Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration

Gabriel H. Loh, Samantika Subramaniam, Yuejian Xie
Georgia Institute of Technology
College of Computing
{loh,samantik,corvarx}@cc.gatech.edu

Abstract

For academic computer architecture research, a large number of publicly available simulators make use of relatively simple abstractions for the microarchitecture of the processor pipeline. For some types of studies, such as those for multi-core cache coherence designs, a simple pipeline model may suffice. For detailed microarchitecture research, such as those that are sensitive to the exact behavior of out-of-order scheduling, ALU and bypass network contention, and resource management (e.g., RS and ROB entries), an over-simplified model is not representative of modern processor organizations. We present a new timing simulator that models a modern x86 microarchitecture at a very low level, including out-of-order scheduling and execution that much more closely mirrors current implementations, a detailed cache/memory hierarchy, as well as many x86-specific microarchitecture features (e.g., simple vs. complex decoders, micro-op decomposition and fusion, microcode lookup overhead for long/complex x86 instructions).

1. Introduction

Simulators are indispensable tools for computer architects, and as such many different simulators have been developed filling many different research and development needs. A variety of detailed, cycle-level models have been developed over the years, but in most cases the hardware organization assumptions are far too simplistic to be able to represent contemporary high-performance processor microarchitectures. For example, the highly popular SimpleScalar 3.0 toolset [1] makes use of out-of-order scheduling logic derived from the *Register Update Unit* (RUU) approach which is now almost two decades old! In other situations, the target or supported ISA(s) of the simulators simply do not expose some of the microarchitectural complexities associated with main-stream CISC (x86) architectures.

In this paper, we introduce a new simulator model called Zesto[1], which provides very detailed cycle-level simulation of a processor microarchitecture similar to contemporary high-performance x86 processors. Zesto was primarily developed to provide a tool to support research in low-level microarchitecture design for the x86 ISA.

[1]Zesto used to be a U.S. national fast-food franchise known for very rich and heavy ice cream. We chose this name because our simulator is very rich (highly detailed) and also very heavy (slow).

2. Related Work (Why Another Simulator?)

There are already a very large number of processor and system simulators available to the community, and so before presenting any new tool, we must first address the question of why this tool is even necessary. We will briefly summarize some of the most relevant existing tools, and we will point out what features are missing or are unnecessary for the types of low-level microarchitectural simulations that our Zesto simulator targets. We are in no way positioning our tool as "better" than any of these other existing tools; for different types of research studies as well as educational objectives, different tools will be appropriate.

2.1. The SimpleScalar Family

The SimpleScalar Toolset is a hugely popular set of simulation tools that have now been available to the public for well over a decade. Hundreds of academic papers have been published where the results were produced with SimpleScalar or SimpleScalar-derived simulators.[2] SimpleScalar, as distributed however, was never meant to be an actual *simulator*; it is a *simulator* ***toolset***. The popular out-of-order timing simulator (sim-outorder) was only meant as an *example* of how one could build a simple cycle-level simulator using the toolset. Sim-outorder was not meant as a reference simulator to be used by the masses. The ease-of-use of the simulator, however, made sim-outorder an almost de facto standard for simulation-based research studies for many years.

One of the strengths, and also a great weakness, of the sim-outorder timing simulator is that it is in fact very simple. The model allows graduate and undergraduate students to quickly get their hands dirty and perform some interesting explorations of out-of-order processors. The simplicity, however, also hides away microarchitectural details so much that many students end up believing that the sim-outorder model is more or less how modern processors really work. The sim-outorder scheduling logic is based off of the nearly two-decade-old Register Update Unit (RUU) organization [24], which treats the reorder buffer and issue queue/reservation stations as a single monolithic block. Other simplifications include combining decode, branch resolution, predictor updates and structural allocation all into the same stage, and the scheduling *and* execution form

[2]This statistic is based off of the "Who's using it?" webpage from the SimpleScalar website which was last updated in 2003 (in that year alone they have listed 110 publications using their tools).

a single hardware loop; the list goes on, but there is no need to belabor the point as this is not really the fault of the SimpleScalar Toolset (they cannot really take the blame for other researchers choosing to use an out-dated model that was primarily intended to be an example simulator).

Since 2001, SimpleScalar has also provided a version 4.0 "pre-release" that includes their MASE timing simulator [11]. The MASE simulator adds more detail to a few critical areas of the processor, such as using a separate reorder buffer and scheduler organization as well as some support for accounting for load-latency misscheduling (replays). MASE also provided a better overall simulator organization employing an oracle functional simulator that executes at fetch (as opposed to at decode in sim-outorder) thereby being able to support simulations with true oracle control-flow information. Recently, an x86 version of SimpleScalar was made available to a limited number of "early testers," and unfortunately it seems that MASE was not extended to support the x86 ISA. The provided x86 timing simulator was simply sim-outorder, which we view as a step backward from MASE. This provided our initial motivation to develop a new out-of-order processor simulator for **x86** that provides **detailed** modeling of a modern x86-based microarchitecture.

2.2. PTLsim

More recently, the State University of New York at Binghamton has released the PTLsim simulator which is a detailed x86 timing simulator that models a much more modern microarchitecture [27]. The simulator uses a variety of code caching techniques, native execution, and even hand-optimized SSE assembly code to provide very high levels of simulation performance (i.e., simulated instructions per second). While PTLsim provides a much greater level of microarchitectural detail than the earlier SimpleScalar models, there are still a variety of areas where our Zesto timing model is even more detailed. Its highly-optimized coding style provides relatively high simulation speeds (including performance critical sections being hand-coded and optimized using inline SSE assembly instructions), which in turn makes the code far more difficult to modify.

2.3. PinTools

While not a simulator per se, the Pin dynamic instrumentation tools [13] provide a way to build simulator-like tools that run on native x86 binaries. Pin provides a powerful tool for a wide variety of computer architecture studies, but we chose not to make use of it because Pin does not provide a natural way to model the effects of wrong-path (branch misprediction induced) instruction behaviors. For the purposes of building a cycle-level model, the PinTool would have to be invoked on every instruction, but since Pin interacts with a program actually executing on the native hardware, wrong-path instructions, which are by definition invisible or even non-existent beyond the ISA interface, cannot be observed. For detailed modeling of modern x86 processors, we also desired to be able to decompose x86 instructions into RISC-like micro-operations, also known as *micro-ops* or *μops*. Since μops are invisible above the ISA boundary, Pin would not be able to provide this information, which would basically force the PinTool to perform a significant amount of decode work anyway, thereby losing a significant amount of the speed benefit from native execution.

2.4. Other Tools

Due to our objective of performing x86-centric microarchitecture research, many of the other excellent simulation frameworks/environments available to the community were not considered due to their lack of support for the x86 ISA. These include SESC [19], GEMS [14], M5 [2], RSIM [9], Liberty [25], Microlib [18], SimFlex [23] and others. While some of these projects report on-going development to add support for x86, such support is not yet available. Again, we emphasize that these are actually very good tools for studying the types of research problems that they were designed for, but the hardware models and assumptions on the pipeline organization did not fit our needs for highly-detailed x86 microarchitecture modeling and exploration.

3. Overall Organization

The high-level organization of the simulator consists of an oracle "execute-at-fetch" functional simulator coupled with the detailed timing simulator written in a mix of C and C++. Figure 1(a) shows this organization. Due to the detailed modeling of many microarchitectural components (described in Section 4), the speed of the simulator is fairly slow (10's of KIPS) compared to some other simulators (MIPS in some case). While those accustomed to academic simulators may balk at this relatively low simulation speed, this rate is comparable to detailed industrial simulators. Due to Zesto's low simulation speed, we did not include true "execute-at-execute" simulation such as that found in MASE [11] and M5 [2]. The organization of the code, however, is structured to allow for easy extension to include execute-at-execute functionality if desired or necessary, for example, for the correct simulation of multiprocessor memory ordering [16].

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

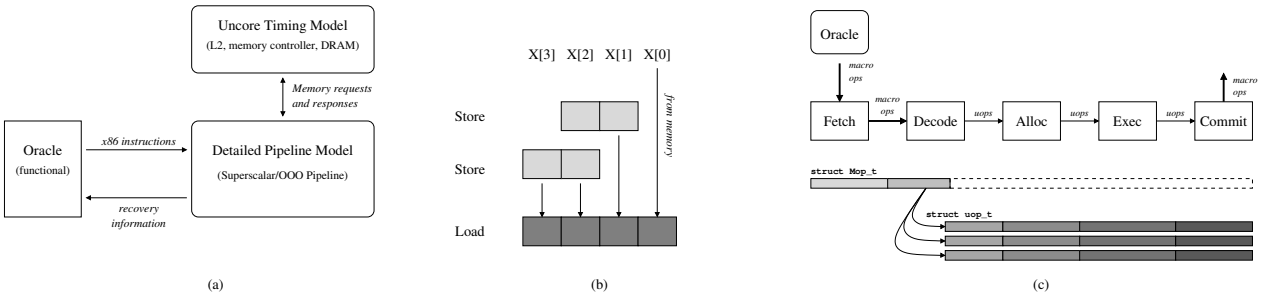

Figure 1. (a) Overall simulator organization, (b) example memory read operation requiring multiple sources, (c) high-level pipeline organization and corresponding macro-op and micro-op data structures.

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 exact interaction of the oracle with the pipeline will be further elaborated later when we discuss the fetch engine in Section 4.

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.

The internal data structures for the x86 macro- and micro-ops, schematically illustrated below the pipeline components, are organized to reflect the pipeline organization. All data associated with a given block are grouped together in individual structures (indicated by different levels of shading). While this increases the verbosity of the actual simulator code (e.g., "uop→decode.is_load" vs. "uop→is_load"), it forces the simulator programmer to always be aware of where data are coming from in the processor. For example, code in the fetch block should not make use of information from the decode stage because in a real pipeline, the instruction has not yet even entered the decode stage, let alone been decoded. By partitioning the information into block-specific structures, any "unrealistic" use of data is explicitly declared in the code (e.g., simply running a grep of "decode" on the fetch block will show the cases where the programmer "cheated"). In some cases, accessing data earlier than is possible may be desirable (e.g., limit studies), and in other cases it may simply be tolerated (e.g., the programmer decides to assume that the BTB can provide some pre-decode information), but the organization forces the programmer to acknowledge the violation.

4. Zesto Microarchitectural Model

The five primary pipeline blocks described in the previous section are only those blocks associated with the main processor pipeline. Many other typical blocks are simply instantiated within these existing blocks (e.g., L1 caches), and all remaining structures outside of a typical core are captured within an "uncore" block. In this section, we will ex-

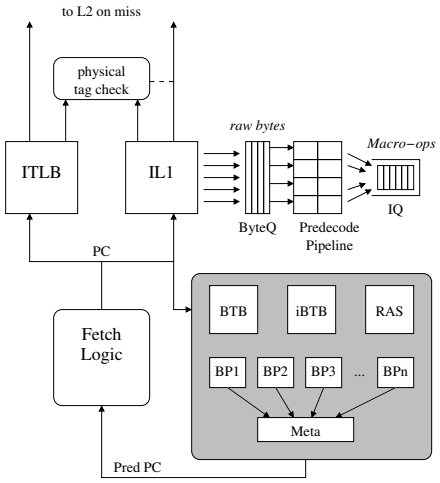

Figure 2. Block-diagram of the Zesto fetch engine components.

plain the details of the models for each of the blocks, focusing specifically on areas where our model provides more detail than most other models, or where there are x86-specific microarchitectural features.

4.1. Fetch

The fetch engine is the entry point into the processor, shown in Figure 2. At the start of each cycle, the fetch engine attempts to make a single cacheline's worth of instructions. Starting from the current PC (program counter), the fetch engine requests macro-ops from the oracle. The fetch engine will continue to do so until fetch needs to go to a different cache line. Note that during this process, the fetch engine has no interaction with the instruction cache, but when this process finishes, all instructions associated with the cache line will have been executed by the oracle. The fetch engine then makes a *single* request to the instruction cache and the ITLB for this one cache line. Contrast this against most other academic simulators where a separate IL1 request is generated for every instruction. A research study for reducing IL1 energy consumption that uses a simulator with per-instruction IL1 access modeling will overinflate the number of observed cache accesses and therefore likely report inflated power savings.

Due to x86's variable-length instruction encoding, some instructions may span across two cache lines. In such cases, the oracle "executes" the instruction along with the first cache line, but the fetch engine must actually issue a second request to the IL1 for the additional cache line (which cannot happen in the same cycle as the original fetch). Any bytes associated with the "split" instruction must wait in the instruction byte queue (byteQ) until the remaining bytes have arrived. Only then can the macro-op proceed into the predecode pipeline. Note that the instruction byte queue is simply an array of raw, undecoded bytes directly fetched from the instruction cache. At this point, the modeled processor does not even "know" where the instructions are (i.e., that requires instruction length pre-decoding which happens later).

Another interesting aspect of the Zesto fetch model is how it handles the interaction between the IL1 and the ITLB. In many simulated processor models, the IL1 and ITLB are effectively two separate and independent cache structures. The fetch latency of an instruction is simply computed as the maximum latency between the two. While the access to both IL1 and ITLB can be started in parallel, in a virtually-indexed, *physically*-tagged IL1, the IL1 tag check cannot happen until the TLB translation has completed. In the case of an ITLB miss, the Zesto fetch model stalls the completion of the IL1 access until the ITLB translation has returned from the lower levels of the cache hierarchy.

From the instruction byte queue, the bytes then enter a predecode pipeline whose job is to perform the basic instruction length decoding. This pipeline stage effectively scans the raw bytes in the instruction byte queue to perform just enough decoding to figure out where instructions start and end. Note that the simulator is not actually performing the work of real instruction length decoding because the oracle has effectively done all of the necessary decoding work, but this models the timing and bandwidth constraints that such a piece of hardware would impose on the delivery of instructions to the rest of the pipeline. At the end of this predecode pipeline (which may be a single stage), the resulting instructions (macro-ops) are placed in an instruction queue (IQ) with one x86 macro-op per entry.

The fetch block is also home to the branch predictor, which includes direction predictors (multiple to support arbitrary hybrid predictors), BTBs (plural because a separate indirect-jump BTB may be instantiated), and the return address stack. The user can configure any number of component direction predictors, and then specify a single meta-predictor that combines the predictions together. The individual components can be any of the supported algorithms (the object-oriented organization of the predictor modules makes adding new algorithms to the simulator very simple), and the meta-predictor can implement a variety of different techniques such as classical tournament-based selection [17], multi-hybrid selection [5], or fusion techniques [4, 12].

4.2. Decode

The decode block accepts x86 macro-ops and converts them into RISC-like μops to be consumed by the remainder of the pipeline. Figure 3 illustrates the components in the Zesto decode process. Multiple macro-ops can enter the decode pipeline per cycle. From a simulation perspective, this pipeline is used primarily to model the latency of the decode

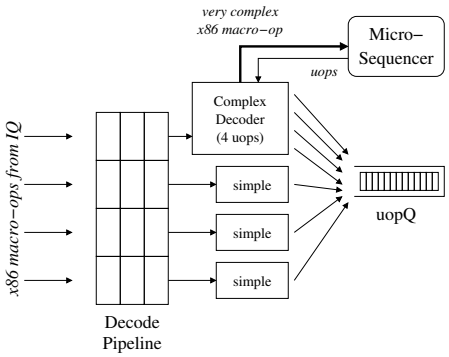

Figure 3. Block-diagram of the Zesto decode pipeline components.

process. Furthermore, the pipeline is "non-compressing" in that a stage must be completely empty before instructions from the previous stage may advance (all or nothing). This is more practical for actual processor implementation because otherwise every pair of pipeline stages would need logic to detect how many slots will be freed up, how many instructions can move forward, and to which slots they will move forward to.

At the end of the pipeline, we crack the macro-ops into μops subject to a variety of decoding constraints. Conceptually, each decode lane has a decoder with a different level of decoding capability. For example, in the original Intel Pentium-Pro, the three-wide "4-1-1" decoder consisted of a complex decoder that could handle most macro-ops so long as they decompose into four μops or less, and the remaining two decoders can only handle macro-ops that convert directly to one μop each. Zesto models this type of constraint by associating each lane with a μop limit; any macro-op that ends up at a decoder with insufficient decode capability stalls until an appropriate decoder is available. Zesto also supports μop fusion [6], and so when this feature is enabled, a fused pair of μops only counts toward a decoder's capacity limit as a single μop. Macro-op fusion is currently not supported [10]. The μops are then placed in a μop queue (μopQ).

Some x86 instructions decode in 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. The x86 ISA also supports repeating ("REP") instructions, where a REP prefix byte causes the instruction to be executed multiple times until an index register satisfies some condition (e.g., is equal to zero). This is used for, among other things, very compact single-instruction implementations of memory move or copy functions. In these cases, Zesto injects additional μops to implement the control flow associated with the REP. For example, a "micro-jump" is inserted before the execution of any other μops to test for the situation where a REP instruction is invoked for zero iterations. At the end of each iteration, an additional μop is also included to update the index register followed by another micro-jump μop to test whether the REP has completed. Since REP instruction are actually interruptible operations, Zesto represents each iteration as a separate macro-op data structure, but tracks statistics as if all iterations formed a single, giant x86 operation.

Much of the microarchitectural modeling of Zesto's decode block deals with x86 specific "features" and therefore are simply not supported by most other simulation models.

4.3. Allocation

The allocation stage handles the assignment of hardware resources to the individual μops. The default Zesto model assumes a Pentium/Core-style physical register organization where each physical register is bound to a corresponding reorder buffer (ROB) entry. Modeling a separate physical register file (or separate integer vs. floating) would be simple but is not currently implemented. Load instructions need to wait for a load queue (LDQ) entry to be available, and similarly for store instructions and the store queue (STQ). All μops require a ROB entry, and all non-NOPs require a reservation station (RS) entry. If any required resource is not available, allocation stalls. Each fused μop occupies only a single RS entry and a single ROB entry.

The allocation stage also performs functional unit assignments for the μops. Execution resources are divided into groups, called dispatch ports, execution ports, or simply ports [8]. Certain types of functional units may be replicated across multiple ports; for example in Figure 4, simple integer execution units (IEU) are available on ports 0, 1 and 5. At allocation time, each μop is bound to a single execution port. After entering the execution stage (described below), the μop will only bid and issue on its designated port. Naturally, μops will only be assigned to ports where an appropriate functional unit exists (e.g., a load would never get bound to port 4 in our example). Our current implementation uses a simple load-balancing heuristic to assign μops to ports [22].

4.4. Execution

The execution block (Figure 4) 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

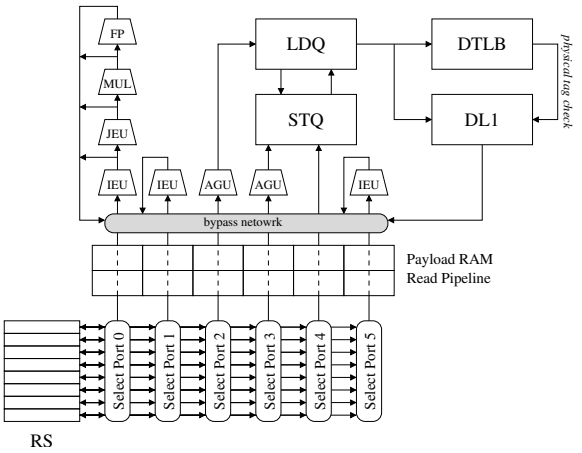

Figure 4. Block-diagram of the execution logic.

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.

Zesto's execution model implements true, decoupled speculative scheduling. The wakeup-select loop causes a μop to broadcast its tag to its dependents, but after this the μop must first access the payload RAM to read its information (e.g., data operands, opcode, etc.) before proceeding to the actual functional units. During the payload RAM read latency (which is explicitly modeled as separate pipeline stages), the issued μop's dependents will continue to be speculatively scheduled; that is, the scheduling loop (wakeup-select) is completely decoupled from the actual execution loop. This means that even if the μop was mis-scheduled, the actual issue resources have now been consumed. Contrast this to the scheduling implementation in the MASE simulator where their instruction mis-scheduling (e.g., due to a cache miss) causes extra delay to be modeled, but the additional resource utilization is not properly accounted for.

When a μop finally reaches the end of the payload RAM stage(s), it checks to see if all of its input operands are available and valid. If they are, then the μop simply proceeds to execute and it deallocates its RS entry (for fused μops, the last μop to issue in the fused pair is responsible for the RS entry deallocation). If one or more inputs is not actually ready, for example if the μop was scheduled assuming that a parent load would hit in the cache but it turned out to be a miss, then a scheduling replay is required. In this case, the operation at the ALU is invalidated, the status of the μop's RS entry is reset, and the μop may be rescheduled for the future (e.g., reschedule the μop assuming its parent load now hits in the L2 cache). Since the μop may have had some of its dependents also speculatively issued, these μops also need to be "snatched back." We assume that this snatch back process can occur in a single cycle, which is reasonable for Intel Core-like microarchitectures. For very aggressively pipelined microarchitectures like the Intel Pentium 4, RS entries are deallocated on issue and so mis-scheduled μops would have to be diverted to a separate buffer and then re-allocated back into the RS [8]. The current Zesto model does not support this, but we feel that this is reasonable given industry's trend away from such super-pipelined designs.

At the end of execution, the μop broadcasts its results to its dependents and then updates its status in the ROB. Note that prior to result broadcast, the bypass bus must be available. We assume one bypass bus per execution port. In the normal situation where one μop is issued to a port per cycle, then only one μop will complete execution on that port per cycle, and so the single bypass path per port will suffice. In the case of multi-cycle instructions, however, more than one μop on the same execution port (but executing on different physical functional units) may complete at the same time. In this case, only one may proceed and the other will have to stall and try again the next cycle.[3] This in turn may lead to additional mis-schedulings of the stalled μop's dependent instructions.

Load Instruction Execution

Zesto uses an Intel Pentium/Core-like approach for the handling of load and store instructions. Each load is represented by a single load μop. After allocation, the load μop is inserted into the RS, and it also has an associated entry in the LDQ. When the load μop issues from the RS, it makes its way through the payload RAM pipeline as usual, and then executes on an address generation unit (AGU) before writing the computed address into its LDQ entry. Note that

[3] For multi-cycle instructions with known latencies (e.g., integer multiply), one could implement a scheduling algorithm that also pre-scheduled bypass usage, but the current implementation in Zesto does not do this.

in the execution port organization shown in Figure 4, ports 2, 3 and 4 each only support a single operation (load, store address and store data, respectively). This reflects an optimized datapath implementation where these three ports do not actually have to have full bypass networks because their results will be written directly to the corresponding load or store queue entries in a direct-mapped fashion.

From the LDQ, a load μop may issue depending on the speculation policy. In a conservative configuration, the load will wait until all younger store addresses have been resolved so that store-load ordering violations can be guaranteed. In more aggressive configurations, Zesto also supports speculative memory disambiguation where a predictor can decide whether or not the load issues from the LDQ to the data cache. When a load does issue, it starts three accesses in parallel: one for the data cache, one for the DTLB, and one to search the STQ for store-to-load forwarding. Similar to the IL1 and ITLB, Zesto assumes that the DL1 is virtually-indexed and physically tagged, and so the DL1 access cannot complete until the virtual-to-physical translation has been performed. A miss causes the request to be enqueued in a MSHR entry which will then be scheduled to proceed to the L2 cache subject to bus and L2 port availability. We will not go into detail about the Zesto cache hierarchy, but we make use of a callback-based approach (similar to several other simulators) and model finite-capacity MSHRs, bus contention, cache port contention on a per-bank basis, and multiple hardware prefetchers. Eventually, a hit either occurs in one of the cache levels, or the request is sent to the memory controller. Split-line DL1 accesses are handled in a manner similar to the IL1 where two cache requests must be made and the load is not considered as completed until both portions have returned their data.

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 [26]. 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. 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.

Store Instruction Execution

In the RS, store instructions are somewhat similar to loads, but they are decomposed into two separate μops: a store address (STA) μop and a store data (STD) μop (although if fused they may share a single RS entry). From a scheduling perspective, the STA and STD μops are completely independent and can issue in any order depending on the arrival of their input tags. The STA μop also issues to an AGU and then places the computed address in the corresponding STQ entry. The STD issues to a STD port that effectively computes nothing; the only purpose is to forward the data operand to the corresponding STQ entry. One may wonder what the purpose of this extra step is since no actual useful computations are being performed; for example, the SimpleScalar 3.0 and 4.0 models allocate the STD operation directly into the STQ entry where it waits for its data operand to arrive. The problem with the "SimpleScalar" approach[4] is that the implementation implications of having the STD operation wait directly in the STQ is that the result bypass network must now be extended to the STQ as well. This would require adding CAM ports to the STQ to support the data capture of the STD operation's input value from the bypass network. The approach used in the Zesto model corresponds to a hardware implementation where the existing CAM logic in the RS entries and payload/bypass network are reused to capture the STD's input, and then from there the value can be directly written into the STQ entry with a lower-overhead direct-mapped RAM port.

For store operations, the STQ entry simply waits until both STA and STD results have arrived. At this point, the STQ searches forward in the LDQ to detect any speculatively mis-ordered load operations. Detecting address matches between stores and loads in x86 is somewhat tricky because of the different memory operation sizes and the lack of alignment restrictions. Consider again the loads and

[4] We are not trying to pick on SimpleScalar; this is a modeling inaccuracy in many timing simulators, but we use SimpleScalar as an example due to its familiarity for many researchers.

stores from Figure 1(b) which exhibit a variety of sizes and alignments. Some stores may overwrite some bytes of other stores (but not all), and the matching logic is further complicated by the fact that word-level address comparisons do not work and so per-byte matching must be performed. This is one of the examples where Zesto carefully models the detailed operation of the processor to support x86 features, but one can easily see how this type of logic can greatly increase simulation overhead. Beyond this, the store operations do not perform any other functions but simply wait until commit to write their values back to the cache.

4.5. Commit

The instruction commit process makes instruction results globally visible to the architected state and deallocates related hardware resources. The x86 ISA, and in particular modern μop-based pipelines, add some complications because commit must appear to the outside world as if the x86 operation executed atomically. That is, the external world should never be able to observe a processor state (i.e., the pipeline takes an interrupt) where part of a macro-op's μops have been committed while others have not. This constraint makes Zesto's commit process more "bursty" as all μops must wait for all other μops within the same flow to complete execution before *any* of them can commence with the commit process. Once the commit process starts for even a single μop in the flow, then all μops must be committed. Such commit behaviors, however, could have significant impact (positive or negative) on research proposals regarding aggressive commit techniques such as early resource deallocation [15].

Store instructions write their values back at commit as well. Zesto makes use of the "senior store queue" organization where a portion of the STQ implements a virtual queue of stores that have committed but not yet completed writing back to memory. The alternative is to implement a separate writeback buffer, but such a buffer would also have to be searched by loads when they execute. Instead of implementing another complex piece of address-matching and data-forwarding logic, the idea of the senior STQ is to reuse the existing associative logic in the STQ to fulfill this need. At commit, store μops release their ROB entries and enter the senior STQ. Physically, entering the senior STQ does not actually involve moving the contents of the STQ entry anywhere; only a simple hardware pointer (STQ_head) needs to be updated. Stores in the senior STQ then attempt to issue to the DL1 and DTLB (with split-line accesses requiring two separate DL1 writes); the store writeback eventually finishes, and the corresponding senior STQ entry is marked as completed. The STQ entries corresponding to this senior STQ are deallocated in order; STQ entries cannot be reallocated to new stores until they have been removed from the senior STQ. Modeling of such behavior increases store queue capacity pressure since the store μops occupy STQ entries for some number of cycles beyond the conventional commit point.

4.6. Uncore

In addition to the main processor pipeline, Zesto also models several "uncore" components (those parts not belonging to the processor cores). In Zesto, the uncore includes the unified L2 cache (including the bus from the cores to the L2), L2 hardware prefetchers, the memory controller, and the DRAM. The L2 cache behaves just like the other cache levels, and so we will not elaborate on it further here. L2 misses issue from the L2 MSHR entries to the memory controller. The memory controller accepts the requests and inserts them in a transaction queue. Zesto supports a memory controller model that attempts to schedule accesses to maximize row buffer hits [20]. The memory controller then issues the requests to the main memory model, which can be configurable as well. One can simply choose to instantiate a simple constant-latency model, but we have also implemented a simple SDRAM timing model that accounts for memory banking, row buffer hits, some of the critical memory timing constraints (e.g., precharge delays, write-to-read delays, etc.), DDR transfers, and periodic refresh.

5. Implementation and Validation

Zesto was implemented on top of the pre-release version of the x86 toolset. We started with the functional μop-level simulator as the basis for our oracle functional execution engine. As such, we also inherit many of the limitations of the original functional simulator, most importantly being the lack of support for 64-bit code and SIMD extension instructions (e.g., MMX/SSE). We made many extensive changes to the μop format. For example, the original format allowed each μop to consume up to four inputs (not including flags), and generate up to three outputs. This did not conform with conventional notions of a "RISC-like" μop. We modified the μop format to have up to three inputs (needed to support some of the fused μop formats) and generate only a single register output. This required rewriting several of the μop flows to use longer sequences of these more RISC-like μops. To handle partial register writes (for example where one μop performs a 16-bit register modification and then a subsequent μop reads the 32-bit version of the register), all byte and word (16-bit) register modifications require an extra μop to be injected which takes the low-width result as one operand, the previous 32-bit result as the second operand (requires careful handling of input dependencies), and then generates a merged 32-bit result so that all subsequent reads do not have to worry about merging register values from multiple instructions. On top of this functional simulator, we then constructed all of the remaining timing models.

The complete validation of Zesto is still an on-going project, but it has reached a state where we have enough

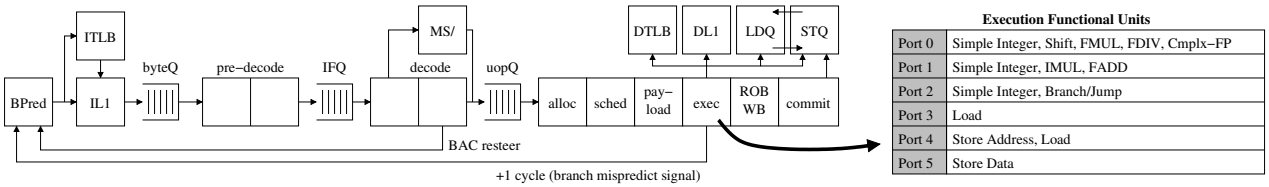

Figure 5. Simulated pipeline organization for the processor model used in our microbenchmark validation experiments.

Fetch			
IL1	32KB, 8-way	ITLB	128-entry, 4-way
BPred	TAGE [21] 4KB	BTB	2K-entry, 4-way
iBTB	512-entry, 4-way	RAS	16-entry
byteQ	3 × 16-byte entries	IQ	18 entries
Decode			
uopQ	24 entries	Decoders	4-1-1-1 with fusion
Out-of-Order			
RS	32 fused entries	LDQ	32 entries
ROB	96 fused entries	STQ	20 entries
Cache/Memory			
DL1	32KB, 8-way, 2-cycle (+1 for AGU)		
DTLB0	16 entries, 4-way	DTLB1	256 entries, 4-way
Mem. Cntlr.	16-entry request queue	DRAM	DDR2-1066/1333

Table 1. Our Intel Core/Merom-like configuration.

confidence in the model to perform research experiments with it. Due to its detailed timing model, there are many parts that would cause the model to deadlock if not implemented correctly. This is particularly true for the components related to speculative scheduling and replay handling. The low-level detailed modeling simply forces us to write many components correctly by construction. In addition, we have also used conventional cycle-by-cycle pipeline event logging to carefully observe the behavior of the processor microarchitecture. We are currently developing a graphical pipeline log viewer that will ultimately be released with the rest of the simulator.

At a recent workshop panel, several researchers from industry argued the importance of cycle-level models and the general importance of validating academic simulation models against real hardware [3]. The consensus was that validation of academic simulators against real hardware is not necessary so long as the proper research *insights* can be provided by the study. Nevertheless, we were not entirely comfortable with using our simulator in a complete "open loop" mode, and so we have made some preliminary efforts to validate our implementation. We configured our simulator to model an Intel Core 2 (Merom) microarchitecture, ran a variety of hand-written microbenchmarks through the Zesto simulator (to completion), and then also ran the same exact unmodified binaries on the native hardware.[5] Thus far, we have only used wall-clock timing measurements, but future work will include more detailed performance-counter based comparisons as well. Figure 5 shows the overall pipeline organization with execution port configurations (individual structures such as the ROB are not shown) and Table 1 lists the baseline processor configuration. We ran the microbenchmarks on three different Intel Core 2 systems with different CPU and memory clock speeds.

Table 2 lists the microbenchmarks used along with the simulated and measured performance results and instruction counts. Each microbenchmark consists of a few nested loops (with the inner most loop unrolled several times to reduce loop overhead) to repeatedly perform some simple tasks. With the exceptions of the dijkstra and tiffdither applications from the MiBench suite [7], these microbenchmarks are all effectively simple toy programs. For each microbenchmark, we ran the program five times in a row and report the average observed time spent executing user code (which is very close to total time since these microbenchmarks contain very few system calls). A comparison of the measured and simulated results shows that Zesto was able to provide reasonably accurate performance estimates for these microbenchmarks. This should not be taken as evidence that every aspect of the model has been perfectly accounted for, but the fact that we could get even a few microbenchmarks to match up with real observed performance across three different machines (albeit with the same pipeline microarchitecture) greatly increases our confidence that at least the majority of critical pipeline details have been modeled with sufficient accuracy. We do not realistically expect to be able to be fully accurate and faithful with respect to the real hardware as there are simply too many technical details where we must guess at the implementations (e.g., branch prediction algorithms, exact details of the cache hierarchy and MSHR architecture, prefetcher and prefetch throttling algorithms, cache replacement policies, and many others).

6. Zesto Multi-Core Support

Multi-core simulation was not one of the original design objectives for Zesto, but we have added a few features to support limited multi-core configurations. Due to the lack of implementation of critical system calls such as fork, join and other synchronization primitives, Zesto's multi-core support is currently limited to multi-programmed workloads rather than true cooperative multi-threaded applica-

[5]The simulator supports functional fast-forwarding to more quickly reach representative portions of the benchmarks, but this was not used in these validation experiments. All microbenchmarks were run to completion.

Microbenchmark Name	Short Description	Instructions		
		Measured	Simulated	|Err%|
bsearch	Binary search on a sorted array	2.321B	2.307B	0.60%
div	Division on an array of integers	2.764B	2.764B	0.00%
dl1	Sequential memory accesses on a small array	4.464B	4.463B	0.01%
fp	Mix of FP add/mul/div operations	2.181B	2.181B	0.00%
memory	Sequential memory accesses on a large array	4.968B	4.967B	0.00%
mul	Multiplication on an array of integers	4.010B	4.010B	0.00%
qsort	Quicksort on random integers	2.997B	2.936B	2.04%
dijkstra	Shortest path in graph (MiBench)	3.604B	3.604B	0.00%
tiffdither	Dither a TIFF image (MiBench)	1.035B	1.032B	0.29%
Average Absolute Error %				0.02%

Runtime (seconds)									
Microbenchmark Name	1.87Ghz CPU/DDR2-1066			2.67GHz CPU/DDR2-1333			3.0GHz CPU/DDR2-1333		
	Measured	Simulated	|Err%|	Measured	Simulated	|Err%|	Measured	Simulated	|Err%|
bsearch	1.363	1.331	2.34%	0.928	0.937	1.01%	0.817	0.832	1.85%
div	1.508	1.530	1.44%	1.048	1.071	2.23%	0.931	0.952	2.29%
dl1	1.516	1.651	8.92%	1.052	1.156	9.82%	0.938	1.027	9.55%
fp	1.406	1.430	1.72%	0.980	1.001	2.16%	0.866	0.889	2.68%
memory	1.825	1.981	8.53%	1.215	1.396	14.87%	1.081	1.241	14.82%
mul	2.051	2.210	7.73%	1.419	1.547	9.03%	1.253	1.375	9.75%
qsort	1.407	1.350	4.07%	0.969	0.953	1.67%	0.861	1.156	1.68%
dijkstra	1.524	1.566	2.77%	1.043	1.101	5.48%	0.935	0.979	4.71%
tiffdither	0.272	0.287	5.51%	0.191	0.201	5.06%	0.169	0.179	5.49%
Average Absolute Error %			4.78%			5.70%			5.87%

Table 2. Microbenchmarks used for preliminary validation. Performance reported in execution time (seconds).

tions. To simulate multiple cores, Zesto simply instantiates n identical cores and connects them all together to a single shared uncore. This implies a L2 cache shared among all of the cores.

Each core simply executes on its own, oblivious to the existence of the other cores except through indirect performance effects. In particular, the shared uncore is a point of contention, where the individual cores may experience additional delays due to bus contention when trying to access the L2. Other effects such as multi-application L2 cache sharing/thrashing may also impact the performance of individual cores.

Each core runs one application, each with its own corresponding oracle. The oracle executes all instructions using the virtual address space of the individual application. These virtual addresses, however, cannot be used throughout the rest of the multi-core system due to address aliasing issues. For example, if core 0 has a load that brings data for virtual address X into the L2 cache, a subsequent load from core 1 to core 1's address X should not result in a hit on core 0's version of X! To handle this, Zesto takes every virtual address, combines it with the core id, and then remaps this pair to a globally unique physical address. The current implementation assumes 4KB pages, and allocates the physical pages on a first-come-first-serve basis.

The current multi-core support for Zesto still needs more work. The current version of Zesto's multi-core support does not model cache coherence invalidations or the enforcement of memory consistency. Since the simulator only handles multi-programmed workloads, each core operates on a disjoint set of physical memory locations, and so coherence probes will *never* result in hits (i.e., invalidations) and similarly for probes into the cores' load queues for enforcing memory consistency.

7. Conclusions

Zesto is not a simulator meant to replace all other simulators. In fact, Zesto was designed to fill a very specific niche in the simulation spectrum. For those interested in pursuing detailed microarchitecture research for x86 pipelines, we believe that x86 is very good for this purpose. There are many types of simulation studies where Zesto is probably quite sub-optimal.

A secondary purpose for the Zesto framework is for education in advanced graduate-level computer architecture courses. The Zesto simulator has been used in such a course at Georgia Tech to provide students with a deeper understanding of the organization of modern high-performance, superscalar, out-of-order microprocessors. The Zesto simulator is available to the public at —.—.—.— (*web-address will be made available after official public release*) with an open license on all of our source code (the portions orig-

inally from SimpleScalar will remain under the terms of the original SimpleScalar license). We will also include all course materials from the associated course; hopefully the community will find both the simulator and the other teaching materials useful for research and education.

Acknowledgments

Funding and equipment were provided by a grant from Intel Corporation. Funding in part was also provided by NSF grants CCF-0702275 and CCF-0643500. Samantika Subramaniam was supported by an Intel fellowship.

References

[1] AUSTIN, T., LARSON, E., AND ERNST, D. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine* (February 2002), 59–67.

[2] BINKERT, N. L., DRESLINSKI, R. G., HSU, L. R., LIM, K. T., SAIDI, A. G., AND REINHARDT, S. K. The M5 Simulator: Modeling Networked Systems. *IEEE Micro Magazine 26*, 4 (July–August 2006), 52–60.

[3] BURGER, D., HILY, S., MCKEE, S., RANGANATHAN, P., AND WENISCH, T. Cycle-Accurate Simulators: Knowing When to Say When. Panel Session at the Sixth Annual Workshop on Duplicating, Deconstructing and Debunking, June 22 2008.

[4] DESMET, V., VANDIERENDONCK, H., AND BOSSCHERE, K. D. A 2bcgskew Predictor Fused by a Redundant History Skewed Perceptron Predictor. In *Proceedings of the 1st Championship Branch Prediction Competition* (Portland, OR, USA, December 2004), pp. 1–4.

[5] EVERS, M., CHANG, P.-Y., AND PATT, Y. N. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In *Proceedings of the 23rd International Symposium on Computer Architecture* (Philadelphia, PA, USA, May 1996), pp. 3–11.

[6] GOCHMAN, S., RONEN, R., ANATI, I., BERKOVITZ, A., KURTS, T., NAVEH, A., SAEED, A., SPERBER, Z., AND VALENTINE, R. C. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal 7*, 2 (May 2003).

[7] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: A Free, Commerically Representative Embedded Benchmark Suite. In *Proceedings of the 4th Workshop on Workload Characterization* (Austin, TX, USA, December 2001), pp. 83–94.

[8] HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYLER, A., AND ROUSSEL, P. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal* (Q1 2001).

[9] HUGHES, C. J., PAI, V. S., RANGANATHAN, P., AND ADVE, S. V. RSIM: Simulating Shared-Memory Multiprocessor with ILP Processors. *Computer 35*, 2 (2002), 40–49.

[10] INTEL CORPORATION. Introducing the 45nm Next Generation Intel Core Microarchitecture. *Technology@Intel Magazine 4*, 10 (May 2007).

[11] LARSON, E., CHATTERJEE, S., AND AUSTIN, T. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software* (Tucson, AZ, USA, November 2001), pp. 1–9.

[12] LOH, G. H., AND HENRY, D. S. Predicting Conditional Branches with Fusion-Based Hybrid Predictors. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques* (Charlottesville, VA, USA, September 2002).

[13] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, June 2005), pp. 190–200.

[14] MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News 33*, 4 (November 2005), 92–99.

[15] MARTÍNEZ, J., RENAU, J., HUANG, M. C., PRVULOVIC, M., AND TORRELLAS, J. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture* (Istanbul, Turkey, November 2002), pp. 3–14.

[16] MAUER, C. J., HILL, M. D., AND WOOD, D. A. Full-System Timing-First Simulation. In *Proceedings of the ACM SIGMETRICS* (Marina Del Rey, CA, USA, June 2002), pp. 108–116.

[17] MCFARLING, S. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[18] PEREZ, D. G., MOUCHARD, G., AND TEMAM, O. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *Proceedings of the 37th International Symposium on Microarchitecture* (Portland, OR, USA, December 2004), pp. 43–54.

[19] RENUA, J., FRAGUELA, B., TUCK, J., LIN, W., PRVULOVIC, M., CEZE, L., SARANGU, S., SACK, P., STRAUSS, K., AND MONTESINOS, P. SESC simulator, January 2005. http://sesc.sourceforge.net.

[20] RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture* (Vancouver, Canada, June 2000), pp. 128–138.

[21] SEZNEC, A., AND MICHAUD, P. A Case for (Partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction Level Parallelism 8* (2006), 1–23.

[22] SHEN, J. P., AND LIPASTI, M. H. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw Hill, 2005.

[23] SIMFLEX: A FAST, ACCURATE, F. F.-S. S. F. F. P. E. O. S. A. Nikolaos Hardavellas and Stephen Somogyi and Thomas F. Wenisch and Roland E. Wunderlich and Shelley Chen and Jangwoo Kim and Babak Falsafi and James C. Hoe and Andreas G. Nowatzyk. In *Proceedings of the ACM SIGMETRICS* (New York, NY, USA, June 2004), pp. 31–35.

[24] SOHI, G. S. Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers 39*, 3 (March 1990), 349–359.

[25] VACHHARAJANI, M., VACHHARAJANI, N., PERRY, D. A., BLOME, J. A., AND AUGUST, D. I. Microarchitectural Exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture* (Istanbul, Turkey, November 2002), pp. 271–282.

[26] YOAZ, A., EREZ, M., RONEN, R., AND JOURDAN, S. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture* (Atlanta, GA, USA, June 1999), pp. 42–53.

[27] YOURST, M. T. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (San Jose, CA, USA, April 2007), pp. 23–34.