ZESTO: RICH AND HEAVY MICROARCHITECTURE SIMULATION

GABRIEL H. LOH
Georgia Institute of Technology
College of Computing
School of Computer Science
266 Ferst Drive, Atlanta, GA 30332-0765
LOH@CC.GATECH.EDU

Abstract

This document describes the Zesto cycle-level x86 simulator, built on top of the SimpleScalar-x86 pre-release. This report describes the different microarchitectural modules, techniques for supporting various "x86-isms," how to use and extend the simulator, and the limited multi-core support. This document is *not* intended to be an exhaustive and comprehensive reference for the simulator, but it aims to provide a high-level understanding and roadmap for the simulator to help potential simulator hackers more quickly get up and running. For a shorter introduction, you may wish to read our ISPASS 2009 paper [5]. Also check the newsgroup (linked off of the zesto.cc.gatech.edu website) for other discussions, frequently asked questions, etc.

Last Modified: April 21, 2009

Contents

1 Introduction

1.1 Why Yet Another Simulator?

This simulator project was started for two main reasons. First, at the time of the start of this project, there were no cycle-level **x86** processor simulators publicly available for academic use. Second, of the many non-x86 simulators available out there, most provided grossly out-dated microarchitectural models (e.g., SimpleScalar 3.0's sim-outorder) and even the more up to date versions (e.g., MASE [4]) did not provide a sufficient level of detail for low-level microarchitecture research.

1.2 Why Not Use Other Simulators?

We are not proposing Zesto as a replacement for all other simulators. Zesto's strength is in low-level cycle-by-cycle modeling of a modern, Intel-styled processor pipeline. For researchers studying proposals that involve "higher level" processing, such as multi-threading, transactional memory, hardware support for security or debugging, etc., a faster simulator with better system support/emulation would be more appropriate.

PTLsim from the State University of New York at Binghamton is a very relevant point of comparison [9]. PTLsim was not available at the time of Zesto's inception, and if it was, it may have served as the foundation for Zesto. Nevertheless, Zesto's microarchitectural model is much more detailed than that used in PTLsim. Porting or copying Zesto's microarchitectural model to PTLsim in the future may be a reasonable endeavor, especially since PTLsim provides full support for x86-64, SSE, etc. Zesto also provides simple multi-core simulation "out of the box" whereas PTLsim requires installation of the Xen hypervisor/virtual machine, installed as root, which can lead to some logistical issues for setup, permissions (esp. for students), batch job scheduling, etc. The tradeoff is that Zesto's multi-core simulation only supports *multi-programmed* workloads, and **not** true multi-threaded applications.

1.3 Why "Zesto"?

Zesto is the name of an ice cream chain in Atlanta, GA, USA. They serve great soft-serve ice cream which is both rich and heavy. We decided to name this simulator after the ice cream as it is also rich (featureful and very detailed) and heavy (slow). There are still Zesto's scattered throughout a handful of states from coast-to-coast. If you happen to live near one, we recommend that you try out their ice cream sometime! See:

- http://en.wikipedia.org/wiki/Zesto_Drive-In
- http://www.zestoatlanta.com

2 Overall Structure

This section provides a description of the overall organization of the simulator. In the following sections, we will provide slightly more in-depth descriptions of specific parts of the simulator and the simulated pipeline, but the best reference is still the actual simulator source code itself. There is no real good substitution to rolling up your sleeves and getting your hands dirty in the code.

2.1 High-Level Description

The overall organization of the Zesto performance simulator is similar to that of MASE [4]. External to the code that tracks all of the microarchitectural state and other cycle-to-cycle details, a functional oracle simulator provides the dynamic stream of instructions. The oracle executes all instructions before the performance simulator ever sees anything, which enables a variety of true "oracle" performance studies, as shown in Figure 1(a). Contrast this to SimpleScalar 3.0's sim-outorder simulator where an instruction is not actually executed by any code until the dispatch function, which may occur many cycles after fetch. This prevents, for example, the simulation of a perfect branch predictor. Unlike MASE and M5 [1], however, the Zesto simulator does not currently support "execute-at-execute" simulation. Zesto can be relatively easily extended to support this (the code was intentionally structured to allow this in the future), but for now it remains an execute-at-fetch simulator since it is slow enough already.

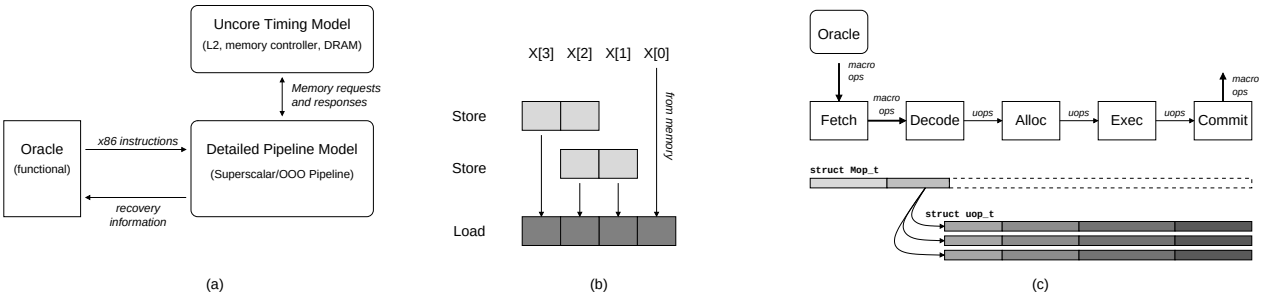

Figure 1: (a) Overall simulator organization, (b) example memory read operation requiring multiple sources, (c) high-level pipeline organization and corresponding macro-op and micro-op data structures.

The functional oracle executes instructions based on the guidance of the modeled fetch engine (to be described later). If the fetch engine directs the oracle down a mispredicted path, the oracle will attempt to execute down the wrong path. The oracle maintains a queue of all "in flight" instructions, so that when the misprediction is eventually detected by the timing simulator, the oracle can simply rewind its state. Store instructions are handled in a fashion similar to MASE's oracle, where the stores are not permitted to actually modify the simulated memory. Instead, an auxiliary speculative memory data structure tracks all stores that are in-flight in the simulation, and all loads must check for forwarded bytes from these stores before obtaining values from the simulated memory. The handling of these speculative stores can get slightly complex due to x86's many memory sizes as well as allowing unaligned and overlapping accesses. For example, the processor may contain a two-byte store to address X[1] (which includes the byte at X[2]) and a two-byte store to address X[2] (which includes X[3]). A four-byte load from address X[0] needs to read bytes from X[0] through X[3], which means the oracle must splice together the one X[1] byte from the first store (but not X[2] which is overwritten by the second store), two bytes from the second store X[2] and X[3], and one more byte X[0] from the simulated memory. This is graphically shown in Figure 1(b). Note that this has nothing to do with the timing simulator, but this careful handling of memory operations is just for the correct operation of the oracle functional simulator.

To "execute" an x86 instruction, the oracle actually first decomposes the original x86 instruction into a sequence of RISC-like micro-operations, or μops, that implement the equivalent functionality of the original

x86 *macro-operation*, or macro-op. The oracle then functionally executes each of the individual μops. In this fashion, all register values, all memory values, and all branch outcomes are known before the instruction even enters the pipeline. This provides the ability to conduct a range of oracle and limit-type studies that are not (at least easily) possible with trace-based or instrumentation-based simulation techniques.

The overall pipeline is divided into five major components or blocks, although it is important to not think of these as actual pipeline "stages" since each component may be comprised of many actual pipeline stages, queues, and other structures. Figure 1(c) shows these as the fetch, decode, allocation (alloc), execution (exec) and commit blocks. The fetch block deals exclusively with the macro-op structures, since in a real processor, μops do not even exist until after the macro-op has been decoded into μops. The decode block deals with both macro-ops and μops, with macro-ops entering at one end, and μops exiting the other. After this point up until the final commit process, the remaining hardware blocks deal with μops directly. Only the final commit process needs to be aware of the original macro-ops to ensure atomic commit of x86 instructions.

2.2 x86 Macro-Ops and Uops

The x86 instructions that a compiler generates (or that you code in assembly) vary in complexity from simple one-byte instructions (e.g., INC) to much more complex entities (e.g., string copy). These instructions (regardless of complexity) are referred to as *macro operations* or simply *macro-ops*. Throughout the simulator code, such instructions are encoded in `struct Mop_t` structures. The internal data structures for the x86 macro- and micro-ops, schematically illustrated below the pipeline components in Figure 1(c), are organized to reflect the pipeline organization.

To maintain a simple pipeline organization, most modern x86 implementations convert the user-visible macro-ops into user invisible *micro-uops* (also called μops or simply uops) in Intel speak, or rops (RISC-ops) in AMD parlance. In our simulator and throughout the rest of this document, we refer to these operations as *uops*, and in they are encoded in `struct uop_t` structures.

The oracle functional simulator actually performs the complete translation from macro-op (Mop) to uops, but this is not exposed in the timing simulator until the decode stage. That is, all stages prior to decode do not operate on `uop_t` structures (they instead use correspondingly larger-grained units of instructions

such as `Mop_t` or fetch groups). After decode, almost all of the simulator code deals with uops until commit.

Simulator statistics use both Mops and uops, depending on the context (this should always be clear either from the name of the stat or from the textual descriptions). For example, at fetch there are actually four different rate metrics presented. The first is Bytes per Cycle (BPC) which is the average number of instruction bytes fetched per cycle, which accounts for things like predicted taken branches (i.e., the bytes after the branch are not counted toward this rate). This makes use of oracle-level information to know what bytes correspond to which instructions (the pipeline cannot know this (at least without some form of decode prediction) since the bytes have not yet been decoded at this point); Instructions per Cycle (IPC) corresponds to the number of Mops fetched; Uops per Cycle (uPC) corresponds to the number of uops fetched per cycle even though the Mops have not yet been decoded into uops (e.g., a Mop that would eventually get decoded into three uops is counted as three for this statistic); and *effective* uops per Cycle (euPC) which will be discussed later in Section 12.4 regarding uop fusion.

2.3 Data Structures

Corresponding to the oracle and performance simulators, there are two primary data structures, both defined in `zesto-structs.h`. The `struct thread_t` structure corresponds to the oracle functional simulator's architectural state. As a result, `thread_t` contains relatively little state; most of the state simply corresponds to architectural state defined by the ISA. The real pipeline state is all stored in `struct core_t`, which includes sub-structures for all of the pipeline stages: fetch, decode, allocation, scheduling, execution and commit. It also includes pointers to many auxiliary structures such as branch predictors and caches. Using these pipeline-stage sub-structures forces you to explicitly identify the stage that you are referencing (e.g., `uop->decode.is_ctrl`), which should help in preventing "impossible" usages of information (e.g., code in the fetch stage should not be accessing any fields of the decode sub-structures unless you have a very good reason for it). Similarly, you cannot "accidentally" make use of oracle information with out having first explicitly referenced a member of `uop->oracle`. This was a danger of sim-outorder-style simulator organization which could lead to unrealistic/impractical results.

2.4 Pipeline Organization

Zesto supports multiple pipeline models/organizations/implementations. Each pipeline model (specified by the "`-model`" command line option) must implement each of the five basic pipeline stages: Fetch, Decode, Allocation, Execute, and Commit. The sub-class definitions for each stage are all found in their respective ZPIPE-* directories (e.g., ZPIPE-fetch). The current version of Zesto includes two models: a Detailed Pipeline Model (DPM) and a Simple(r) Timing Model (STM). The discussion of pipeline organization in the following sections refer to the DPM model. The STM model is derived from DPM, but many microarchitectural details are simplified, approximated, or simply omitted. In our test runs, the STM model runs approximately 20% faster than the DPM model in single-core mode, 35% faster in dual-core mode, but only 10% faster in quad-core mode.[1]

The DPM pipeline model used in Zesto is roughly based on our best-guess of what a modern Intel x86 pipeline would look like. This is illustrated in Figure 2, which includes the names of some of the relevant knobs. The fetch stage of the Zesto simulator operates on entire lines from the instruction cache which are placed in the byte queue. A pre-decode pipeline performs the initial decoding of the variable-length x86 instructions to individual macro-ops, which are placed into the instruction queue (IQ, a.k.a. IFQ) with one macro-op per IFQ entry. From here, the instructions proceed to the decoder pipelines. Zesto supports asymmetric decoders that can each produce different numbers of uops (e.g., 4-1-1 in the original Pentium-Pro). Instructions that require more uops than a decoder can handle must wait for a sufficiently equipped decoder to be available. In the case that none of the decoders can handle the macro-op, the uop-sequencer will provide the corresponding uop flow. After decoding, the uops are placed in a uop queue (uopQ). The allocation pipeline gets the uops from the uopQ and then performs the resource allocation (e.g., allocating an RS entry, ROB entry, and LDQ or STQ entry if needed). At the end of allocation, the uops are placed into their respective buffers (e.g., RS entries) where the scheduling function will schedule the uops for execution in dataflow order subject to execution constraints (e.g., structural hazards). As part of the scheduling and execution process, uops read their operands and other information from a payload RAM and then proceed

[1] We used a fixed processor configuration, where in particular the L2 cache size was the same regardless of core count. For the single and dual-core runs, a lot of the simulation time was spent on modeling the cores. For the quad-core runs, the L2 cache contention was so great that the majority of the simulation time was spent on the cache hierarchy, and so speeding up the core models did not do very much to help the overall simulation time.

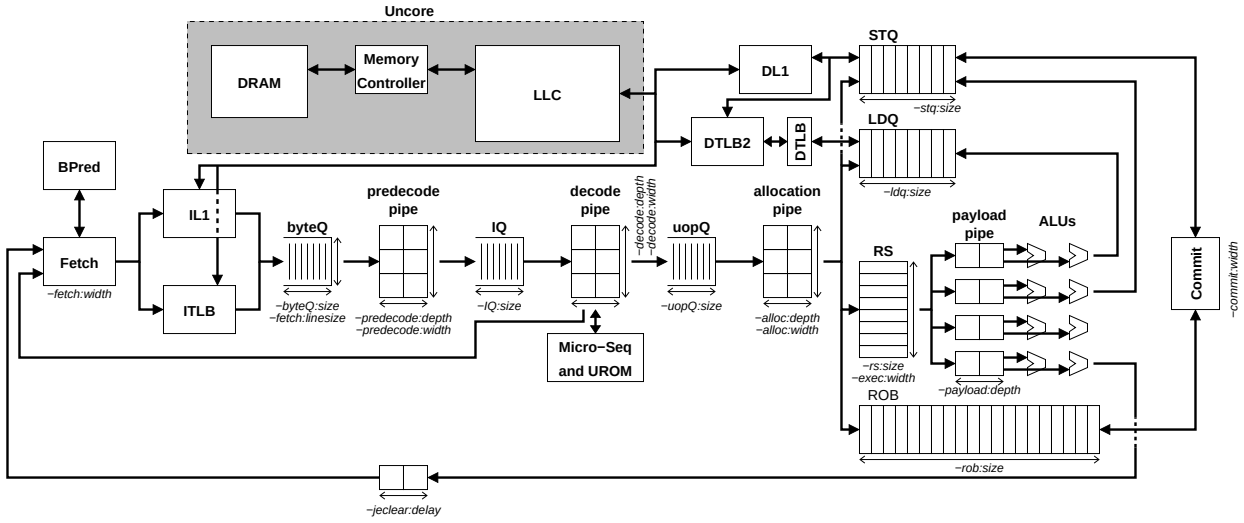

Figure 2: Overview of the Zesto (DPM) pipeline model.

to the actual execution units. Finally, the commit function retires uops from the pipeline, but does so only when all uops corresponding to a single macro-op have all completed without faults (i.e., while uops may execute out of order, it must appear to the external world that either the entire macro-op has executed or none of it has executed). More details will be provided in the following sections.

For multi-core simulations, the section labeled "Uncore" in Figure 2 is instantiated only once. All other components are replicated once per core.

2.5 Zesto Components

Several microarchitectural blocks have been written using a more componentized approach in the hopes of making it easier to add new algorithms/implementations. Examples include the branch direction predictor, the branch target predictor, prefetchers and main memory. These can all be found in the ZCOMPS-* directories. This evolved due to the original C implementation, and have been converted to proper classes using C++'s inheritance/OOP. The idea is that you should be able to simply copy one of the files (e.g., copy `bpred-2bc.cpp` to `bpred-mypred.cpp`), make a few changes to the file to reflect your new implementation of the component, and enter `make` and be done. The makefile includes calls to a script that will automatically integrate your new component without you needing to touch *any* other files. See the

corresponding sections for more information about particular types of Zesto components.

3 Oracle

The oracle provides the execution part of the "execute-at-fetch" model. The oracle interacts with the fetch stage to determine what instructions to execute, and it also interfaces with several other pipeline stages when recovery/rollback is required. Although the oracle always has perfect knowledge of branch outcomes, it will execute down the wrong path as directed by the (imperfect) branch predictions of the fetch unit.

The oracle maintains an array of all instructions/macro-ops (Mops) in-flight in the simulated processor (equivalent to the ISQ in MASE). Each entry consists of a single x86 macro instruction, which in turn may contain many uops. Prior to delivering an instruction to the fetch stage, the oracle executes each and every uop corresponding to the requested instruction. Each entry in this "MopQ" contains all of the information to undo the effects of the instruction (e.g., previous register and/or memory values). On a pipeline flush (e.g., due to a mispredicted branch or a load ordering violation), the out-of-order core notifies the oracle which instruction it should roll back to, and the oracle uses this "undo" information to recover the previous state of the pipeline. This organization allows nested mispredictions, as well as recoveries while in the shadow of other unresolved branches.

While not currently implemented, the oracle would eventually also provide the checker infrastructure for when a full execute-at-execute model gets implemented.

4 Fetch

The fetch stage (Figure 3) includes the PC generation (i.e., branch prediction) up to and including enqueuing Mops into the instruction queue (IQ). This represents all of the traditional fetch activities as well as pre-decoding to determine instruction lengths/boundaries.

Fetch uses `core->fetch.PC` to determine where to fetch from. This address is enqueued in the byteQ, and then the fetch logic simulates multiple branch predictions by simply walking through each instruction and making branch predictions until either a taken branch is encountered or the end of the current cache line is reached. The byteQ only has a limited width, which is typically less than the width of the

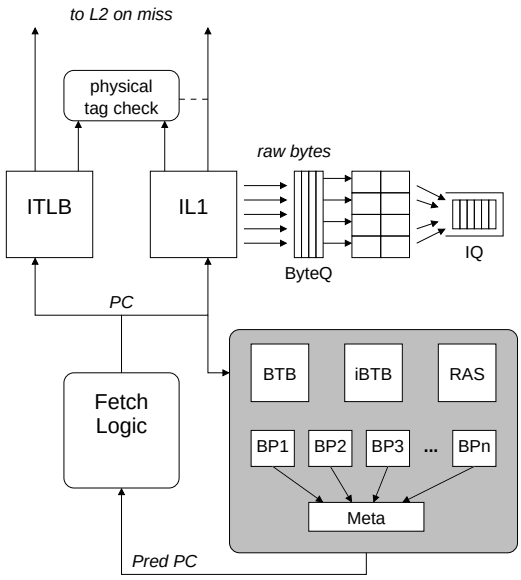

Figure 3: Block-diagram of the Zesto DPM fetch engine components.

instruction cache line size. This is the maximum number of bytes that can be fetched per cycle.

On each cycle, the fetch engine also scans the byteQ for any requests that have not yet been sent to the IL1 and ITLB. If any exist, up to one IL1 and one ITLB request can be processed per cycle (typically corresponding to the same instruction). The cache hierarchy, which includes both the IL1 and ITLB, uses a callback mechanism similar to that used in MASE [4], which is described in more detail in Section 10. When the request returns from the IL1 and ITLB, the `byteQ->when_fetched` and `byteQ->when_translated` fields get set.

The pre-decode pipeline (instruction length decoding) attemps to dequeue instructions from the byteQ. It does so in program order, and must wait until both `when_fetched` and `when_translated` fields are set. These bytes are then consumed and they make their way down the pre-decode pipeline. A single byteQ entry's worth of bytes may supply instructions for several cycles. When the pre-decode has consumed all of the instructions in the byteQ entry, the entry gets deallocated, which allows another request to start. To ensure no stalls in the fetch process, the byteQ should have as many entries as the IL1 hit latency. Note that the simulator properly handles instructions that are split across two cachelines by waiting until *both* cachelines have been translated and fetched.

Instructions propagate down the pre-decode pipeline. Note that all in-order pipelines in the Zesto simulator are "non-serpentine." A serpentine pipeline is one that allows arbitrary compression of its contents. A non-serpentine pipeline advances all instructions in lock-step fashion. That is, all instructions in a stage advance to the next stage, or none of them do. This in turn means that instructions cannot advance unless the next stage is empty, or all instructions in the next stage will also advance. This is different than what is implemented in most publicly-available simulators, but this is closer to what is done in real pipelines. Finally, as instructions leave the pre-decode pipeline, the fetch engine inserts these into the instruction queue (IQ) which serves as the buffer/interface between fetch and decode.

5 Decode

The decode logic (Figure 4) is organized as a non-serpentine pipeline, with a variable number of stages as well as width. The width corresponds to the total number of decoders, and hence the maximum number of macro-ops that can be decoded per cycle. Each decoder has a specified limit on the maximum number of uops that it can generate. Instructions are assigned to decoders in program order, so if there are three instructions A, B and C to decode, A gets assigned to decoder 0, B to decoder 1, and C to decoder 2. If the decoders have a limit of 4-1-1 (the first decoder can generate up to four uops, and the other two can only handle instructions that decompose to a single uop), and A, B and C decompose to 4, 1 and 2 uops, respectively, then A and B get decoded (generating a total of 5 uops), and C must wait until the next cycle where it will get assigned to decoder 0. If uop-fusion is enabled, then a fused uop still only counts as a single uop with respect to a decoder's output capacity. If in the previous example, instruction C decoded to two uops that were fused (e.g., load+add), then A, B and C could all be decoded in a single cycle.

Some x86 instructions decode into μop "flows" that exceed four μops or have other decoding complications. For these, the decode logic must rely on the support of a micro-op sequencer (MS) to generate the appropriate μop flow. For complex x86 instructions, decode must first wait for decoder lane zero (conventionally the most complex decoder) to be available, which then communicates with the MS to decode the instruction. The MS then provides multiple decoded μops per cycle (up to the decode width) until the flow is complete, and then the decoding process transfers back to the regular hard-wired decoders.

As far as the actual coding implementation goes, the entire decode pipeline is still simply a bunch of

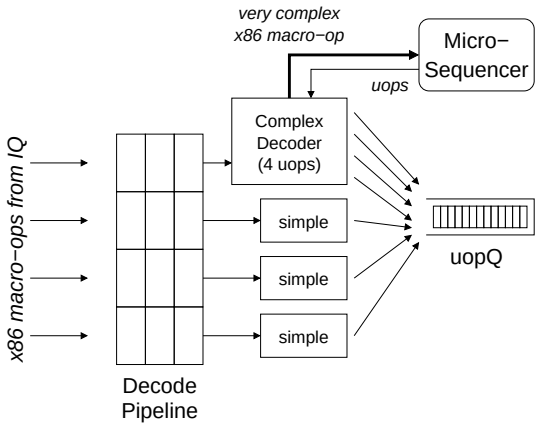

Figure 4: Block-diagram of the Zesto decode pipeline components.

pointers to macro-ops. Only at the last stage of the decode pipeline when uops get inserted into the uopQ do we separate the uops from the macro-ops. Macro-ops can only enter the decode pipeline subject to the decoder capacity/capability constraints, and then the simulator does nothing else beyond shuffling the macro-ops down the pipeline. When they reach the end, the only constraint is the capacity of the uopQ.

6 Allocation

The allocation pipeline deals only with uops, which are dequeued from the uopQ. This pipeline is also non-serpentine, and no real action occurs until the uops make it to the last stage of the pipeline. At this point, a uop may stall here for any number of cycles until all of the resources that it needs are available. For most uops, the only resources are an entry in the reorder buffer (ROB) and a reservation station (RS) entry. For loads and stores, a load or store queue entry, respectively, is also required. Zesto does not explicitly model physical register allocation or register renaming. Since we model a Pentium-Pro/Core-style execution core, ROB-entry allocation is effectively the same as physical register allocation. We do not maintain a register alias table (RAT), because the oracle already has all of the dependency information; allocation simply makes a copy of these pointers. When all resources are available, the uop can leave the front-end in-order pipeline and await scheduling and execution in the out-of-order engine. Allocation is also responsible for assigning an "execution port" or "port binding" to each uop, which will be explained in greater detail in the next

section.

Prior to insertion into the reservation stations, the allocation function also checks whether an instruction is ready to issue. If so, the instruction gets placed in a ready queue (readyQ). More about scheduling will be discussed in the next section.

7 Scheduling and Execution

The execution block (Figure 5) handles the dynamic scheduling of μops, movement between the reservation stations (RS) and the execution units, the actual execution, memory instruction scheduling, and result writeback. μops reside in the RS until they have received all input tags. At this point, the μop bids for permission to proceed to its functional unit, but the select-grant process occurs on a port-by-port basis. That is, on any given cycle, the ready μops bound to port 0 will compete to issue to port 0, and this occurs completely independently from the scheduling activities on any other ports. For a k-issue superscalar processor with n RS entries, such an organization only requires k separate n-choose-oldest-1 select blocks, rather than a single monolithic n-choose-oldest-k block which would be incredibly difficult to implement for current clock speeds. The tradeoff of such an approach is that even in the case of multiple ready μops bound to a single port, any unutilized functional units on other ports cannot be used by these μops. Most other timing simulators do not consider these types of resource constraints which can lead to overly optimistic instruction schedules. For some types of research studies, such detail will not be important; for studies involving the detailed design and modification of the instruction scheduling or execution logic, *not* considering this level of detail can lead to very different performance results and possibly impractical solutions.

There are two distinct loops related to the operation of the execution core: scheduling and execution. The instruction scheduling loop is the traditional wakeup and select process. Uops that issue broadcast their destination tags to all of their dependents. Uops that have received all of their input tags then compete for access to execution resources. When granted access, the uop issues. After issuing, the uop accesses the payload RAM which would contain its relevant information for execution (e.g., operands, destination, opcode, etc.). This payload access could take multiple cycles depending on the processor implementation, and also accounts for any additional wire delay between the scheduling core and the execution units. After leaving the payload RAM pipeline, the uop proceeds to its designated functional unit for the actual execution.

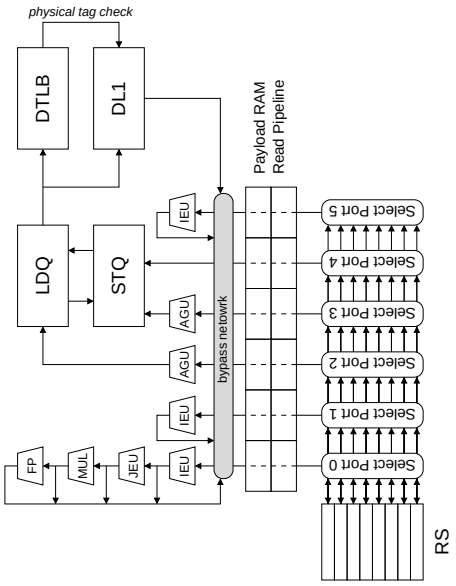

Figure 5: Block-diagram of the Zesto execution logic.

Only at this time does its RS entry get deallocated (more on scheduling mispredictions later). Finally, after execution, the uop forwards its result to its dependents and updates its status in the ROB.

The two distinct loops can potentially create a disconnect. For the two loops to remain in sync, the latency that the scheduler assumes for a uop must be the same as the latency that actually occurs when the uop executes. For many uops, the latency is fixed and this does not cause any complications. Load uops, however, can have very variable latencies depending on whether an access hits or misses in the DL1 cache (and then whether it hits in the L2, and also how long the queuing delays are between the different layers, and the variable access latency inherent in DRAMs). As uops leave the payload RAM and reach the execution units, the processor performs one last check to see if all input data operands are valid. This can happen, for example, if the input comes from a load that the scheduler had assumed would hit in the DL1 cache, but it turns out that the load missed and so the data are not yet available. As a result, the load's dependents reach execution before their data are ready. In this situation, execution gets aborted and the uop gets "snatched back" into the RS. Note that the uop's original RS entry has not yet been deallocated, so snatching back the uop only requires that the uops status be reset so that it will eventually attempt to issue again. The uop's invalid input will be reset, so that it will now re-await a new tag broadcast from its parent

to notify the dependency resolution of this operand.

The latency from issue until execute is a parameter that can be set by the simulator user. With a larger schedule-to-execute latency, a larger number of uops can potentially be mis-scheduled. A load's dependents may get mis-scheduled because a predicted cache hit turned out to be a miss. In the meantime, the dependent's dependents also get scheduled, and their dependents may get scheduled on the cycle after that. The simulator recursively traverses the data-dependency graph and resets the ready status for all inputs that correspond to invalid/not-yet-ready values. There are several possible implementations of this in hardware, but the simulator does not explicitly model the exact mechanisms. In practice, this is only feasible if the schedule-to-execute latency is short (i.e., ok for Core-style pipelines, not ok for super-pipelined Pentium4 styles).

The execution resources consist of multiple functional units, including simple integer units, multipliers, dividers, shifters, branch/jump units, load ports, store address ports, store data ports (see next section for more on store address/data handling), and floating point units. The units are organized into "execution ports." Each port effectively has a certain number of units statically bound to it. During allocation, each uop gets assigned to one of these ports ("port-binding"). In some cases, a uop can only be bound to one specific port (e.g., if port 4 is the only one with a divide unit, then all divide uops will always be bound to port 4). In other cases, more than one port may have an appropriate functional unit (port 1 and port 2 may both have simple integer ALUs), and so a port-binding decision must be made. The simulator currently implements a simple load balancing port-binding algorithm, where the allocator keeps track of how many uops are currently assigned to each port, and then assigns a uop to the valid port with the lowest load ("valid" meaning that the port contains the correct ALU type for the uop). Uops are effectively de-assigned from a port when they successfully execute (port loading information is updated at the same time as RS entry deallocation).

The simulator also explicitly models bypass network contention. We assume a single dedicated bypass bus per execution port, and so only one functional unit per port may writeback per cycle. Contention may occur when uops with different latencies issue on different cycles (since only one uop can issue per port, per cycle), but still end up completing execution on the same cycle. The current implementation does not attempt to arbitrate in any particularly intelligent fashion (e.g., oldest first), but what happens is that whichever uop

gets processed first (determined by an arbitrary (static) ordering of the ALUs) gets to go, and the other uop (or uops) wait until the next cycle.

8 Loads and Stores

Load and store execution is slightly different than that used in the traditional SimpleScalar family of timing simulators, but Zesto's implementation more closely models the approach taken by modern Intel x86 processors. The old Simplescalar implementation breaks each load and store into two components: an address computing operation and a data handling operation. The address computing operations are inserted into the RS and schedule just like any other operation, and the data handling operations are placed in the load-store queue. In Zesto, stores get decomposed into a store address (STA) uop and a store data (STD) uop, but both get inserted into the RS. The load is simply a single stand-alone uop.

We will explain the stores first, as they are actually simpler. The STA uop waits in the RS until its operands have arrived, it bids for access to a STA port, executes, and then deposits the computed address in the store queue (STQ). The STD uop also waits in the RS until the data operand has arrived, and then goes through the same process, ultimately depositing the data component of the store in the STQ as well. The STA and STD may issue out-of-order with respect to other. Finally, when both STA and STD components have made it to the STQ, the STQ broadcasts the address to younger loads to detect forwarding opportunities and/or load ordering violations. The microarchitectural reason for not placing the STD uop directly in the STQ as is done in the earlier SimpleScalar implementations is that this forces the STQ to also be directly involved in the regular execute loop. That is, the STQ entries would need CAM ports to monitor the bypass network to capture the store data values.

Load uops start off in the RS to await their operands as usual. After issuing and making its way through the payload RAM pipeline, the uop performs its first phase of execution on the address generation unit (on the load port), and this result gets written into the load's corresponding LDQ entry. From here, the load first checks to see whether it may continue any further. Typical reasons why it may not include earlier store addresses that are still unresolved, or the presence of a predicted store dependency even when loads are permitted to speculatively execute past unresolved stores. If the load is permitted to execute, it still must have access to a read port from the DL1 cache, the DTLB, and the STQ. If all are available, then the

DL1 and DTLB requests get queued, and the uop is also inserted into the STQ read pipeline. The STQ read pipeline operates like any of the other functional unit pipelines. When the uop reaches the end of the pipeline, we scan the STQ to see if there is a forwarded store hit. If so, we take that value and broadcast it to our dependents and that concludes execution. If not, then we simply wait for the load from the cache hierarchy. The cache requests use a callback facility, similar to that employed in the SimpleScalar 4.0 MASE simulator, that will update the uop's state and broadcast the result to its dependents when the access finally returns from memory.

In the case that a load's data spans more than one cacheline (x86 allows unaligned accesses), then two separate DL1 requests are issued, although we only perform a single STQ search (the model currently does not consider alignment issues within the STQ). The load is not complete until both accesses have returned. The current version does not properly handle the case where the load not only crosses cache-line boundaries, but also crosses page boundaries. In this case, an additional TLB lookup would also be required to find the physical page where the second portion of the data resides. The model simply only performs a single translation corresponding to the address of the first byte (lowest address) of the request.

Zesto models the STQ search as having the same latency as a DL1 hit; doing otherwise would cause great confusion for the out-of-order speculative instruction scheduling [8]. In the case of a hit, the value is forwarded to the load's dependents and that concludes the load's execution. Note that the forwarding can only come from a single STQ entry. It is possible that, for example, a four-byte load has a match on one byte with a one-byte store, and also has a match on two other bytes with a different two-byte store. The load's final value would then have to be stitched together from one byte from the first store, two bytes from the second, and one byte from the value retrieved from the data cache. Instead of attempting to model the rather complicated multi-match/multi-forward/splice-with-DL1 logic, Zesto simply forbids this case. In such a situation, the load must simply wait for all of the relevant stores to writeback to the data cache and then read all of the bytes from the DL1 in a single access. Partial forwarding may occur when the one sourcing store can provide *all* of the bytes for the requesting load. Note also that in the case of a STQ hit, the original DL1 and DTLB accesses continue to progress through the cache hierarchy, possibly using up bus bandwidth, occupying MSHR entries, and even triggering prefetches. When the request eventually comes back with its result, it will discover the load was already satisfied by the STQ and the result is dropped. It is possible that

when the cache request returns, the load does not even exist anymore (it may have committed or been flushed due to being on the wrong path of execution). In a real processor, it would be very difficult to have a STQ hit trigger a mechanism that somehow "chased down" the original load request (which may have even already left the chip to access main memory). The original load request does, however, generate additional activity in the cache hierarchy, and studies including such traffic when modeling the performance of cache/memory could observe different results.

9 Commit

The commit stage retires or commits instructions in program order. Normally (i.e., other non-x86 simulators) for a commit width of N, this just means that up to N instructions can commit on any given cycle. On an x86 machine, this works slightly differently due to the uop decomposition of the instructions/macro-ops. The Zesto commit stage can commit up to N uops per cycle, but macro-op commit must be all or nothing. Therefore, a uop cannot commit until its corresponding macro-op is ready to commit, and the macro-op cannot commit until all of its constituent uops have all completed execution. This prevents parts of a macro-op from updating architected state, only to later discover that another part caused a fault and this instruction should have been aborted. In addition to updating architected state, the commit stage also deallocates any remaining resources held by the uops (e.g., ROB entry, LDQ entry). Note that committing individual uops from the same macro-op may still be spread over multiple cycles (i.e., if the flow-length exceeds the commit width), but the commit process is still "atomic" per macro-op in that once the first uop is committed, it is guaranteed that all remaining uops from the flow will eventually be committed as well.

Store instructions are handled slightly differently. At commit, the store operation attempts to writeback its results to the cache hierarchy. The store is not permitted to commit until it has successfully enqueued requests to both the DL1 and the DTLB. After these transaction have been sent into the cache hierarchy, the store can deallocate its ROB entry. The store's STQ entry, however, does not get deallocated at this time. Since the store writeback operation may take several cycles (or several hundred cycles if there's a miss to main memory), the processor keeps a copy of the store data to forward to any subsequent loads. The STQ entries can only be deallocated when the entire store writeback procedure has fully completed. This portion of the STQ (stores that have committed but not deallocated) is also known as the *senior store queue*. The

simulator keeps a pointer to the senior STQ head, which points at the oldest committed but not deallocated store. When the store completes writeback, this pointer can be advanced to mark the deallocation of this STQ entry.

Stores that write to a data range that spans more than one cacheline are handled in a similar fashion as split-access loads. A second store write request is issued, but only one translation is performed. The (senior) STQ entry cannot be deallocated until both writes have completed.

10 Caches

Zesto's cache hierarchy implementation is somewhat more complex than the original sim-outorder version. The cache uses a variable-latency model with callback functions to initiate any actions that should be taken upon access completion (e.g., broadcast result to dependents, mark STQ entry ready for deallocation). Figure 6 sketches the cache processing steps. While this is illustrated as one flow-chart, the steps of this chart may be spread out over very many cycles (i.e., in the case of a cache miss). Requests are initially entered into the cache through the use of the cache_enqueue function, which can only happen if the appropriate bank is available. After some number of cycles (depending on the cache access latency), we check to see if the request hits in the cache (either in the main array or in the writeback-buffer/victim cache). On a hit, the appropriate callback functions are invoked and that ends the cache request. On a miss, a different callback function may be called, for example, to notify the instruction scheduler of a latency misprediction. Also on a miss, a MSHR entry must be allocated. If no MSHR entries are available, then the request stalls in the cache bank (i.e., the bank is locked up, which may prevent other requests to this bank from getting serviced) until an MSHR entry becomes available. After the request is placed in an MSHR entry, it can be forwarded to the next-level cache when the bus to the next level is available (and provided that the destination bank in the next-level cache is not locked up). The user can specify different MSHR scheduling policies either using a first-come-first-serve order, or by specifying a priority ordering between loads, stores, prefetches and writebacks. Once enqueued in the next level cache, this same process repeats all over again, but of course being applied to the data structures of the next-level cache. If a request for the same address is already pending in another MSHR entry, a new MSHR entry is still allocated, but the request is not forwarded to the next-level cache. Instead, the MSHR entries are linked so that when the first request is satisfied, the result

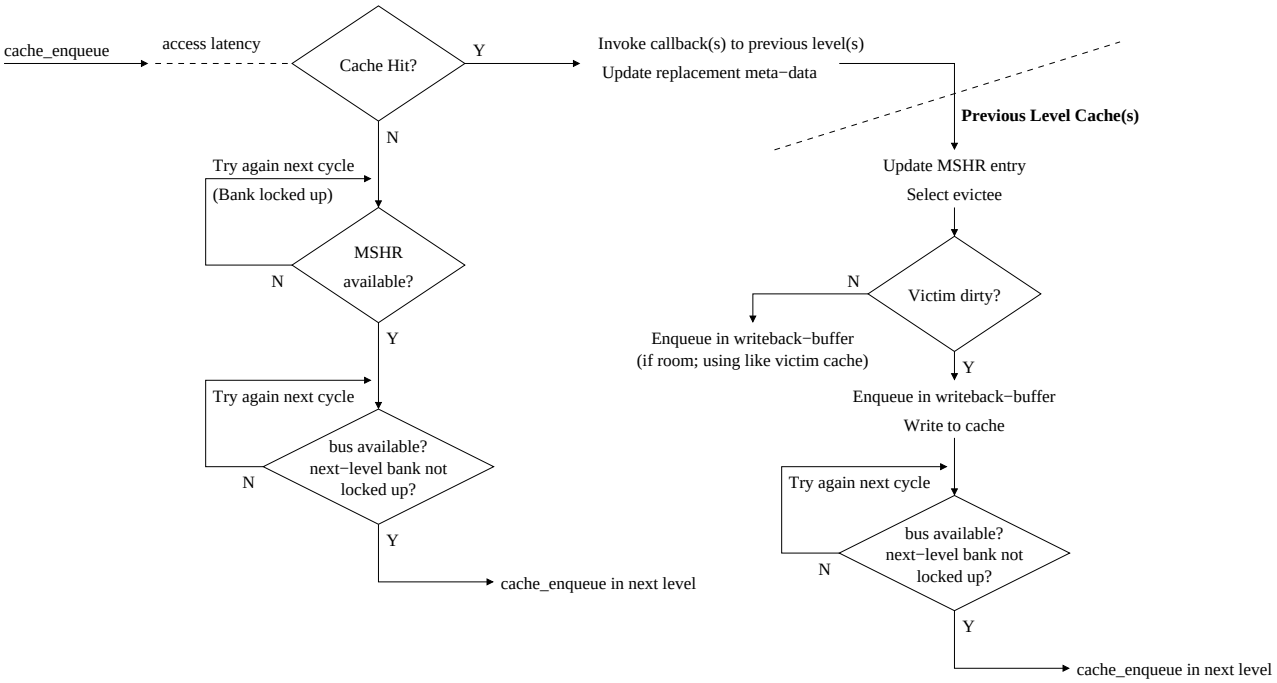

Figure 6: Sketch of the steps involved in cache processing.

can be forwarded to any other MSHR entries requesting the same cache line.

Eventually the request will get satisfied at some level of the memory hierarchy (possibly from main memory), at which point multiple callback functions may be invoked. The first is a request-specific callback function that notifies the original requester that the requested data is now available. For instruction fetches, the callback function updates the corresponding data structures in the fetch stage. For data requests, the callback function causes the output value to be propagated (bypassed) to the load's dependent instructions, and also makes sure that any scheduling issues get taken care of (in the presence of speculatively misscheduled operations, the dependents may need to be rescheduled). A separate fill callback function is responsible for propagating the requested cacheline to the appropriate caches (e.g., a cacheline read from main memory may need to be installed in the L2 and the DL1 caches). This function effectively updates the corresponding MSHR entries with the cacheline. Before deallocating the MSHR entry, the cache fill must be completed. The cache first selects a victim, and inserts the victim in the writeback buffer (which also serves as a victim cache). The cache line can then be installed where the victim used to reside, the cache replacement metadata is updated, and the MSHR entry can be deallocated. Dirty lines in the writeback buffer must contend

with the MSHR for access to the bus to the next level to perform their writeback operations.

Each cache can have one or more hardware data prefetchers associated with it. Either on every access, or only on misses (controlled by -X:pf:miss (where X specifies the cache, e.g., -dl1:pf:miss)) the prefetchers are consulted to see whether there are any predicted cachelines that should be requested. The request is enqueued in a prefetch FIFO (PFF). The FIFO is managed in a strict circular-queue fashion, which means it may be possible that a request overwrites a previous request that has not yet been acted on. This is by design; the intuition is that if a prefetch request has not been acted upon by the time another request overwrites the first one, then the original prefetch probably would not have been of much use from a timeliness perspective.

Even though a prefetcher may predict a line to be prefetched, the decision as to whether this prediction should be acted upon is still subject to several constraints. Prefetching may be prevented if the MSHRs already contain too many requests, demand or prefetch (controlled by -X:pf:thresh); if there are already too many outstanding prefetches occupying MSHR entries (-X:pf:max); or if the utilization of the bus to the next level is too great. The bus utilization throttle system uses two threshold to provide some hysteresis. If the bus utilization falls below a low watermark (-X:pf:lowWM) then prefetching is enabled. If the bus utilization increases above the high watermark (-X:pf:highWM), then prefetching is prevented. The utilization monitored over fixed time intervals (specified by -X:pf:WMinterval).

10.1 Caches and System Calls

The Zesto system call implementation is largely the same as that of the original SimpleScalar approach. That is, the pipeline is drained, and then the system call is *functionally* executed without any timing impact. Zesto adds the option that any memory accesses invoked by the system call can be filtered through the cache hierarchy. When the system call is functionally executed, the memory handler logs all of the read/write activity. The current implementation of the system call memory handlers process everything on a byte-by-byte basis, so any back-to-back reads from the same eight-byte block are all treated as the same memory access (similar for writes), otherwise the memory hierarchy can get swamped with a whole mess of tiny one-byte accesses. Before the oracle allows the pipeline to continue executing past the system call, it first enqueues all of these requests to the cache hierarchy (DL1 only; DTLB/translations are not done), with a configurable delay between requests. This is still only an approximation as this approach does not track

dependencies between the memory requests, and so latencies that are dependent on cache hits/misses will not be accurately reflected. This does, however, help to keep the cache hierarchy a little more up-to-date.

11 Main Memory

Upon a miss in the last level cache, a request is sent across the front side bus (FSB) to the memory controller (MC). The MC contains a queue that is similar to the MSHRs. This queue contains all of the requests to read or write memory. The MC implementation is much more sophisticated than the model used in sim-outorder/sim-mase (i.e., no memory controller), but is still simpler than some other models [3]. In particular, Zesto only implements a single, unified request queue for all reads and writes (as opposed to separate RRQ and WRQ's). The MC assumes an open-page mode for memory access, and therefore attempts to schedule accesses to the same pages together to maximize row buffer hits [6]. If no access to the same page can be found, then the MC reverts to a simple first-come-first-serve scheduling.

The actual memory latency depends on the underlying memory model (see ZCOMPs section). The default release of Zesto contains two simple memory models. The first is identical to the original SimpleScalar model where the main access takes a constant number of cycles, and subsequent chunks require some additional constant number of cycles. The second model is a simple SDRAM model that tracks individual ranks and banks, noting the current open page and accounting for pre-charge latencies, CAS and RAS latencies, and refresh effects.

12 x86-isms

There are several features in the x86 ISA that require slightly different support in the microarchitectural simulator that are not needed in most other ISAs. Some of these are described here.

12.1 Partial Register Accesses

Due to the evolution of the x86 architecture from the original 8-bit registers to the 16-bit and 32-bit versions (SimpleScalar-x86/Zesto does not currently support the 64-bit extensions), there exists architecture-level support for accessing registers at different bit-width granularities. This introduces complexity due to the

fact that instructions of varying widths can be intermixed. For example, consider following instructions:

- MOV EAX, 0xFFFFFFFF
- MOV AX, 0xBBBB
- MOV AL, 0xDD
- ADD EBX, EAX

The resulting value in EBX (0xFFFFBBDD) is a combination of the previous three instructions. Making all three instructions explicit input dependents for the ADD instruction is impractical because it forces each uop to support up to six inputs! (Consider the case where EBX is also formed from three similar operations.) This in turn would greatly increase the complexity of the reservation stations since they would need to track up to six inputs per entry. Furthermore, this also creates an aliasing/identity problem, as different names can correspond to the same physical locations.

In the original Pentium processors, such *partial register* accesses would lead to pipeline stalls. Instead of attempting to splice a value together from multiple previous outputs, the pipeline would simply wait until each partial write had written its result back to the architected register file. At that point, the RF would have merged all of the results, and a single read from the RF would deliver the correct value. These partial register stalls were a source of considerable performance degradation in legacy codes. Modern x86 implementations have removed the impact of such stalls. While we do not know the exact implementation of how these stalls are removed, the Zesto simulator implements this by forcing all instructions to always output a full 32-bit value. In some cases, this simply requires reading the original register value as an extra operand. For example, "MOV AX, BX" becomes "MOV' EAX, EBX, EAX", where the operation MOV' reads the bottom 16 bits from EBX, the top 16 bits from EAX, and splices them together to output the final 32-bit value for EAX. In some situations where a uop cannot support the additional input operand (i.e., the original uop already has the maximum number of inputs allowed per uop), then we inject an additional merging uop. This additional uop is architecturally invisible. For example, "ADD AL, BL" becomes "ADD' tmp, AL, EBX" followed by "MERGE_L EAX, tmp, EAX". ADD' is simply the RISC form for the 16-bit add. This uop sequence uses an architecturally invisible temporary register (which can be renamed) to store the result of the add, and then the MERGE uop splices in the original upper 24 bits of EAX (the newly updated

lower 8 bits corresponding to AL come from the tmp register). This approach avoids the stalls due to partial register accesses, but it can potentially increase the critical path length through the code due to the additional MERGE uops.

This always-merge approach can also introduce some additional data dependencies. For example, the code

- ADD AH, 0xFF
- ADD AL, 0xBB

should theoretically be able to execute in a single cycle since the two operations are in fact independent. With our merging-uop approach, this actually requires *four* dependent uops:

- ADD' tmp, AH, 0xFF
- MERGE_H EAX, tmp, EAX
- ADD' tmp, AL, 0xBB
- MERGE_L EAX, tmp, EAX

Our native uop format actually supports three-input operations, and so for many of the partial register writing uops, the merge is automatically folded in without needing the additional MERGE uops. Refer to `target-x86/x86flow.def` to see all of the cases where MERGE is employed. Zesto also supports uop-fusion (described later) to reduce the overhead of these additional partial-merge uops.

The choice of three inputs is due to address computation instructions (e.g., STA, LD) that can potentially have up to three register operands (i.e., register, base and index). The entire uop-flow implementation has been rewritten to support three-operand uops, which is more realistic than the four-uop format in the original SimpleScalar/x86 pre-release.

12.2 Flags

Many/most integer operations in x86 set one or more of the flag bits (e.g., carry, zero) in the flag register. A few instructions read the flags (e.g., jump conditional), and so the flags introduce another source for

data dependencies. In Zesto, the flag data dependency is treated very similarly to other normal register dependencies. A uop that reads a flag adds itself to the odep list of the previous instruction that writes the flags. While PTLsim models the flags as three separately rename-able banks of flags [9], Zesto treats the flags as a single renamed register. This approach can introduce some additional data-dependencies similar to those due to merging of partial register writes. For example

- SET ZF
- SET CF
- JNZ (conditional jump based on flag ZF)

Strictly speaking, the JNZ should only have to wait on the first SET instruction. Due to our single bank of flags, this really breaks down into:

- SET.ZF FLAGS
- SET.CF FLAGS
- JNZ FLAGS

where each instruction is serially dependent on the previous due to data dependencies through the FLAGS register. Splitting this into separately renamed portions is left for future implementations of Zesto. This should not be a major performance limiter as many operations set all of the flags (e.g., ADD) and so do not need to be serialized with respect to earlier reads and writes of the flag. If the setting and usage of the flags are not separated by other flag modifying instructions, then this is also not an issue.

12.3 REP

The x86 ISA allows instructions to have a REP (repeat) prefix. There are a couple of variants that allow for very compact (i.e., single macro-op) encodings of fairly complex operations such as memset, memcpy and strcpy. To support these instructions in Zesto, each iteration is treated as a separate macro-op with respect to the simulator's data structures (i.e., one `struct Mop_t` instance per iteration), but all of the bookkeeping treats the operation as a single macro-op. Therefore, a thousand-iteration REP instruction may decompose

into one thousand `struct Mop_t`'s each of four uops a piece, but the simulator counts this entire flow of one thousand macro-ops as a single x86 instruction with respect to the instruction simulation limit and statistics keeping (e.g., this will be reported as a single macro-op with a four-thousand uop flow length). For commit, each `struct Mop_t` can be committed separately (this is necessary as the reorder buffer is not likely able to buffer thousands of Mops).

The REP instructions introduce additional control flow at the uop level. After each iteration's worth of uops, a conditional branch is effectively needed to test whether the REP exit condition has been met. We explicitly model this by injecting additional "micro-jump" uops. Prior to the first iteration, a single micro-jump is injected to test for zero-iteration REP instructions (this seems pointless, but such instructions can exist and so must be checked for to ensure correctness). Then each full iteration gets one micro-jump appended to the end of the iteration's uop flow (along with a uop to decrement the index/counting register). Due to the extra control, REP instructions can only be decoded by decoder number zero as we assume that assistance from the microcode sequencer is required. Note that a zero-iteration REP still actually counts as a single macro-op in the simulator.

12.4 uop-Fusion

Modern Intel x86 implementations use an optimization called uop fusion. For the purposes of most of the pipeline stages, what would be two separate uops in previous implementations are combined or fused into a single uop. For example, a store instruction would traditionally get decoded into STA and STD uops, which would occupy two slots in the uopQ, two slots in the allocation pipeline, two RS entries, and two ROB entries. With uop-fusion, the decoder simply emits a single "store" uop that only occupies a single entry in each of the previously mentioned structures. During schedule and execute, the two halves of the uop can issue and execute independently as if they were in fact two separate, distinct, traditional uops. To implement fusion in Zesto, the simulator simply links all fused uops together in a linked list (right now, only pairs of uops get fused, but this structure makes it relatively simple to fuse uop-triplets (or more) if desired).

From a code perspective, each of the component uops is tracked as its own separate uop data structure and then they are just linked together. In a real implementation, only a single uop would be emitted by the decoder. For the purposes of computing uops-per-cycle (uPC) rates, we also only count the entire fused uop

as a single uop. If you are interested in the equivalent original uop rate, then the *effective* uops per Cycle (euPC) metric provides this value. For example, the x86 macro operation "ADD [EAX], EBX" counts as a single Mop, would get decoded into four basic uops (load, add, store-addr, store-data), or get decoded into two fused uops if fusion is enabled (load-add, sta-std). For computing IPC, this counts as one instruction; for computing uPC, this counts as two (fused) uops; for computing euPC, this counts as four "primitive" uops.

13 Zesto Components (ZCOMPs)

Besides providing a detailed microarchitecture simulator for a modern x86-based architecture, Zesto also aims to be a flexible platform for rapid computer architecture exploration (rapid at least in terms of development/debugging effort). To that extent, several structures have been factored out into what we call Zesto Components, or ZCOMPs for short. Each family of ZCOMPs has a C++ virtual base class, and so adding new components is straightforward.

This componentized organization started many years ago when hacking the old SimpleScalar bpred.c/h implementations where everything was smashed into the same data structures and functions. This made it very error prone to add new branch prediction algorithms, and generated very large, messy switch statements in multiple locations throughout bpred.c. The current implementation places each algorithm in its own distinct .cpp file.

13.1 Example

As an example, let us consider the branch predictor ZCOMPS. Let us assume we want to add a new branch prediction algorithm that uses a table of saturating 2-bit counters (e.g., bimodal [7]). When the counter is strongly-taken or strongly not-taken, the prediction follows the counter. If the counter is in either of the weak states, then the predictor falls back on a backwards-taken/forward-not-taken static prediction. To start, one would simply make a copy of the existing bimodal predictor implementation:

- cd ZCOMPS-bpred
- cp bpred_2bc.cpp bpred_mypred.cpp

Open up the file bpred_mypred.cpp with your favorite editor (e.g., vim or emacs). There are several locations in the file where you want to replace the string "2bC" with the string of your choice (e.g., "mypred"). The easiest thing to do in this example is to perform a global search and replace (e.g., ":%s/2bC/mypred/g" in vim).

Now, go to the Lookup function at line 69. BPRED_LOOKUP_HEADER is a macro that creates a standard lookup-method declaration consistent with the declaration in the parent virtual class. To implement the desired change, replace line 76 with:

```
if(*sc->current_ctr == MY2BC_WEAKLY_TAKEN ||||
        *sc->current_ctr == MY2BC_WEAKLY_NT)
                return (tPC < PC)
else
                return MY2BC_DIR(*sc->current_ctr);
```

This tests to see if the selected saturating counter is in one of the "weak" states and then returns the corresponding prediction. The variable tPC is the target PC, and so if it is less than PC, this is a backwards branch. At this point, save the file, return to the main Zesto directory, and issue a "make sim-zesto" or "make sim-bpred" command. This will now recompile the simulator. Note that you do not need to inform the Makefile about this new file bpred_mypred.cpp. The Makefile automatically invokes a script that checks the ZCOMPS-bpred directory for all branch prediction implementations and automatically includes all of these. This in turn generates a .list file that has a list of #include directives to cause all of these individual files to get imported into zesto-bpred.cpp (see ZCOMPS-bpred.list). The zesto-bpred.cpp file includes this list at two separate points. One is to simply import all of the definitions of the derived classes. The other is for predictor initialization. At the top of bpred_mypred.cpp, you can find a section that is enclosed by "#ifdef BPRED_PARSE_ARGS" which contains a small code snippet to parse the command line options for the predictor configuration. If you run "./sim-zesto -bpred mypred:foo:8192 ...", then the string "mypred:foo:8192" will be stored in the variable `opt_string`. We typically just use a simple `sscanf` call to parse out the arguments and then pass them to the actual predictor creation function (constructor).

Note that the constructor can have whatever arguments you want for your predictor. These just need to be properly passed in when the constructor gets called from the command-line argument parsing code

(located at the top of the file). The main point of this is that the user should be able to quickly implement new algorithms by simply copying one of the existing files, make the relevant changes, and recompile.

13.2 Other ZCOMPs

Besides, the branch direction predictors, Zesto supports several other ZCOMPs. These include a branch prediction fusion/hybridization mechanism, branch target buffers, return address stack (RAS) predictors, memory dependence predictors, cache prefetchers, memory controllers, and main memory/DRAM models.

The branch prediction fusion components are algorithms for taking multiple component branch direction predictors and combining them to provide a better overall prediction. For example, one can specify a 2K-entry 2bC predictor, a 4K-entry gshare predictor with a 7-bit global history, and a 4K-entry McFarling-style tournament meta-predictor by specifying: "-bpred 2bc:2bC:2048 2lev:gshare:1:4096:7:1 -fusion mcfarling:meta:4096". The command line option -bpred can take multiple branch prediction configuration strings (if this is the last argument before providing the executable name, use a single '-' to terminate the list of predictor configuration strings). It is not a requirement, but the convention used here is that configuration strings take the format "algorithm:name:opt1:opt2:..." where the additional name field allows the user to differentiate between variants of the same underlying algorithm (e.g., "-bpred 2lev:**gshare-short**:1:4096:4:1 2lev:**gshare-long**:1:4096:12:1 2lev:**PAs**:1024:8192:6:0"). These names are used in the final statistics dump so that you can easily differentiate between the component predictors.

The BTB and RAS predictors are fairly straightforward. The BTB is similar in structure to the branch direction predictors, expect the BTBs return addresses instead of boolean predictions.

The memory dependence predictor provides a 0/1 prediction on whether a load can safely execute without incurring an ordering violation. The current implementation is not quite true to a real machine implementation, as the `check_load_issue_conditions` function can call the dependence prediction lookup function every cycle. For a real implementation, the prediction should probably be made once up front somewhere in the decode pipeline. The current implementation, however, makes it much easier to support an oracle dependence predictor.

The cache prefetchers basically only have one interface ("lookup") of particular interest. The lookup function informs the prefetcher of the current memory access. The prefetcher then returns an address to

prefetch from and it also updates its own internal state, if any.

The memory ZCOMPs are also fairly straightforward. The access function provides the address to be read/written, and the function returns the latency of the operation. The other function is refresh, which does not take any arguments and invokes the component's refresh handler.

14 Multi-Core Support

The current version of Zesto provides some limited support for modeling multi-core processors. The system calls required for "true" multi-threaded apps are not implemented (e.g., fork, join), and so the current version can only handle multi-programmed workloads (i.e., no sharing of memory between threads). For N independent programs, each program will be loaded from a .eio file (only EIO traces are supported in multi-core mode), and the state will be captured in separate `struct thread_t`'s. That is, each program has its own dedicated oracle functional simulation state. Then, each of these `struct thread_t`'s is associated with a separate `struct core_t` that represents the actual microarchitectural core. Each core effectively has its own private copy of every microarchitectural module, with the exception of uncore components such as the L2 cache (including the L2 bus) and main memory (including the memory controller and FSB). The simulator proceeds cycle by cycle, where in each cycle, Zesto simulates one pipeline stage at a time for each core. For example, a two-core simulation would proceed as: commit(core_0), commit(core_1), exec(core_0), exec(core_1)... There are a few stages where we change the order of the cores each cycle to prevent one core from always getting to "go" first. This is important since some stages (e.g., fetch, LDST_exec) may cause the cores to access shared resources (e.g., L2 bus or L2 cache), which using a fixed order would allow one core to always get "first dibs" on these resources. For these stages, we basically implement a round-robin policy where on each cycle a different core receives "first dibs."

Each of the simulated processes/programs has its own dedicated 32-bit virtual memory space. This, however, must get mapped into a single shared physical memory space. We implement this by mapping each virtual page to its own unique physical page, and by having all cache hierarchy components (and main memory, of course) be completely physically indexed in the underlying simulator implementation (this may be orthogonal to whether you assume that a DL1 is, say virtually indexed physically tagged). Each {thread-id,virtual-page} pair gets mapped to a physical page on a first-come-first-serve basis. On a memory access,

the simulator checks its own internal "page table" to see if it has seen this particular {thread-id,virtual-page} combination before. If so, then it returns the mapped physical page number. If not, it assigns the next sequentially available physical page (we never deallocate physical pages, and so the physical page free pool is simply a pointer that keeps getting incremented).

Simulating multiple processes at the same time causes a secondary problem in that the simulator itself is running in a 32-bit memory space and can easily run out of memory. To get around this, Zesto actually only keeps a limited amount of the simulated memory in actual memory, and stores the rest in a backing file on disk. Simulating, say, eight different SPEC2006 applications (many of which have large working sets) ends up causing a lot of swapping to the disk, which makes the simulation very slow, but you can still do it. Without the paging, the simulator would simply run out of memory and die.

Note that the current implementation has absolute *no support* for modeling cache coherence. Since we are not currently supporting multi-threaded applications with shared memory, cache misses due to invalidations are simply not an issue. From a timing perspective, however, we are not accounting for potential contention on the coherency buses and cache snoop ports, and so this can introduce some error for workloads with high amounts of write traffic.

Running the Simulator

The command line for running Zesto in dual-core mode is:

./sim-zesto {*options*} -fastfwd F -max:inst M -tracelimit T -cores 2 app1.eio app2.eio

All cores are identical, and so whatever core-related options are specified by {*options*} is applied to all cores. Gzip compressed .eio files are also allowed (e.g., app1.eio.gz), and to simulate more cores, just increase the argument for the -cores option and list the appropriate number of .eio files. The fast-forwarding skips the first F instructions of each application in a round-robin fashion. The branch predictors and caches can be warmed during this phase by including the -warm:bpred and -warm:caches options. Note that for the last-level cache, this is an approximation to truly warmed cache state because all programs are progressing at the exact same rate, whereas in a cycle-level simulation the variances in the IPC rates would cause different memory reference patterns.

The limit -max:inst specifies how many instructions each application should have its statistics collected over. That is, after skipping the first F instructions, timing information (e.g., total cycles, IPC, etc.) will only be collected for the next M instructions. Since different applications will reach the M-instruction limit at different times, Zesto "freezes" the stats for any application that reaches the limit, but otherwise allows that application to continue executing. In this fashion, (almost) all of the statistics reported at the end of simulation are all based on the same number of executed instructions.

In a scenario with shared resources (e.g., front-side bus contention), it is usually not appropriate to simply allow one program to simply stop execution (especially in the context of using some form of sampled or reduced benchmark subsetting) as the remaining cores would no longer have to compete with this completed application. So even though we may only want to simulate, say, 10 million instructions per benchmark, if benchmark A has an IPC of 1.0, and benchmark B has an IPC of 0.1, then A will finish its first 10 million instructions in 10 million cycles, while B will take 100 million cycles to execute the same number of instructions. Note that A will effectively execute an additional 90 million instructions while waiting for B to finish up. Although the statistics are still collected for only 10 million instructions per benchmark (20 million total), the simulator still must do the work of simulating a total of 110 million instructions: 90 million more than we collect statistics for or an overhead of 4.5×! One of the output stats ("loop_inst_overhead") reports the total additional instruction overhead due to the continued simulation of cores that have reached their instruction limits.

Since .eio files only support a finite sample of execution, an application that has reached its limit but is left running will eventually run out of work that can be simulated. At this point, the simulator simply restarts the program's execution from the start of the .eio file (no additional fast-forwarding). The -tracelimit option specifies the maximum number of instructions supported by the .eio files. For our simulations, we typically do something like collect .eio files that can support (T=) one billion instructions each, fast-forward for (F=) 500 million instructions to warm the caches, and then perform detailed simulation for (M=) 250-500 million instructions.

A Sim-Zesto Command Line Options

General:

- **-config** <**string**>: load configuration from a file
- **-dumpconfig** <**string**>: dump configuration to a file
- **-h** <**bool**>: print help message
- **-seed** <**int**>: random number generator seed (0 for timer seed)
- **-q** <**bool**>: initialize and terminate immediately
- **-ignore_notes** <**bool**>: suppresses printing of notes
- **-redir:sim** <**string**>: redirect simulator output to file (non-interactive only)
- **-redir:prog** <**string**>: redirect simulated program output to file
- **-nice** <**int**>: simulator scheduling priority
- **-** <**bool**>: ignored flag used for terminating list options
- **-fastfwd** <**long long**>: number of inst's to skip to *per benchmark* before timing simulation
- **-tracelimit** <**long long**>: maximum number of instructions *per trace/EIO file*
- **-cores** <**int**>: number of cores (if > 1, must provide this many eio traces)
- **-max:inst** <**long long**>: maximum number of inst's to execute *and collect stats for* **per core**
- **-max:uops** <**long long**>: maximum number of uop's to execute *and collect stats for* **per core**
- **-model** <**string**>: pipeline model type (e.g., DPM, STM)

Fetch related:

- **-bpred** <**string list...**>: branch direction predictor configuration string(s)
- **-bpred:fusion** <**string**>: branch prediction fusion/meta-prediction algorithm configuration string
- **-bpred:btb** <**string**>: branch target buffer configuration string
- **-bpred:ibtb** <**string**>: indirect branch target buffer configuration string
- **-bpred:ras** <**string**>: return address stack predictor configuration string
- **-jeclear:delay** <**int**>: additional latency from branch-exec to jeclear (front-end flush)
- **-il1** <**string**>: 1st-level instruction cache configuration string
- **-itlb** <**string**>: instruction TLB configuration string
- **-il1:pf** <**string list...**>: 1st-level instruction cache prefetcher configuration string(s)
- **-il1:pf:fifosize** <**int**>: IL1 prefetch FIFO size (entries)
- **-il1:pf:buffer** <**int**>: IL1 prefetch buffer size (entries)
- **-il1:pf:filter** <**int**>: IL1 prefetch filter size (entries)
- **-il1:pf:filterreset** <**int**>: IL1 prefetch filter reset interval (cycles)

- **-il1:pf:thresh** <**int**>: IL1 prefetch threshold (only prefetch if MSHR occupancy < thresh)
- **-il1:pf:max** <**int**>: Max IL1 prefetch requests in MSHRs at a time
- **-il1:pf:lowWM** <**double**>: IL1 low watermark for prefetch control
- **-il1:pf:highWM** <**double**>: IL1 high watermark for prefetch control
- **-il1:pf:WMinterval** <**int**>: IL1 sampling interval (in cycles) for prefetch control (0 = no PF controller)
- **-il1:pf:miss** <**bool**>: generate prefetches for IL1 only from miss traffic
- **-byteQ:size** <**int**>: number of entries in byteQ
- **-byteQ:linesize** <**int**>: linesize of byteQ (in bytes)
- **-predecode:depth** <**int**>: number of stages in predecode pipe
- **-predecode:width** <**int**>: width of predecode pipe (in Macro-ops)
- **-IQ:size** <**int**>: size of instruction queue (in Macro-ops - placed between predecode and decode)
- **-warm:bpred** <**bool**>: warm bpred during functional fast-forwarding

Decode related:

- **-decode:depth** <**int**>: decode pipeline depth (stages)
- **-decode:width** <**int**>: decode pipeline width (Macro-ops)
- **-decode:targetstage** <**int**>: stage of decode pipeline where the branch address calculator resides
- **-decode:branches** <**int**>: max branches decoded per cycle
- **-decoders** <**int list...**>: maximum number of uops generated for each decoder (e.g., 4 1 1)
- **-MS:latency** <**int**>: additional latency for accessing ucode sequencer (cycles)
- **-uopQ:size** <**int**>: number of entries in uopQ
- **-fuse:none** <**bool**>: disable all uop fusion rules
- **-fuse:all** <**bool**>: enable all uop fusion rules
- **-fuse:loadop** <**bool**>: enable load-op fusion
- **-fuse:stastd** <**bool**>: enable sta-std fusion
- **-fuse:partial** <**bool**>: enable partial register write merging fusion

Allocation related:

- **-alloc:depth** <**int**>: alloc pipeline depth (stages)
- **-alloc:width** <**int**>: alloc pipeline width (uops)
- **-flush:drain** <**bool**>: use drain-flush after misprediction (instructions stall at alloc until back-end drained)

Execution related:

- **-rs:size** <**int**>: number of reservation station entries
- **-ldq:size** <**int**>: number of load queue entries
- **-stq:size** <**int**>: number of store queue entries
- **-exec:width** <**int**>: maximum issues from RS per cycle (equal to num exec ports)
- **-payload:depth** <**int**>: number of cycles for payload RAM access (schedule-to-exec delay)
- **-fp:penalty** <**int**>: extra cycle(s) to forward results to FP cluster
- **-pb:ieu** <**int list...**>: IEU port binding
- **-pb:jeu** <**int list...**>: JEU port binding
- **-pb:imul** <**int list...**>: IMUL port binding
- **-pb:idiv** <**int list...**>: IDIV port binding
- **-pb:shift** <**int list...**>: SHIFT port binding
- **-pb:fadd** <**int list...**>: FADD port binding
- **-pb:fmul** <**int list...**>: FMUL port binding
- **-pb:fdiv** <**int list...**>: FDIV port binding
- **-pb:fcplx** <**int list...**>: FCPLX port binding
- **-pb:lda** <**int list...**>: LD port binding
- **-pb:sta** <**int list...**>: STA port binding
- **-pb:std** <**int list...**>: STD port binding

For the execution port bindings, specifying -exec:width as 4 (for example) defines four execution ports 0, 1, 2 and 3. For each -pb:X option, you specify exactly which ports a particular type of functional unit is present on. For example, "-pb:ieu 0 3" says there are two integer execution units, and they are located on ports 0 and 3. This means that any instruction using an IEU must be assigned to either execution port 0 or 3 (this assignment occurs during the alloc stage). See the config/merom.cfg file for an example functional unit port binding, which is our best guess for the port bindings of an Intel Merom-generation Core 2 pipeline based off of whatever information we could dig up off the Internet.

- **-X:lat** <**int**>: X execution latency (cycles)
- **-X:rate** <**int**>: X execution issue rate (cycles)

For each functional unit type, there are two options related to the operation latency and issue rate. The execution latency is simply the number of cycles that it takes for an instruction to complete execution; note that this does not include any of the "pre-execution" work such as scheduling or payload-RAM access (e.g., integer addition (IEU) has a latency of one cycle). The issue rate option is used for modeling partially-pipelined or non-pipelined functional units. It specifies how often a new operation can be initiated. Setting the issue rate to one cycle means that the unit is fully pipelined (i.e., a new operation can issue every cycle), whereas setting the rate to be equal to the operation latency models a completely non-pipelined functional unit. One can also set the issue rate somewhere in between these two extremes to model partially pipelined units (e.g., floating point divide where the normalization work can be overlapped with the iterative portion of execution).

- **-dl1** <**string**>: 1st-level data cache configuration string
- **-dl1:mshr_cmd** <**string**>: 1st-level data cache MSHR request scheduling policy
- **-dtlb** <**string**>: data TLB configuration string
- **-dtlb2** <**string**>: L2 data TLB configuration string
- **-dl1:pf** <**string list...**>: DL1 prefetcher configuration string(s)
- **-dl1:pf:fifosize** <**int**>: DL1 prefetch FIFO size (entries)
- **-dl1:pf:buffer** <**int**>: DL1 prefetch buffer size
- **-dl1:pf:filter** <**int**>: DL1 prefetch filter size
- **-dl1:pf:filterreset** <**int**>: DL1 prefetch filter reset interval (cycles)
- **-dl1:pf:thresh** <**int**>: DL1 prefetch threshold (only prefetch if MSHR occupancy < thresh)
- **-dl1:pf:max** <**int**>: Maximum DL1 prefetch requests in MSHRs at a time
- **-dl1:pf:lowWM** <**double**>: DL1 low watermark for prefetch control
- **-dl1:pf:highWM** <**double**>: DL1 high watermark for prefetch control
- **-dl1:pf:WMinterval** <**int**>: sampling interval (in cycles) for DL1 prefetch control (0 = no PF controller)
- **-dl1:pf:miss** <**bool**>: generate DL1 prefetches only from miss traffic
- **-dl2:*** <**xxx**>: all of the same knobs are applicable to the per-core DL2 caches as well
- **-warm:caches** <**bool**>: warm caches during functional fast-forwarding
- **-memdep** <**string**>: memory dependence predictor configuration string
- **-rob:size** <**int**>: number of reorder buffer entries
- **-commit:width** <**int**>: maximum number of uops committed per cycle
- **-commit:branches** <**int**>: maximum number of branches committed per cycle

Uncore related:

- **-LLC** <**string**>: last-level data cache configuration string
- **-LLC:mshr_cmd** <**string**>: last-level data cache MSHR request scheduling policy
- **-LLC:bus** <**int**>: CPU clock cycles per LLC-bus cycle (this is the bus between the core(s) and the LLC, *not* the FSB)
- **-LLC:rate** <**int**>: access LLC once per this many CPU clock cycles (for modeling LLC's clocked at slower speeds)
- **-LLC:mshr_fcfs** <**bool**>: use first-come first-serve scheduling of MSHR requests (default prioritizes loads)

- **-LLC:pf** <**string list...**>: LLC prefetcher configuration string(s)
- **-LLC:pf:fifosize** <**int**>: LLC prefetch FIFO size
- **-LLC:pf:buffer** <**int**>: LLC prefetch buffer size
- **-LLC:pf:filter** <**int**>: LLC prefetch filter size
- **-LLC:pf:filterreset** <**int**>: LLC prefetch filter reset interval (cycles)
- **-LLC:pf:thresh** <**int**>: LLC prefetch threshold (only prefetch if MSHR occupancy < thresh)
- **-LLC:pf:max** <**int**>: maximum LLC prefetch requests in MSHRs at a time
- **-LLC:pf:lowWM** <**double**>: LLC low watermark for prefetch control
- **-LLC:pf:highWM** <**double**>: LLC high watermark for prefetch control
- **-LLC:pf:WMinterval** <**int**>: sampling interval (in cycles) for LLC prefetch control (0 = no PF controller)
- **-LLC:pf:miss** <**bool**>: generate LLC prefetches only from miss traffic
- **-fsb:width** <**int**>: front-side bus width (in bytes)
- **-fsb:ddr** <**bool**>: front-side bus double-pumped data (DDR)
- **-fsb:speed** <**double**>: front-side bus speed (MHz)
- **-cpu:speed** <**double**>: CPU speed (MHz)
- **-MC** <**string**>: memory controller configuration string
- **-dram** <**string**>: DRAM/main memory configuration string

Some of the cache/TLB-related command-line options are composed of multiple sub-options combined into a single string; these are detailed below. Many of the Zesto Components also use similarly formatted parameters, but these are too many to list here; please refer to the corresponding source files for explanations of the pertinent parameters.

-il1 A:B:C:D:E:F:G:H:I

A: Name of the cache, e.g., "IL1", used as a prefix in stats output

B: Number of sets

C: Set associativity

D: Cache line size (bytes)

E: Number of banks

F: Bank width (bytes), used for determining interleaving, should be > line size

G: Latency (cycles)

H: Replacement policy (see zesto-cache.cpp:cache_create for options)

I: Number of MSHR entries

-dl1, -dl2, -LLC A:B:C:D:E:F:G:H:I:J:K:L:M

A: Name of the cache

B: Number of sets

C: Set associativity

D: Cache line size (bytes)

E: Number of banks

F: Bank width (bytes)

G: Latency (cycles)

H: Replacement policy

I: Allocation policy (i.e., allocate-on-miss vs. allocate-on-write)

J: Write policy (i.e., write-back vs. write-through)

K: Number of MSHR entries

L: Number of writeback-buffer/victim cache entries

M: Writeback buffer uses write combining?

-itlb, -dtlb, dtlb2 A:B:C:D:E:F:G

A: Name of the TLB

B: Number of sets

C: Set associativity

D: Number of banks

E: Latency (cycles)

F: Replacement policy

G: Number of MSHR entries

The DL1, DL2, and LLC have additional options for specifying the scheduling of MSHR requests (-dl1:mshr_cmd and -LLC:mshr_cmd). The option "FCFS" can be given under which MSHR requests are simply scheduled in a first-come first-serve fashion. Otherwise, a four-character string can be specified to explicitly prioritize request types. The four characters are 'R' (reads/loads), 'W' (writes/stores), 'P' (prefetches), and 'B' (writeBacks for dirty lines); all four characters must be included in the configuration string. An example string of "RWPB" indicates that if given a choice, any outstanding reads will first be serviced, then followed by writes, prefetches and writeback requests in that order. Among each priority class, requests are still scheduled in a first-come first-serve fashion (e.g., if two reads/loads are both candidates for sending to the next-level cache, whichever one was enqueued in the MSHR first gets priority).

B Supporting Simulators

This section briefly describes the other simulators that are provided.

B.1 sim-fast

This is effectively the same as the the original sim-fast. This is a functional simulator with no timing/cycle counting. Each macro-op is executed as a single instruction (i.e., no uop decomposition). The SimPoint-compatible basic block vector profiling code is enabled to generate .bb files [2]. Note that the PIN-related code has been removed. This was simply done early on to get the Zesto simulation infrastructure up and running as quick as possible. The PIN support should be replaced at some point in the future.

B.2 sim-eio

This is similar to sim-fast, but you should use this to generate EIO trace files once you have used the original sim-fast to generate the corresponding .bb files and have used SimPoint to determine the locations of your samples.

B.3 sim-uop

This is another functional simulator, but this operates at the level of the uops. Each macro-op gets decoded into its constituent uop flow, and then each uop gets individually executed. Similar to sim-fast, there is no timing or cycle counting here. The functional simulation still only occurs at the level of x86 macro-operations (i.e., the uops or *not* functionally executed), and so many of the uop fields do not actually get populated during the main execution loop. If you want to make use of any of these, make sure you double check that the fields do in fact contain valid values.

B.4 sim-bpred

This simulator is derived from sim-fast. It performs the same basic functional simulation, but control transfer instructions cause the branch predictor lookups and updates to be simulated. This again is an in-order, timing-less simulation, so the predictor simulation behaves in a strict alternating predict-then-update manner

(as opposed to a real processor where predict and update are decoupled causing the update to occur many cycles after the initial and multiple subsequent (and possibly wrong-path) prediction lookups).

C Differences Between the STM and DPM Models

The default Detailed Pipeline Model (DPM) a very high level of detail for simulating out-of-order pipelines (although it is of course not perfect). The Simple Timing Model (STM) is a simpler model that does not fully model all of the microarchitectural nuances handled by DPM, but it is also faster. If you simply invoke "./sim-zesto" with no arguments or benchmark, then the simulator will provide a dump of all of the available command line options ("knobs"). Most knob descriptions are appended with a designation in square brackets where 'D' indicates that the knob is used for the DPM model and 'S' for the STM model. That is, if a knob is listed with "[D]" and you're running the STM model, the knob value will be ignored. The current version of the STM model was directly derived from the DPM model, and may be further simplified.

D TODO

There are still several tasks to do, but this initial release can at least let people start playing around with the simulator. Some of these tasks include:

- Pipeline visualization
- Support for multi-threaded apps (implement fork, join, et al.)
- Add support for x86-64, MMX/SSE, etc.
- Debug syscalls to get the rest of SPEC2006 running.
- Add a simple in-order (maybe superscalar) pipeline model.

References

[1] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro Magazine*, 26(4):52–60, July–August 2006.

[2] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, Madison, WI, USA, June 2005.

[3] Ibrahim Hur and Calvin Lin. Adaptive History-Based Memory Schedulers. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 343–354, Portland, OR, USA, December 2004.

[4] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, AZ, USA, November 2001.

[5] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA, April 2009.

[6] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 128–138, Vancouver, Canada, June 2000.

[7] Jim E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, Minneapolis, MN, USA, May 1981.

[8] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 42–53, Atlanta, GA, USA, June 1999.

[9] Matt T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 23–34, San Jose, CA, USA, April 2007.