic on floating point numbers in various formats as specified in the *Floating Point Format and Merging* page.

`maddf msubf`

## Fused Multiply Add and Subtract

| **Mnemonic** | **Syntax** | **Operation** |
|---|---|---|
| `maddf` | `rd = ra,rb,rc` | rd = ra $\leftarrow$ (ra $\times$ rb) + rc |
| `msubf` | `rd = ra,rb,rc` | rd = ra $\leftarrow$ (ra $\times$ rb) - rc |

**Notes:**

- The `maddf` and `msubf` uops perform fused multiply and accumulate operations on three operands.
- The full internal precision is preserved between the multiply and add operations; rounding only occurs at the end.
- These uops are primarily used by microcode to calculate floating point division, square root and reciprocal.

## `sqrtf rcpf rsqrtf`
## Square Root, Reciprocal and Reciprocal Square Root

| Mnemonic | Syntax | Operation |
|---|---|---|
| `sqrtf` | `rd = ra,rb` | rd = ra $\leftarrow$ sqrt(rb) |
| `rcpf` | `rd = ra,rb` | rd = ra $\leftarrow$ 1 / rb |
| `rsqrtf` | `rd = ra,rb` | rd = ra $\leftarrow$ 1 / sqrt(rb) |

**Notes:**

- These uops perform the specified unary operation on rb and merge the result into ra (for a single precision scalar mode only)
- The `rcpf` and `rsqrtf` uops are approximates - they do not provide the full precision results. These approximations are in accordance with the standard x86 SSE/SSE2 semantics.

`cmpf`

## Compare Floating Point

| Mnemonic | Syntax | Operation |
|---|---|---|
| `cmpf.`*`type`* | `rd = ra,rb` | rd = ra $\leftarrow$ CompareFP(ra, rb, type) ? -1 : 0 |

**Notes:**

- This uop performs the specified comparison of *ra* and *rb*. If the comparison is true, the result is set to all '1' bits; otherwise it is zero. The result is then merged into ra.
- The *cond* field in the uop encoding holds the comparison type. The set of compare types matches the x86 SSE/SSE2 CMPxx instructions.

`cmpccf`

## Compare Floating Point and Generate Condition Codes

| Mnemonic | Syntax | Operation |
|---|---|---|
| `cmpccf.`*`type`* | `rd = ra,rb` | rd.flags = CompareFPFlags(ra, rb) |

**Notes:**

- This uop performs all comparisons of *ra* and *rb* and produces x86 condition code flags (ZF, PF, CF) to represent the result.
- The semantics of the generated condition code flags exactly matches the x86 SSE/SSE2 instructions `COMISS`/`COMISD`/`UCOMISS`/`UCOMISD`.
- Unlike most encodings, the *size* field holds the comparison type of the two values as follows:
  - `00: cmpccfp`: single precision ordered compare (same semantics as x86 SSE `COMISS`)
  - `01: cmpccfp.u`: single precision unordered compare (same semantics as x86 SSE `UCOMISS`)
  - `10: cmpccfd`: double precision ordered compare (same semantics as x86 SSE2 `COMISD`)
  - `11: cmpccfd.u`: double precision ordered compare (same semantics as x86 SSE2 `UCOMISD`)

`cvtf.i2s.ins cvtf.i2s.p cvtf.i2d.lo cvtf.i2d.hi`

## Convert 32-bit Integer to Floating Point

| **Mnemonic** | **Syntax** | **Operation** | **Used By** |
|---|---|---|---|
| `cvtf.i2s.ins` | `rd = ra,rb` | rd = ra ← Int32ToFloat(rb) | `CVTSI2SS` |
| `cvtf.i2s.p` | `rd = zero,rb` | rd[31:0] = Int32ToFloat(rb[31:0])<br>rd[63:32] = Int32ToFloat(rb[63:32]) | `CVTPI2PS` |
| `cvtf.i2d.lo` | `rd = zero,rb` | rd = Int32ToDouble(rb[31:0]) | `CVTSI2SD`<br>`CVTPI2PD` |
| `cvtf.i2d.hi` | `rd = zero,rb` | rd = Int32ToDouble(rb[63:32]) | `CVTPI2PD` |

**Notes:**

- These uops convert 32-bit integers to single or double precision floating point
- The semantics of these instructions are identical to the semantics of the x86 SSE/SSE2 instructions shown in the table
- The uop *size* field is not used by these uops

`cvtf.q2s.ins cvtf.q2d`

## Convert 64-bit Integer to Floating Point

| **Mnemonic** | **Syntax** | **Operation** | **Used By** |
|---|---|---|---|
| `cvtf.q2s.ins` | `rd = ra,rb` | rd = ra ← Int64ToFloat(rb) | `CVTSI2SS` (x86-64) |
| `cvtf.q2d` | `rd = ra` | rd = Int64ToDouble(ra) | `CVTPI2PS` (x86-64) |

**Notes:**

- These uops convert 64-bit integers to single or double precision floating point
- The semantics of these instructions are identical to the semantics of the x86 SSE/SSE2 instructions shown in the table
- The uop *size* field is not used by these uops

`cvtf.s2i cvt.s2q cvtf.s2i.p`

## Convert Single Precision Floating Point to Integer

| **Mnemonic** | **Syntax** | **Operation** | **Used By** |
|---|---|---|---|
| `cvtf.s2i` | `rd = ra` | rd = FloatToInt32(ra[31:0]) | `CVTSS2SI` |
| `cvtf.s2i.p` | `rd = ra` | rd[31:0] = FloatToInt32(ra[31:0]) | `CVTPS2PI` |
| | | rd[63:32] = FloatToInt32(ra[63:32]) | `CVTPS2DQ` |
| `cvtf.s2q` | `rd = ra` | rd = FloatToInt64(ra) | `CVTSS2SI` (x86-64) |

**Notes:**

- These uops convert single precision floating point values to 32-bit or 64-bit integers
- The semantics of these instructions are identical to the semantics of the x86 SSE/SSE2 instructions shown in the table
- Unlike most encodings, the *size* field holds the rounding type of the result as follows:
  - `x0:` normal IEEE rounding (as determined by FPSR)
  - `x1:` truncate to zero

`cvtf.d2i cvtf.d2q cvtf.d2i.p`

## Convert Double Precision Floating Point to Integer

| Mnemonic | Syntax | Operation | Used By |
|---|---|---|---|
| `cvtf.d2i` | `rd = ra` | rd = DoubleToInt32(ra) | `CVTSD2SI` |
| `cvtf.d2i.p` | `rd = ra,rb` | rd[63:32] = DoubleToInt32(ra) | `CVTPD2PI` |
| | | rd[31:0] = DoubleToInt32(rb) | `CVTPD2DQ` |
| `cvtf.d2q` | `rd = ra` | rd = DoubleToInt64(ra) | `CVTSD2SI` (x86-64) |

**Notes:**

- These uops convert double precision floating point values to 32-bit or 64-bit integers
- The semantics of these instructions are identical to the semantics of the x86 SSE/SSE2 instructions shown in the table
- Unlike most encodings, the *size* field holds the rounding type of the result as follows:
  - `x0`: normal IEEE rounding (as determined by FPSR)
  - `x1`: truncate to zero

`cvtf.d2s.ins cvtf.d2s.p cvtf.s2d.lo cvtf.s2d.hi`

## Convert Between Double Precision and Single Precision Floating Point

| Mnemonic | Syntax | Operation | Used By |
|---|---|---|---|
| `cvtf.d2s.ins` | `rd = ra,rb` | rd = ra ← DoubleToFloat(rb) | `CVTSD2SS` |
| `cvtf.d2s.p` | `rd = ra,rb` | rd[63:32] = DoubleToFloat(ra) rd[31:0] = DoubleToFloat(rb) | `CVTPD2PS` |
| `cvtf.s2d.lo` | `rd = zero,rb` | rd = FloatToDouble(rb[31:0]) | `CVTSS2SD` `CVTPS2PD` |
| `cvtf.s2d.hi` | `rd = zero,rb` | rd = FloatToDouble(rb[63:32]) | `CVTPS2PD` |

**Notes:**

- These uops convert single precision floating point values to double precision floating point values
- The semantics of these instructions are identical to the semantics of the x86 SSE/SSE2 instructions shown in the table
- The uop *size* field is not used by these uops

# Chapter 28

# Performance Counters

PTLsim maintains hundreds of performance and statistical counters and data points as it simulates user code. In Section 8, the basic mechanisms and data structures through which PTLsim collects these data were disclosed, and a guide to extending the existing set of collection points was presented.

This section is a reference listing of all the current performance counters present in PTLsim by default. The sections below are arranged in a hierarchical tree format, just as the data are represented in PTLsim's data store. The types of data collected closely match the performance counters available on modern Intel and AMD x86 processors, as described in their respective reference manuals.

## 28.1 General

As described in Section 8, PTLsim maintains a hierarchical tree of statistical data, defined in `stats.h`. The data store contains a potentially large number of snapshots of this tree, numbered starting at 0. The final snapshot, taken just before simulation completes, is labeled as "final". Each snapshot branch contains all of the data structures described in the next few sections. Snapshots are enabled with the `-snapshot-cycles` configuration option (Section 10.3); if they are disabled, only the "0" and "final" snapshots are provided.

## 28.2 Summary

The `summary` toplevel branch summarizes information about the simulation run across all cores:

`summary:` general information

- `cycles:` total number of simulated cycles completed
- `insns:` total number of complete x86 instructions committed

- `uops`: total number of uops committed
- `basic_blocks`: total number of basic blocks executed

`snapshot_uuid`: the universally unique ID (UUID) of this snapshot. This number starts from 0 and increases to infinity.

`snapshot_name`: name of this snapshot, if any. Named snapshots can be taken by the `ptlcall_snapshot()` call within the virtual machine, or by the `-snapshot-now` *name* command.

## 28.3 Simulator

The `simulator` toplevel branch represents information about PTLsim itself:

`version`: PTLsim version information

- `build_timestamp`: the date and time PTLsim (specifically, `ptlsim.o`) was last built
- `svn_revision`: Subversion revision number for this PTLsim version
- `svn_timestamp`: Date and time of Subversion commit for this version
- `build_hostname`: machine name on which PTLsim was compiled
- `build_compiler`: gcc compiler version used to build PTLsim

`run`: runtime environment information

- `timestamp`: time (in POSIX seconds-since-epoch format) this instance of PTLsim was started
- `hostname`: machine name on which PTLsim is running
- `kernel_version`: Linux kernel version PTLsim is running under. For PTLsim/X, this is the domain 0 kernel version
- `hypervisor_version`: PTLsim/X Xen hypervisor version
- `executable`: the executable file being run under simulation (userspace PTLsim only)
- `args`: the arguments to the executable file (userspace PTLsim only)
- `native_cpuid`: CPUID (brand/model/revision) of the host machine running PTLsim
- `native_hz`: core frequency (cycles per second) of the host machine

`config`: the configuration options last passed to PTLsim for this run

`performance`: PTLsim internal performance data

- rate: operations per wall-clock second (i.e. in outside world, not inside the virtual machine), averaged over entire run. These are the status lines PTLsim prints on the console and in the log file as it runs.
  - cycles_per_second: simulated cycles completed per second
  - issues_per_second: uops issued per second
  - user_commits_per_second: x86 instructions committed per second

## 28.4 Decoder

The decoder toplevel branch represents the x86-to-uop decoder, basic block cache, code page cache and other common structures:

throughput: total decoded entities

- basic_blocks: total basic blocks (uop sequence terminated by a branch) decoded
- x86_insns: total x86 instructions decoded
- uops: total uops produced from all decoded x86 instructions
- bytes: total bytes in all decoded x86 instructions

bb_decode_type: predominant decoder type used for each basic block

- all_insns_fast: number of basic blocks all instructions in the basic block were in the simple regular subset of x86 and could be decoded entirely by the fast decoder (decode-fast.cpp)
- some_insns_complex: number of basic blocks in which one or more instructions required complex decoding

page_crossings: alignment of instructions within page

- within_page: number of basic blocks in which all bytes in the basic block fell within a single page
- crosses_page: number of basic blocks in which some bytes crossed a page boundary (i.e. required two MFN invalidate locators)

bbcache: basic block cache accesses

- count: basic blocks currently in the cache (i.e. at the time the stats snapshot was made)
- inserts: total insert operations

- `invalidates`: invalidation operations by type
  - `smc`: self modifying code required page to be invalidated
  - `dma`: DMA into page with existing translations required page to be invalidated
  - `spurious`: `exec_page_fault` assist determined the page has now been made executable
  - `reclaim`: garbage collector discarded unused LRU basic blocks
  - `dirty`: page was already dirty when new translation was to be made
  - `empty`: page was empty (has no basic blocks)

`pagecache`: physical code page cache

- `count`: physical pages currently in the cache (i.e. at the time the stats snapshot was made)
- `inserts`: total physical page insert operations
- `invalidates`: invalidation operations by type
  - `smc`: self modifying code required page to be invalidated
  - `dma`: DMA into page with existing translations required page to be invalidated
  - `spurious`: `exec_page_fault` assist determined the page has now been made executable
  - `reclaim`: garbage collector discarded unused LRU basic blocks
  - `dirty`: page was already dirty when new translation was to be made
  - `empty`: page was empty (has no basic blocks)

`reclaim_rounds`: number of times the memory manager attempted to reclaim unused basic blocks (possibly with several attempts until enough memory was available)

## 28.5 Out of Order Core

The out of order core is represented by the `ooocore` toplevel branch of the statistics data store tree:

`cycles`: total number of processor cycles simulated

`fetch`: fetch stage statistics

- `stop`: totals up the reasons why fetching finally stopped in each cycle
  - `stalled`: fetch unit was already stalled in the previous cycle
  - `icache_miss`: an instruction cache miss prevented further fetches
  - `fetchq_full`: the uop fetch queue is full

  - `bogus_rip`: speculative execution redirected the fetch unit to an inaccessible (or non-executable) page. The fetch unit remains stalled in this state until the mis-speculation is resolved.
  - `microcode_assist`: microcode assist must wait for pipeline to empty
  - `branch_taken`: taken branches to non-sequential addresses always stop fetching
  - `full_width`: the maximum fetch width was utilized without encountering any of the events above

- `opclass`: histogram of how many uops of various operation classes passed through the fetch unit. The operation classes are defined in `ptlhwdef.h` and assigned to various opcodes in `ptlhwdef.cpp`.
- `width`: histogram of the fetch width actually used on each cycle
- `blocks`: blocks of x86 instructions fetched (typically the processor can read at most e.g. 16 bytes out of a 64 byte instruction cache line per cycle)
- `uops`: total number of uops fetched
- `user_insns`: total number of x86 instructions fetched

`frontend`: frontend pipeline (decode, allocate, rename) statistics

- `status`: totals up the reasons why frontend processing finally stopped in each cycle
  - `complete`: all uops were successfully allocated and renamed
  - `fetchq_empty`: no more uops were available for allocation
  - `rob_full`: reorder buffer (ROB) was full
  - `physregs_full`: physical register file was full even though an ROB slot was free
  - `ldq_full`: load queue was full (too many loads in the pipeline) even though physical registers were available
  - `stq_full`: store queue was full (too many stores in the pipeline)
- `width`: histogram of the frontend width actually used on each cycle
- `renamed`: summarizes the type of renaming that occurred for each uop (of the destination, not the operands)
  - `none`: uop did not rename its destination (primarily for stores and branches)
  - `reg`: uop renamed destination architectural register
  - `flags`: uop renamed one or more of the ZAPS, CF, OF flag sets but had no destination architectural register

    - `reg_and_flags`: uop renamed one or more of the ZAPS, CF, OF flag sets as well as a destination architectural register

- `alloc`: summarizes the type of resource allocation that occurred for each uop (in addition to its ROB slot):
    - `reg`: uop was allocated a physical register
    - `ldreg`: uop was a load and was allocated both a physical register and a load queue entry
    - `sfr`: uop was a store and was allocated a store forwarding register (SFR), a.k.a. store queue entry
    - `br`: uop was a branch and was allocated branch-related resources (possibly including a destination physical register)

`dispatch`: dispatch unit statistics

- `source`: totals up where each operand to each uop currently resided at the time the uop was dispatched. These statistics are broken out by cluster.
    - `waiting`: how many operands were waiting (i.e. not yet ready)
    - `bypass`: how many operands would come from the bypass network if the uop were immediately issued
    - `physreg`: how many operands were already written back to physical registers
    - `archreg`: how many operands would be obtained from architectural registers
- `cluster`: tracks the number of uops issued to each cluster (or issue queue) in the processor. This list will vary depending on the processor configuration. The value *none* means that no cluster could accept the uop because all issue queues were full.
- `redispatch`: statistics on the redispatch speculation recovery rmechanism (Section 20.3.2)
    - `trigger_uops` measures how many uops triggered redispatching because of a mis-speculation. This number does not count towards the statistics below.
    - `deadlock_flushes` measures how many times the pipeline must be flushed to resolve a deadlock.
    - `dependent_uops` is a histogram of how many uops depended on each trigger uop, not including the trigger uop itself.

`issue`: issue statistics

- `result`: histogram of the final disposition of issuing each uop
    - `no-fu`: no functional unit was available within the uop's assigned cluster even though it was already issued

  - `replay`: uop attempted to execute but could not complete, so it must remain in the issue queue to be replayed. This event generally occurs when a load or store detects a previously unknown forwarding dependency on a prior store, when the data to actually store is not yet available, or when insufficient resources are available to complete the memory operation. Details are given in Sections 21 and 22.2.
  - `misspeculation`: uop mis-speculated and now all uops after and including the issued uop must be annulled. This generally occurs with loads (Section 21) and stores (Section 22.2.1) when unaligned accesses or load-store aliasing occurs. This event is handled in accordance with Section 20.3.2.
  - `refetch`: uop and all subsequent uops must be re-fetched to be decoded differently. For example, unaligned loads and stores take this path so they can be cracked into two parts after being refetched.
  - `branch_mispredict`: uop was a branch and mispredicted, such that all uops after (but not including) the branch uop must be annulled. See Section 20 for details.
  - `exception`: uop caused an exception (though this may not be a user visible error due to speculative execution)
  - `complete`: uop completed successfully. Note that this does *not* mean the result is immediately ready; for loads it simply means the request was issued to the cache.

- `source`: totals up where each operand to each uop was read from as it was issued
  - `bypass`: how many operands came directly off the bypass network
  - `physreg`: how many operands were read from physical registers
  - `archreg`: how many operands were read from committed architectural registers
- `width`: histogram of the issue width actually used on each cycle in each cluster. This object is further broken down by cluster, since various clusters have different issue width and policies.
- `opclass`: histogram of how many uops of various operation classes were issued. The operation classes are defined in `ptlhwdef.h` and assigned to various opcodes in `ptlhwdef.cpp`.

`writeback`: writeback stage statistics

- `total_writebacks`: total number of results written back to the physical register file
- `transient`: transient versus persistent values
  - `transient`: the result technically does not have to be written back to the physical register file at all, since all consumers sourced the value off the bypass network and the result is no longer available since the destination architectural register pointing to it has since been renamed.
  - `persistent`: all values which do not meet the conditions above and hence must still be written back

- `width:` histogram of the writeback width actually used on each cycle in each cluster. This object is further broken down by cluster, since various clusters have different issue width and policies.

`commit:` commit unit statistics

- `uops:` total number of uops committed
- `insns:` total number of complete x86 instructions committed
- `result:` histogram of the final disposition of attempting to commit each uop
  - `none:` one or more uops comprising the x86 instruction at the head of the ROB were not yet ready to commit, so commitment is terminated for that cycle
  - `ok:` result was successfully committed
  - `exception:` result caused a genuine user visible exception. In userspace PTLsim, this will terminate the simulation. In full system PTLsim/X, this is a normal and frequent event. Floating point state dirty faults are counted under this category.
  - `skipblock:` This occurs in rare cases when the processor must skip over the currently executing instruction (such as in pathological cases of the `rep` x86 instructions).
  - `barrier:` the processor encountered a barrier instruction, such as a system call, assist or pipeline flush. The frontend has already been stopped and fetching has been redirected to the code to handle the barrier; this condition simply commits the barrier instruction itself.
  - `smc:` self modifying code: the instruction attempting to commit has been modified since it was last decoded (see Section 6.4)
  - `stop:` special case for when the simulation is to be stopped after committing a certain number of x86 instructions (e.g. via the `-stopinsns` option in Section 10.3).
- `setflags:` how many uops updated the condition code flags as they committed
  - `yes:` how many uops updated at least one of the ZAPS, CF, OF flag sets (the `REG_flags` internal architectural register)
  - `no:` how many uops did not update any flags
- `freereg:` how many uops were able to free the old physical register mapped to their architectural destination register at commit time
  - `pending:` old physical register was still referenced within the pipeline or by one or more rename table entries
  - `free:` old physical register could be immediately freed
- `free_regs_recycled:` how many physical registers were recycled (garbage collected) later than normal because of one of the conditions above

- `width`: histogram of the issue width actually used on each cycle in each cluster. This object is further broken down by cluster, since various clusters have different issue width and policies.
- `opclass`: histogram of how many uops of various operation classes were issued. The operation classes are defined in `ptlhwdef.h` and assigned to various opcodes in `ptlhwdef.cpp`.

`branchpred`: branch predictor statistics

- `predictions`: total number of branch predictions of any type
- `updates`: total number of branch predictor updates of any type
- `cond`: conditional branch (`br.cc` uop) prediction outcomes, broken down into correct predictions and mispredictions
- `indir`: indirect branch (`jmp` uop) prediction outcomes, broken down into correct predictions and mispredictions
- `return`: return (`jmp` uop with `BRANCH_HINT_RET` flag) prediction outcomes, broken down into correct predictions and mispredictions
- `summary`: summary of all prediction outcomes of the three types above, broken down into correct predictions and mispredictions
- `ras`: return address stack (RAS) operations
  - `push`: RAS pushes on calls
  - `push_overflows`: RAS pushes on calls in which the RAS overflowed
  - `pop`: RAS pops on returns
  - `pop_underflows`: RAS pops on returns in which the RAS was empty
  - `annuls`: annulment operations in which speculative updates to the RAS were rolled back

## 28.6 Cache Subsystem

The cache subsystem is listed under the `ooocore/dcache` branch.

`load`: load unit statistics

- `issue`: histogram of the final disposition of issuing each load uop
  - `complete`: cache hit
  - `miss`: L1 cache miss, and possibly lower levels as well (Sections 21.4 and 25.2)

  - `exception`: load generated an exception (typically a page fault), although the exception may still be speculative (Section 21)
  - `ordering`: load was misordered with respect to stores (Section 22.2.1)
  - `unaligned`: load was unaligned and will need to be re-executed as a pair of low and high loads (Sections 5.6 and 21)
  - `replay`: histogram of events in which a load needed to be replayed (Section 21)
    * `sfr-addr-and-data-not-ready`: load was predicted to forward data from a prior store (Section 22.2.1), but neither the address nor the data of that store has resolved yet
    * `sfr-addr-not-ready`: load was predicted to forward data from a prior store, but the address of that store has not resolved yet
    * `sfr-data-not-ready`: load address matched a prior store in the store queue, but the data that store should write has not resolved yet
    * `missbuf-full`: load missed the cache but the miss buffer and/or LFRQ (Section 25.2) was full at the time

- `hit`: histogram of the cache hierarchy level each load finally hit
  - `L1`: L1 cache hit
  - `L2`: L1 cache miss, L2 cache hit
  - `L3`: L1 and L2 cache miss, L3 cache hit
  - `mem`: all caches missed; value read from main memory

- `forward`: histogram of which sources were used to fill each load
  - `cache`: how many loads obtained all their data from the cache
  - `sfr`: how many loads obtained all their data from a prior store in the pipeline (i.e. load completely overlapped that store)
  - `sfr-and-cache`: how many loads obtained their data from a combination of the cache and a prior store

- `dependency`: histogram of how loads related to previous stores
  - `independent`: load was independent of any store currently in the pipeline
  - `predicted-alias-unresolved`: load was stalled because the load store alias predictor (LSAP) predicted that an earlier store would overlap the load's address address even though that earlier store's address was unresolved (Section 22.2.1)
  - `stq-address-match`: load depended on an earlier store still found in the store queue

- `type`: histogram of the type of each load uop
  - `aligned`: normal aligned loads

  - `unaligned`: special unaligned load uops `ld.lo` or `ld.hi` (Section 5.6)
  - `internal`: loads from PTLsim space by microcode

- `size`: histogram of the size in bytes of each load uop
- `transfer-L2-to-L1`: histogram of the types of L2 to L1 line transfers that occurred (Section 25)
  - `full-L2-to-L1`: all bytes in cache line were transferred from L2 to L1 cache
  - `partial-L2-to-L1`: some bytes in the L1 line were already valid (because of stores to those bytes), but the remaining bytes still need to be fetched
  - `L2-to-L1I`: all bytes in the L2 line were transferred into the L1 instruction cache
- `dtlb`: data cache translation lookaside buffer hit versus miss rate (Section 25.4)

`fetch`: instruction fetch unit statistics (Section 17.1)

- `hit`: histogram of the cache hierarchy level each fetch finally hit
  - `L1`: L1 cache hit
  - `L2`: L1 cache miss, L2 cache hit
  - `L3`: L1 and L2 cache miss, L3 cache hit
  - `mem`: all caches missed; value read from main memory
- `itlb`: instruction cache translation lookaside buffer hit versus miss rate (Section 25.4)

`prefetches`: prefetch engine statistics

- `in-L1`: requested data already in L1 cache
- `in-L2`: requested data already in L2 cache (and possibly also in L1 cache)
- `required`: prefetch was actually required (data was not cached or was in L3 or lower levels)

`missbuf`: miss buffer performance (Sections 25.2 and 25.3)

- `inserts`: total number of lines inserted into the miss buffer
- `delivers`: total number of lines delivered to various cache hierarchy levels from the miss buffer
  - `mem-to-L3`: deliver line from main memory to the L3 cache
  - `L3-to-L2`: deliver line to the L3 cache to the L2 cache

  - `L2-to-L1D`: deliver line from the L2 cache to the L1 data cache
  - `L2-to-L1I`: deliver line from the L2 cache to the L1 instruction cache

`lfrq`: load fill request queue (LFRQ) performance (Sections 25.2 and 25.3)

- `inserts`: total number of loads inserted into the LFRQ
- `wakeups`: total number of loads awakened from the LFRQ
- `annuls`: total number of loads annulled in the LFRQ (after they were annulled in the processor core)
- `resets`: total number of LFRQ resets (all entries cleared)
- `total-latency`: total latency in cycles of all loads passing through the LFRQ
- `average-miss-latency`: average load latency, weighted by cache level hit and latency to that level
- `width`: histogram of how many loads were awakened per cycle by the LFRQ

`store`: store unit statistics

- `issue`: histogram of the final disposition of issuing each store uop
  - `complete`: store completed without problems
  - `exception`: store generated an exception (typically a page fault), although the exception may still be speculative (Section 22.1)
  - `ordering`: store detected that a later load in program order aliased the store but was issued earlier than the store (Section 22.2.1)
  - `unaligned`: store was unaligned and will need to be re-executed as a pair of low and high stores (Sections 5.6)
  - `replay`: histogram of events in which a store needed to be replayed (Sections 22.2 and 22.1)
    * `wait-sfraddr-sfrdata`: neither the address nor the data of a prior store this store inherits some of its data from was ready
    * `wait-sfraddr`: the data of a prior store was ready but its address was still unavailable
    * `wait-sfrdata`: the address of a prior store was ready but its data was still unavailable
    * `wait-storedata-sfraddr-sfrdata`: the actual data value to store was not ready (Section 22.2), in addition to having neither the data nor the address of a prior store (Section 22.1)

      * `wait-storedata-sfraddr`: the actual data value to store was not ready (Section 22.2), in addition to not having the address of the prior store (Section 22.1)
      * `wait-storedata-sfrdata`: the actual data value to store was not ready (Section 22.2), in addition to not having the data from the prior store (Section 22.1)

- `forward`: histogram of which sources were used to construct the merged store buffer:
  - `zero`: no prior store overlapping the current store was found in the pipeline
  - `sfr`: data from a prior store in the pipeline was merged with the value to be stored to form the final store buffer
- `type`: histogram of the type of each store uop
  - `aligned`: normal aligned store
  - `unaligned`: special unaligned store uops `st.lo` or `st.hi` (Section 5.6)
  - `internal`: stores to PTLsim space by microcode
- `size`: histogram of the size in bytes of each store uop
- `commit`: histogram of how stores are committed
  - `direct`: store committed directly to the data cache in the commit stage (Section 24)
- `commits`: total number of committed uops
- `usercommits`: total number of committed x86 instructions
- `issues`: total number of uops issued. This includes uops issued more than once by through replay (Section 19.3).
- `ipc`: Instructions Per Cycle (IPC) statistics
  - `commit-in-uops`: average number of uops committed per cycle
  - `issue-in-uops`: average number of uops issued per cycle
  - `commit-in-user-insns`: average number of x86 instructions committed per cycle

    ***NOTE:*** Because one x86 instruction may be broken up into numerous uops, it is ***never*** appropriate to compare IPC figures for committed x86 instructions per clock with IPC values from a RISC machine. Furthermore, different x86 implementations use varying numbers of uops per x86 instruction as a matter of encoding, so even comparing the uop based IPC between x86 implementations or RISC-like machines is inaccurate. Users are strongly advised to use relative performance measures instead (e.g. total cycles taken to complete a given benchmark).

`simulator`: describes the performance of PTLsim itself. Useful for tuning the simulator.

- `total_time:` total time in seconds *(not simulated cycles!)* spent in various parts of the simulator. Please refer to the source code (in `ooocore.cpp`) for the range of code each time value corresponds to.

- `cputime:` PTLsim simulator performance

  - `fetch:` seconds spent in fetch stage
  - `decode:` seconds spent decoding instructions (in decoder subsystem)
  - `rename:` seconds spent in allocate and rename stage
  - `frontend:` seconds spent in frontend stages
  - `dispatch:` seconds spent in dispatch stage
  - `issue:` seconds spent in ALU issue stage, not including loads and stores
  - `issueload:` seconds spent issuing loads
  - `issuestore:` seconds spent issuing stores
  - `complete:` seconds spent in completion stage
  - `transfer:` seconds spent in transfer stage
  - `writeback:` seconds spent in writeback stage
  - `commit:` seconds spent in commit stage

## 28.7 External Events

- `assists:` histogram of microcode assists invoked from any core

- **traps:** histogram of x86 interrupt vectors (traps) invoked from any core (PTLsim/X only)

# Bibliography

[1] M. Yourst. *PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator.* ISPASS 2007, April 2007.

[2] *XenSource Community Web Site.*

[3] *Xen page at Cambridge.*

[4] *Xen and the Art of Virtualization.* I. Pratt et al. Ottowa Linux Symposium 2004.

[5] *Xen page at Cambridge.*

[6] *Xen and the Art of Virtualization.* I. Pratt et al. Ottowa Linux Symposium 2004.

[7] *Xen 3.0 Virtualization.* I. Pratt et al. FOSDEM 2006.

[8] *Introduction to Xen 3.0.*

[9] *Xen Performance Study.*

[10] *QEMU Internals.* F. Bellard. Tech Report, 2006.

[11] *Bochs IA-32 Emulator Project.*

[12] *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor.* J. Sugerman et al.

[13] *Simics.*

[14] *SimNow: Fast Platform Simulation Purely in Software.* R. Bedichek (AMD). Hot Chips 2004.

[15] *IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1,* Chapter 19, "Introduction to Virtual Machine Extensions".

[16] *AMD64 Architecture Programmer's Manual, Volume 2: System Programming,* Chapter 15, "Secure Virtual Machine".

[17] E. Kelly et al. *Translated memory protection apparatus for an advanced microprocessor.* U.S. Patent 6199152, filed 22 Aug 1996. Assn. Transmeta Corp.

[18] J. Banning et al. *Fine grain translation discrimination.* U.S. Patent 6363336, filed 13 Oct 1999. Assn. Transmeta Corp.

[19] J. Banning et al. *Translation consistency checking for modified target instructions by comparing to original copy.* U.S. Patent 6594821, filed 30 Mar 2000. Assn. Transmeta Corp.

[20] K. Ebcioglu et al. *Dynamic Binary Translation and Optimization*. IEEE Trans. Computers, June 2001.

[21] K. Ebcioglu, E. Altman. *DAISY: Dynamic Compilation for 100% Architectural Compatibility.* IBM Research Report RC 20538, 5 Aug 1996.

[22] E. Altman, K. Ebcioglu. *DAISY Dynamic Binary Translation Software.* Software Manual for DAISY Open Source Release, 2000.