Tejas: A Java based Versatile Micro-architectural Simulator

Smruti R. Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, Eldhose Peter
Department of Computer Science, Indian Institute of Technology, New Delhi, India
E-mail: {srsarangi,rajshekark,prathmesh.kallurkar,mcs132582,eldhose}@cse.iitd.ac.in

***Abstract*—In this paper, we present the design of a new Java based, cycle-accurate, heterogeneous architectural simulator, *Tejas*. *Tejas* is a trace driven simulator, which is platform-independent. It can simulate binaries in any ISA and corresponding to virtually any operating system. It can itself run on virtually any machine. It is one of the fastest cycle accurate simulators available in academia. This is achieved through employing optimized data structures, improving the simulator's cache locality, and reducing the amount of wasteful work done. *Tejas* offers a rich library of architectural features that are modular and highly configurable. *Tejas* has been validated against real hardware (Dell PowerEdge R620 server) and has been shown to be more accurate than some of the most popular architectural simulators.**

I. Introduction

An architecture simulator is arguably the most important tool in computer architecture education, design, and research. It is used to teach basic concepts, prototype new designs, and estimate performance, temperature, and power. As a natural consequence of such requirements, many architectural simulators [1–5] have been developed and distributed under open source licenses over the last two decades. Most research groups use and extend them to evaluate their proposals. Research in the design of architectural simulators has mainly focused on incorporating novel features [6], parallelization [4, 7], and speed enhancing techniques [4, 7, 8] by sacrificing on cycle accurate guarantees. However, the problem of having a flexible and platform independent simulation framework that can run over the cloud, and can incorporate instruction emulators through a standardized interface has not received a lot of attention. We solve this problem in this paper by proposing the design of *Tejas*, which has been released under an open source Apache 2 license.

An architectural simulator consists of an instruction emulator that executes the workload, and a timing simulation engine that simulates the execution of the workload on a hypothetical machine. In *Tejas* we have decoupled these modules. This was done because instruction emulators are typically platform dependent, or can often be proprietary and make their services available over the network. The simulation engine, however, need not be tied to any platform. All other popular open source simulators (to the best of our knowledge) are written in C/C++, which makes them dependent on a platform to a large extent. In *Tejas*, the simulation engine is made platform independent by being implemented in Java. It can interact with many different instruction emulators that are possibly executing on remote machines through standardized interfaces. We have tested *Tejas* on Linux, Windows and OSX platforms, and present the results for the Linux platform in this paper. The transfer of the execution trace from the emulator to the simulator can be either online (the current distribution of *Tejas* supports shared memory, memory mapped files and sockets) or offline through trace files. Note that with trace files, the platform specific phase of trace collection needs to be done only once. The simulation of the trace, whose outcome is agnostic to the underlying platform, can then be done any number of times on any kind of machine.

Apart from being portable, *Tejas* is competitive with other simulators in terms of functionality. It simulates non-uniform caches, out-of-order pipelines, complex on-chip networks (NoCs), relaxed memory models, and CUDA based GPGPUs. In terms of speed, it is faster than several popularly used cycle accurate simulators, and has additionally been thoroughly validated against native hardware.

Our contributions can be divided into two categories. The first category of techniques are focused on creating an efficient transfer mechanism between the emulators and the timing simulator, and on leveraging the features of Java to the utmost extent. The second category of optimizations are general techniques that can be used with all high performance architecture simulators. They focus on methods to efficiently and accurately simulate out-of-order pipelines, memory hierarchies, implement coherence protocols with relaxed memory models, and simulate complex NoCs. We evaluate the cumulative effect of all our proposed techniques in Section VII for both serial and parallel workloads. We compare our results with two popular cycle accurate simulators: Gem5 and Multi2Sim. *Tejas* is on an average 25% faster than Multi2Sim and 220% faster than Gem5 (for serial workloads). The error in estimating execution time for serial and parallel workloads is 11.45 and 18.77% respectively, which is mostly better than previously published results. Tejas provides comparable architectural simulation capabilities with a fairly small code base – it contains 32k lines of code, which is significantly lesser than other competing simulators (Sniper: 76k lines, Multi2Sim: 108k lines, Gem5: 170k lines).

Readers can look at the full version of the paper [9] with the attached appendices for sample code in our custom ISA, VISA, details of the translation process, and simulation results on the Windows and OSX platforms.

A. Related Work and Motivation

Some of the most popular cycle accurate architecture simulators are SimpleScalar [2], Sesc [3], Gem5 [5], and Multi2Sim [6]. They typically have built-in emulators for specific ISAs, which can emulate the execution of binaries. Typically both the emulator and timing simulator are a part of the same software package. Multi2Sim is the closest to *Tejas*

in terms of functionality. Like *Tejas* it supports the simulation of heterogeneous systems with CPUs and GPUs. Similar to *Tejas*, Multi2Sim and Gem5 can also simulate the full software stack inclusive of the OS and hypervisor. Unlike, *Tejas* none of the popularly used simulators have platform independence as a major design goal.

There is another line of work that focuses on parallelizing simulation such as the papers on Sniper [4], Graphite [7], and ZSim [10]. Simulators such as Sniper [4] and ESesc [1] also incorporate novel techniques such as sampled simulation or analytical modeling, which allow them to skip a large number of instructions because their behavior is predictable. Both these techniques are orthogonal to our approaches. *Tejas* can be parallelized (see [11]), and can also be extended to incorporate sampled simulation.

Motivation: We believe that platform independence is an important goal to pursue. The first reason is that most users typically struggle with building and running a simulator on their systems. This is primarily because of the dependence on libraries with often antiquated versions, and because of discrepancies between compiler optimizations and OS libraries (such as stack randomization on Ubuntu platforms). Second is the need to make the simulator cloud-compliant. The typical usage of an architectural simulator is to perform design space explorations. This involves a massive number of mutually independent simulation runs, an ideal workload for a cloud setup. However, clouds are usually made up of heterogeneous platforms (ISA, OS, libraries). The file mode of *Tejas* provides the necessary insulation from all platform-specific idiosyncrasies.

Additionally Java is known for its superior programming constructs, support for garbage collection, type safety, and ease of debugging. Skeptics might argue that Java has traditionally been associated with lower performance. To debunk this myth we considered three of the most commonly used data structures in our simulator: priority queue, circular queue, and set associative cache. We implemented them in both C++ (g++ 4.7.2, optimization level 3, standard template library, libstdc++ 6.0) and Java (JDK 7) and turned on all the optimizations. We executed a workload containing a billion operations on a 2.53 GHz Intel Nehalem based system for all the three data structures. Java was found to be 6.5%, -4.9% and 23% faster than C++ for the priority queue, circular queue and cache structures respectively. The gain is primarily due to efficient just-in-time compilation features of the JVM.

II. ARCHITECTURE OF *Tejas*

The high level architecture of *Tejas* is shown in Figure 1. The emulator emulates the execution of the program and generates traces containing dynamic information such as instructions executed, load/store addresses, and branch outcomes (taken/not-taken). The emulator is dependent on the platform. The collected traces are then sent to the timing simulator, which is written in Java. Subsequently, the timing simulator translates the instructions into a custom instruction set called VISA (see Section V). The simulated cores take the stream of VISA instructions as an input and simulate the pipelines. They send the memory accesses to memory elements through the simulated on-chip network. At the end of the simulation, we report detailed statistics regarding the usage of each component, the IPCs, the miss rates of different memory structures, and the power consumption figures.

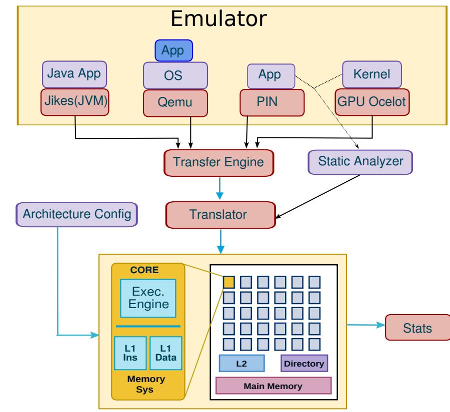

Fig. 1: High level architecture of *Tejas*

III. EMULATOR

Tejas is a trace driven simulator. The onus of correct execution of the benchmark is on the emulator, and not *Tejas*. Since many efficient open-source emulators are already available, our effort is limited to instrumenting these emulators for feeding information to *Tejas*. Currently, we have instrumented (1) Intel Pin [12] to emulate x86 binaries, (2) Qemu [13] to emulate full systems including the OS and hypervisor, (3) GPU Ocelot [14] to emulate CUDA-based GPGPU workloads, and (4) Jikes RVM to emulate Java bytecode. Note that PIN, Qemu, and Jikes have been released for a wide variety of platforms: Windows, Linux, and MacOS X, and thus they are aligned to our goal of platform independence. The modular design of Tejas allows the user to easily connect to different emulators, corresponding to different target platforms.

The emulator executes the concerned benchmark, and sends *events of interest* to the simulator. The primary events of interest are instruction type, operands, load/store value, and branch information. For parallel benchmarks, the emulator logs the synchronization action for each thread. For full system simulation, events such as system calls, interrupts, and process-switches are noted. The events of interest are encoded using a generic packet interface. The packet contains three fields `<ip, packet-type, value>`, each encoded as a 64 bit long value. The only exception is a packet describing the assembly text of an instruction. For such packets, the value field is a character array of 64 bytes (see Section IV).

IV. TRANSFER ENGINE

Emulated application threads run in parallel, and gigabytes of trace data are generated per second. It is necessary to send all of this data to the timing simulator using a high throughput channel. Here again, *Tejas*'s modular design allows any communication mechanism to be employed. The current distribution of *Tejas* provides support for: (1) shared memory, (2) files, (3) pipes and (4) network sockets. We find shared memory to be the best online mechanism. For an offline solution, we can save the traces in files, and run the timing simulation using the traces from the files.

To measure the peak throughput of each online mechanism, we wrote a small compute intensive loop based benchmark. For

TABLE I: Comparison of IPC techniques

IPC Technique	Speed (MB per second)
Sockets	9
Memory Mapped File	21
Shared Memory	24

this experiment, we measured the time taken to transfer 1 GB of data from the emulator to the simulator on an Intel Core i7 desktop (2.4 GHz processor, 4 GB RAM, Ubuntu Linux 12.04). Table I shows the results. We observe that sockets are the slowest (10 MBps). This is because of the overhead of system calls. However, they can be used to run the emulator remotely. For high throughput, shared memory is the best option (24 MBps). Communication with memory mapped files is slower, because we need to synchronize data with the hard disk, or the disk cache in main memory. Since we have a finite number of buffers, we need a method for the consumer(Java thread) to indicate to the producer(emulator) that it cannot accept more packets for the time being. In the case of sockets, we use a dedicated socket from the consumer to the producer, and for shared memory or memory mapped files, we use shared variables to indicate the status of the consumer.

Shared Memory: We will now briefly discuss the implementation of a shared memory based transfer engine for Linux on x86 platforms. We use the $shmget$ and $shmat$ calls to get and attach shared memory segments to the emulator processes. We avoid using a separate shared memory segment for each application thread because underlying operating systems typically place a limit on the number of shared memory segments a process can use. Subsequently, we divide the segment into n contiguous regions, one for each application thread. In each region we have a header, and a circular queue. We fix the size of each packet to 192 bytes, and allocate space for 50 packets in each circular queue. The header contains the status of the thread, the number of outstanding packets in the queue ($count$), and its head and tail pointers.

At the Java side, we use the Java Native Interface (JNI) to access shared memory. It allows us to write code in C that can be linked to the JVM in runtime. The second issue is that of *locking*. We need to be able to get locks to update the pointers to the circular queue, and the $count$ variable. We use a Peterson lock for this purpose. The implementation of the lock is optimized for the Intel x86 TSO (total store order) memory model. It uses a single fence instruction. JNI uses Intel x86 string instructions to transfer large amounts of data between memory locations in one go. Hence, we homogenized the structure of a packet, at the cost of space. Each packet contains three 64 byte fields. We use a single JNI call, $memcpy(void*, const\ void*, size_t)$ to copy an entire memory area into the address space of Java threads. Secondly, note that Java is big-endian while Intel x86 is little-endian. While transferring values to the JVM, JNI routines seamlessly convert little endian values to big endian ones.

Compensation for OS Jitter: We observed that for multi-threaded benchmarks, the simulated IPC across multiple runs of the same configuration varied by around 5%. This magnitude of noise is not acceptable in practical research scenarios. The primary reason for the variability is that some application threads were not getting scheduled by the OS as much as others. In this case, their associated cores in the timing simulator were remaining idle. To compensate for this, we impose a constraint that allows a simulated core to remain idle only if the corresponding application thread is waiting for an external blocking event such as a lock to be released. This technique reduced the variability (in terms of IPC) between runs from around 5% to 1%.

File Interface: *Tejas* can also read instruction execution traces from files. This mechanism is platform independent and enables fully deterministic simulations. While working with realistic workloads such as the Apache web server benchmark, the trace files can become extremely large. We use the *GZip* compressor to compress the trace files, not only because of a possible space crunch but also because of the performance degradation due to the increased amount of disk I/O. We achieved a compression ratio of 70-80% for all our workloads. We then use the *GZIP* file reader that is a part of Java 7 to incrementally decompress the trace file on-the-fly and provide the traces to the timing simulator. In the case of the GPU traces [15], we achieve a further reduction in the size of the trace files using a simple observation that all the blocks of a CUDA kernel contain the same set of instructions. Hence, we store the information regarding these instructions separately in a hash file. Next, we generate a set of new binary trace files containing only the instruction pointers (in the hash file), branch outcomes and memory addresses. This lead to an additional 10-fold decrease in the size of the trace files for all of the benchmarks in the *Rodinia* benchmark suite.

V. TRANSLATION MODULE

We translate all our simulated ISAs into a RISC ISA called *Virtual Instruction Set Architecture*, or VISA. This is done to provide a homogeneous timing simulation interface (across ISAs). It is fairly flexible, and has sufficient information to perform a timing simulation. This is a standard technique. Other cycle accurate architecture simulators such as Gem5 [5], SESC [3], and PTLsim [16] also adopt a similar approach. We acknowledge that this translation from the native ISA to VISA can be a source of inaccuracy. However, our experiments with the x86 ISA show acceptable simulation error statistics (11.45%, see Section VII). We believe that simpler native ISAs such as ARM or MIPS, which are more closer to VISA, will display an even lower translation-induced error. We employ caching to improve the speed of translation. The *translation cache* contains assembly strings of recently translated target instructions, and their mappings to VISA counterpart(s). This technique helped leverage the benefits of temporal/spatial locality and realize a 3.4x improvement in performance..

A CISC instruction set such as x86 contains hundreds of instructions. Some of these instructions are very rarely used. We ignore such instructions. This approach ensures that VISA does not get bloated with ISA specific details, and is still able to represent a majority of the operations performed. We have noted that the dynamic coverage of the translator is $> 99\%$ for a large number of benchmarks. We are aware of the fact that the translation of CISC instructions (e.g., x86) to VISA based RISC instructions can be a source of error. We borrow ideas from the translation engine of the PTLsim simulator [16] that has been rigorously validated against native x86 hardware.

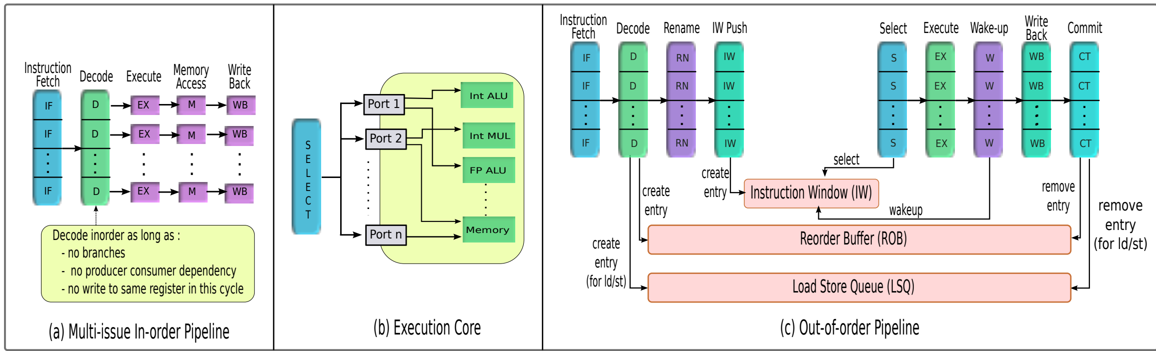

Fig. 2: Pipelines in *Tejas*

A. *Static Analysis of the Benchmark*

The communication of instructions between the emulator(s) and the simulator can be a bottleneck for performance. We chose to reduce the amount of the information communicated, without compromising on correctness, by doing some initial pre-processing. The benchmark executable undergoes a static analysis phase, where a map of each basic block to a VISA basic block is constructed (with place-holders for memory addresses and branch outcomes). During run-time, we need to only transfer the program counter (PC) of each executed instruction. The IP is then used to look up the static map and obtain the micro-ops to be simulated. Additionally, memory addresses in the case of memory operations, and branch outcomes in the case of branches, are transferred from the emulator. These are used to fill the place-holders. The rest of the details (e.g., opcode, register operands) can be inferred from the constructed map.

VI. Micro-architectural Simulation

A. *Semi Event Driven Model*

One of the popular ways to simulate hardware is to adopt an iterative style, as done in SimpleScalar [2]. In every iteration, each structure is advanced through one clock cycle. Although highly simple, often we end up simulating structures that are idle. The alternative is to follow a discrete event model as the one used by SESC [3]. A list of events is maintained in an event queue. In each cycle, we dequeue the events scheduled for that cycle and process them, which may in turn generate more events. An event queue is essentially a priority queue. Operations on a priority queue are computationally expensive, even for moderately sized queues.

To capture the best of both worlds we adopt a semi-event driven model. For activities that occur regularly with predictable latencies we use an iterative approach. An example is decoding, which happens every cycle unless there is a stall. For activities that occur irregularly with unpredictable latencies (e.g, executing a load/store) an event-driven approach is used.

B. *Pipelines*

A multi-issue inorder pipeline and a complex out-of-order(OoO) pipeline have been implemented in *Tejas*. The multi-issue inorder pipeline shown in Figure 2(a) is based on the design used by the Intel Pentium processor [17]. The out-of-order pipeline implementation (see Figure 2(c)) simulates aggressive speculation. It simulates detailed wakeup, replay, select, and multi-level bypass mechanisms to allow immediate resolution of data dependencies. Additionally, we simulate a register-allocation table (RAT) based renamer, a reorder buffer, and a physical register file. The sizes of each unit including the issue, decode, and retire widths are configurable through an XML file. Similarly, the number of functional units of each type, their latencies and reciprocals of throughput can be specified.

The pipeline stages are simulated in reverse order to ensure correctness: commit/write-back first and fetch last. The reason for this is that at time t, we want the ith stage to work on the output that the $(i-1)^{th}$ stage generated at time $t-1$. If we simulate stages in the forward order, the ith stage will work on the output that the $(i-1)^{th}$ stage generated at time t.

We observed that there are many corner cases that can arise while simulating an aggressive OoO pipeline. Let us consider one such example. A dependent instruction j may be in the rename stage when i completes (see Figure 2). Since i has not yet reached the write-back stage, j cannot get its operand from the register file. And since j is not in the instruction window, it misses the wakeup signal. j is now indefinitely stuck in the pipeline. To handle such cases in *Tejas*, we send two wakeup signals to the instruction window in successive cycles. This ensures j gets its operands and makes progress. Our GPU pipeline is described in detail in the paper by Malhotra et al. [15]. Our pipeline is modeled according to the designs used in NVIDIA's Tesla, Fermi and Kepler GPUs.

C. *Branch Predictor*

The emulator sends the traces of only those instructions that were on the correct control path, and merely informs the simulator if a branch was taken or not taken. When the simulator receives a branch, it performs the prediction. The predicted outcome and the outcome provided by the emulator are subsequently compared. A mismatch indicates a misprediction, which is simulated by stalling the pipeline for a pre-defined (configurable) number of cycles as is done in other simulators (see SESC [3]). A wide array of branch predictors are implemented in *Tejas*: Always Taken, Always Not Taken, Bimodal, GAg, GAp, PAg, PAp, GShare, TAGE [18] and tournament predictors.

D. *Cache Hierarchy*

Figure 3 shows the high level working of a cache, which is the basic building block in *Tejas*'s memory hierarchy. The

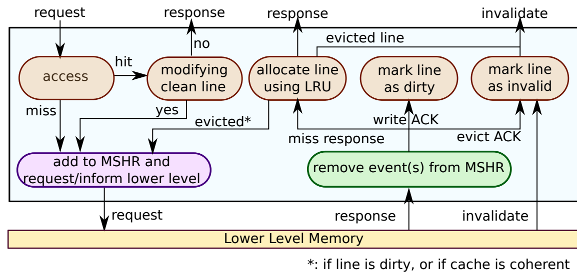

Fig. 3: High level working of a cache (simplified)

cache model is highly detailed, allowing specification of size, latency, associativity, block size, write mode, port type, port occupancy, number of ports and the MSHR (Miss Status Holding Register) size. A cache can be private to a core, or shared among a set of cores. Non-uniform cache access (NUCA) is also supported (see Section VI-F).

A structure critical to both simulator accuracy and performance is the MSHR, which has not received a lot of attention in competing simulators. It contains those cache requests that encountered a miss and are awaiting fulfillment. The MSHR is a fixed size structure; its filling up results in back pressure – requiring the upper cache levels/pipeline to back off from sending further requests. The MSHR is a highly used structure, and implementing the MSHR as a naive array results in serious performance issues. Every time a response comes from the lower levels, a sequential search (O(n)) is required to find all requests corresponding to the received cache line. These requests are then serviced, and removed from the MSHR. Removing an entry from an array requires re-arranging the elements, which is again expensive (O(n)). Experimentation with different data structures and algorithms revealed that implementing the MSHR using a linked list of linked lists is the best alternative.

Each second level list consists of requests for the same block. When a miss for address x occurs, we first sequentially search the first level list for a list corresponding to x. If found, we append the request to this list; else, we create a new list and append it to the first level list. This sequential search is not that expensive because of the two-level scheme – the length of the first level list is typically not very long. When the response (address x) from a lower level arrives, we again perform a sequential search at the first level, find the list of requests corresponding to x, and service them. This linked list is then purged (O(1)).

E. *Coherence Protocol*

We implemented directory based cache coherence in *Tejas*. A useful feature is the flexibility of specifying coherence at any level (L1 or beyond). *Tejas* implements the MESI protocol with additional extensions. However, the simple "text-book" 4-state protocol does not capture the entire complexity of a real coherence protocol. A coherence protocol has a very important temporal component – messages take time to reach the destination over the NoC. During this transit, the different components are not consistent with each other. Effectively, it cannot be clearly ascertained, which of the 4 states the system is in. Hence practical coherence protocols have 100+ states. Diligently handling every possible scenario will result in a massive state explosion. The code will become very unwieldy, and implementing an industry strength protocol is not the goal of an academic simulator.

Hence, in *Tejas*, we adapted our protocol to handle only those scenarios that occur with a reasonably high probability. If any of the unhandled scenarios transpire, we perform a *corrective operation* – gracefully move the system to a legal state and continue. We audit the frequency of such corrective operations – they were found to occur only around once every 10^5 data cache accesses. Table II lists some scenarios that require a corrective operation.

TABLE II: Example scenarios requiring *corrective operations*

Case	Sequence of Events
write miss at c_1, and c_2 is a sharer	$DIR \xrightarrow{INV} c_2$ reaches before $DIR \xrightarrow{FWD} c_2$
simultaneous write hit at c_1 and c_2	say, c_2 reaches DIR first: DIR:$c_2[M]$, DIR:$c_1[I]$; $c_1 \xrightarrow{WH} DIR$ reaches
DIR and c_1 simultaneously evict the same line at c_1	c_1 evicts line, DIR:$c_1[I]$; $c_1 \xrightarrow{INV} DIR$, $DIR \xrightarrow{INV} c_1$

c_i: coherent caches, DIR: directory
INV: invalidate msg, FWD: forward line msg, WH: inform write hit
DIR:$c_i[X]$: DIR updates state(MESI) of c_i to X

Consider the following sequence of events. Cache c evicts address X, and informs the directory. It then receives a read to the same address, and requests the directory for X. The *evict* and *ReadMiss* messages may be reordered by the NoC, possibly resulting in the *ReadMiss* request reaching first. The directory, however, believes that c is a sharer for address X at this point. To avoid such inconsistent states, we adopt a technique called *line locking*. Whenever a coherent cache performs a write (write-hit) or an eviction, it locks the line and informs the directory. As long as the line is locked, the line may not be evicted. The directory updates its state and sends an acknowledgment to the cache. The cache on receipt of the ACK updates its state and unlocks the line. This reduces the frequency of inconsistent states, resulting in a significant decrease in the number of corrective operations performed, thus improving the simulation accuracy.

F. *Non-Uniform Cache Architectures (NUCA) and DRAM*

Tejas implements NUCA caches, where we decouple the physical organization of a cache from its logical organization. We provide both static NUCA and dynamic NUCA. The NUCA implementation in *Tejas* is augmented with a victim buffer. It is essentially a buffer associated with each bank containing the block addresses of the last k lines evicted from the bank and placed in another bank. This is required to handle a tricky scenario in D-NUCA called the *two-copy problem* – consider two banks b_1 and b_2. A request comes to b_2 for address x, which is absent in b_2 and present in b_1. b_2 forwards the request to b_1. Before the request reaches b_1, the latter evicts line x and sends it to b_2. When the request reaches b_1, it suffers a miss, and the request is sent to the lower levels of the memory hierarchy. The request is serviced and the line is placed in b_1. Now, address x is wrongly present in both banks. A victim buffer helps us avoid this by enabling us to place a line from the lower level in a bank, only if it does not have a corresponding entry in the victim buffer. This was necessary to ensure correctness because this situation was fairly common in our DNUCA implementation.

At the moment, we are developing a detailed DRAM model (DDR3 and DDR4). Additionally, we are also working on simulating hardware transactional memory.

G. Network-on-Chip (NoC)

The NoC implementation is generic – it can connect components such as cores, cache banks, directories and memory controllers. Every component of *Tejas* has a *network interface* attribute. This is essentially the component's point of contact with the rest of the system. If A wants to send a message to B, it simply executes $B.getPort().put(msg)$. If A and B have the same network interface (assume a private L1 wishes to communicate with its write buffer), then the message is simply transferred to the receiver immediately. If A and B have different network interfaces then the message is sent over the network.

Our NoC implementation is fairly rich. We support packet switch traffic in both cut-through and worm-hole routing modes. We support a variety of topologies: bus, ring, mesh, torus, fat tree, omega and butterfly. *Tejas* supports both static (X-Y, West-First, North-Last, Negative-First) and adaptive routing schemes. We simulate complex multi-port routers with support for virtual channels, bypassing, and lookahead routing. *Tejas* is the first public domain simulator to also support optical networks. We support SWMR (single writer-multi reader) and MWSR (multi writer-single reader) networks with support for arbitration and optical barriers.

H. Power and Temperature Modeling

Tejas uses the McPAT [19] power model. We have an additional *power pack* that takes the default XML based configuration file as input, creates a configuration file for McPAT, runs McPAT, and creates a new *Tejas* configuration file that has the energy per access for each structure. At the end of the simulation we report the power consumption of all the micro-architectural structures. We can additionally interface with the Orion2 [20] power model for NoCs, and also generate an energy consumption trace that can be provided as an input to a temperature simulator such as HotSpot. McPAT [19] and Orion2 [20] have been rigorously validated by the respective authors.

I. Relaxed Memory Consistency Models

Tejas supports sequential consistency as well as weak consistency models such as RC and TSO. For simulating relaxed memory models, we need to have support for *memory fence* instructions, and we need to incorporate the concept of read and write *completion* in the memory system. To simulate the *memory fence* instruction and simulate memory access completion, we need the memory system to send an acknowledgement after a memory access (load/store/sync. operation) is globally visible. Read operations are synchronous by nature. Write operations, however, require explicit acknowledgments to be sent. We thus need to keep track of the state of the memory system. Only when all the changes pertaining to a store request are reflected in the memory system and subsequent loads are guaranteed to get the value written by the store (or a newer value), we signal the completion of the request. This required changes in the MSHR and the directory. In specific, we signal the completion of a store, when we get exclusive access to the line in the directory.

J. Memory Optimizations

One of the main contributors towards *Tejas*'s performance is its superior cache locality. As elaborated in the high-level architecture (Section II), there are four main phases in the simulation process – *emulation*, *communication*, *translation* and *micro-architecture simulation*. The straight-forward way of implementing this would be to operate at the granularity of a single instruction – emulate a single instruction, communicate the information packets, translate it to a set of VISA instructions, and simulate these. Such an approach, though simple, suffers from poor locality, in both the code and data segments. Instead, we work at a higher granularity of 50 packets. This results in considerable time spent executing one phase before moving to the next. The improved cache locality results in improved simulation speed. Increasing the granularity further proved counter-productive as the buffering structures between the phases became prohibitively large.

One of the biggest advantages of Java is dynamic memory management. It makes programming very easy because the user does not have to manually free memory segments. However, the garbage collector has its limits. For extremely frequently instantiated data structures such as the *Instruction* object or the *Event* object, dynamic memory management is associated with very large performance overheads. Hence, we decided to use *pooling* (object reuse) for the Instruction objects. Instruction objects are created in the translator (right after translation), and are destroyed upon retirement. Since these points are very clearly defined in the code of our simulator, it was very easy to fetch and return objects to a pool of instructions. We further optimize the pool by making its size variable – we start with a reasonably small sized pool (function of the number of static instructions in the executable), and grow it on demand. This provides an additional improvement in performance. A related optimization is with respect to operands – an object representing every legally possible operand is created at the start of simulation. References to these are used during the simulation, instead of creating them anew. They do not have any dynamic component, allowing multiple instruction objects to reference the same operand object.

We did not use pooling for Event objects because this would unnecessarily complicate the code. Event objects are used throughout the simulator's code and there is no clear point at which they stop getting used. We thus employ an alternate strategy. We avoid creating multiple instances of the *Event* class by reusing the same object as much as possible. The *Event* class is highly configurable and can represent many kinds of sub-events, so that the same event object can go through the NoC to the directory, and then to the owner of the cache line. We further reduce the frequency of the invocation of the garbage collector by hand coding some of the frequently used data structures such as the *linked list*. The implementation of the linked list in *java.util.LinkedList* allocates a heavy container object during each addition operation, and encapsulates the given data in it. This increases the chances of the invocation of the garbage collector. Instead, we have a $next$ pointer with the parent class of every simulation element that can be a part of a linked list. We thus do not require an encapsulating object.

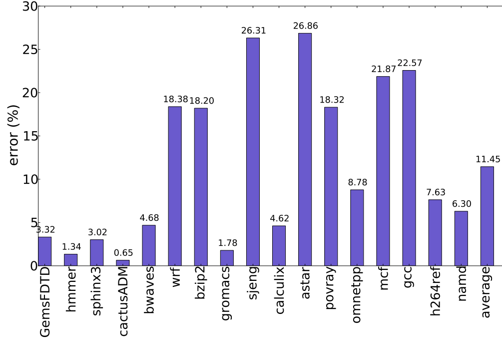

Fig. 4: Validation : *SPEC CPU2006* suite

VII. Evaluation

A. Validation against Native Hardware

In this section, we discuss the validation of *Tejas* against a PowerEdge R620 Dell server (for details see Table III). We used a suite of 17 serial workloads from the *SPEC CPU2006* benchmarks and a suite of 11 parallel workloads from the *SPLASH-2* benchmarks. First, with the help of the Linux "perf" command, we recorded the number of cycles taken to execute each of the benchmarks on the target machine (averaged across 10 runs). Subsequently, *Tejas* was configured to mimic the target machine as closely as possible (see Table IV for details). The simulated time reported by *Tejas* was compared against the results obtained from the native hardware, as per the standard procedure adopted by other simulators (see the validation results of Gem5 [21]).

TABLE III: Details of the reference hardware

Parameter	Value	Parameter	Value
Microarchitecture	Intel Sandybridge	Number of cores	12
Main Memory	32 GB	Memory Type	ECC DDR3
L1 i-cache and d-cache	32 KB	L2 cache	256 KB
L3 Cache	15 MB	Frequency	2GHz
Hyper-threading/DVFS	Disabled	Reorder Buffer	168 micro-ops
Load Buffer Size	64	Store Buffer Size	64
Operating System	Ubuntu 12.10 Linux 3.5.0-36-generic, 64-bit		

Sequential benchmarks validation: Figure 4 shows the validation error for serial workloads. The average absolute error (in execution time) is 11.45%. 10 out of 17 benchmarks have an error less than 10%. Only 4 benchmarks have errors in the 20-30% range (*sjeng*, *astar*, *mcf*, and *gcc*).

Parallel benchmarks Validation: Figure 5 shows the validation error for parallel workloads. The average absolute error was observed to be 18.77%. In this case, the average error is greater primarily because of the jitter introduced by variable reordering of the application threads by the OS scheduler. Secondly, there are hardware events such as I/O events that induce jitter, and lastly we are not privy to all the details of the operation of the cache coherence protocols in Intel systems. Only 3 benchmarks had errors more than 25% namely *radiosity*, *radix* and *water-spatial*. For most of the benchmarks, the error ranges from 10 to 17%.

Comparison with other Simulators: *Tejas* is more accurate for both serial and parallel benchmarks as compared to most of the other widely used architecture simulators (for which published results for x86 hardware are available). MARSS [22] is a cycle-accurate simulator based on PTLsim. It is a tool built on QEMU and provides fast full system

TABLE IV: Simulation parameters

Parameter	Value	Parameter	Value
Pipeline			
Retire Width	4	Integer RF (phy)	160
Issue Width	6	Float RF (phy)	144
ROB size	168	Predictor	TAGE [18]
IW size	54	Bmispred penalty	8 cycles
LSQ size	64		
iTLB	128 entries	dTLB	128 entries
Integer ALU	3 units	lat = 1 cycle	RoT = 1
Integer Mul	1 unit	lat = 3 cycles	RoT = 1
Integer Div	1 unit	lat = 21 cycles	RoT = 12
Float ALU	1 units	lat = 3 cycles	RoT = 1
Float Mul	1 unit	lat = 5 cycles	RoT = 1
Float Div	1 unit	lat = 24 cycles	RoT = 12
RoT : reciprocal of throughput			
Private L1 i-cache, d-cache			
Write-mode	Write-Through	Block size	64
Associativity	8	Size	32 kB
Latency	3 cycles		
Private Unified L2 cache			
Write-mode	Write-Back	Block size	64
Associativity	8	Size	256 kB
Latency	6 cycles		
Shared L3 cache			
Write-mode	Write-back	Block size	64
Associativity	8	Size	15 MB
Latency	29 cycles		
Main Memory Latency		200 cycles	
NOC and Traffic			
Topology	Bus	Latency	1 cycle
Flit size	32 bytes		

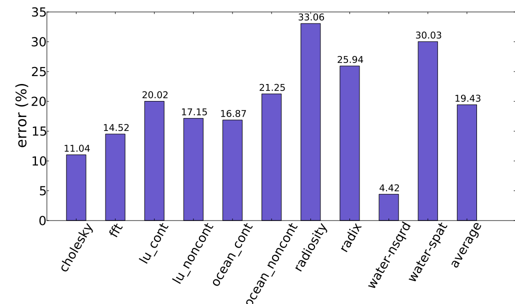

Fig. 5: Validation : *SPLASH-2* suite

simulation. MARSS has been validated against a x86 target machine with the Intel Xeon E5620 processor. For the *SPEC CPU2006* benchmark suite, it has errors ranging from -59.2% to 50%, with an average absolute error of 23.46%.

Sniper [4] sacrifices cycle accuracy in favor of simulation speed by employing sampled simulation. It has been validated against a 4-socket Intel Xeon X7460 Dunnington shared memory machine. An average absolute error of 25% has been reported for the *SPLASH-2* benchmark suite (we report 18.77%). FastMP [8] simulates a subset of cores in detail and uses this information to approximate memory traffic for other cores. FastMP has been validated against a real x86 machine using the *SPEC CPU2006* benchmark suite. It suffers from an average error of 9.56% but for some of the benchmarks the error is as high as 40% (our maximum error is roughly 27%). To the best of our knowledge other popular simulators such as SESC [3] and MacSim [23] have not been validated. Multi2sim, which is a heterogeneous simulator, has been validated for the GPU framework but its CPU simulation framework has not been validated and published (to the best of our knowledge). Our GPU pipeline has been validated by Malhotra et al. [15] (error $< 7.67\%$). Gem5 has been validated

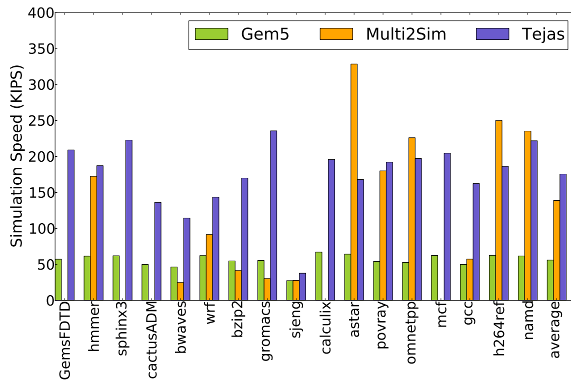

Fig. 6: Speed – serial workloads

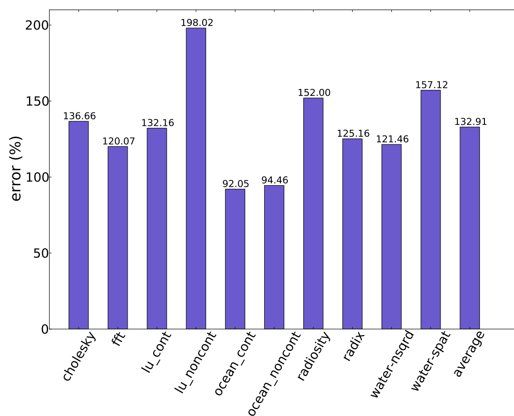

Fig. 7: Speed – parallel workloads

against an ARM Cortex-A15 processor with a mean error of 13% for *SPEC CPU2006* benchmarks and mean error of 17% for dual core simulations of parallel benchmarks. Note that Cortex-A15 has a much simpler pipeline than Intel Sandybridge, yet our error numbers are better for serial benchmarks.

B. Performance

Next, we compare the performance of *Tejas* with two popularly used cycle accurate simulators: Gem5 [5] and Multi2Sim [6]. We run all our experiments on a Dell server (see Table III). Figure 6 shows the relative speeds of simulation (including emulation, trace transfer and translation) in terms of *kilo-instructions simulated per second* (KIPS) for serial benchmarks (our Multi2Sim simulations did not complete for some benchmarks). *Tejas* outperforms Gem5 and Multi2Sim, with an average speed of around 175 KIPS. It is on an average 25% faster than Multi2Sim and 220% faster than Gem5. Figure 7 shows the simulation speeds of different benchmarks in the Splash2 suite. The average simulation speed of *Tejas* is around 140 KIPS. We were unable to simulate the Splash2 suite in Multi2Sim with acceptable validation numbers. The Gem5 simulator simulates parallel workloads only in the full system mode, which would not make for a fair comparison against *Tejas*. Both *Tejas* and Multi2Sim can simulate parallel applications without simulating the OS. In any case, the simulation speed of Gem5 is 3-4 KIPS for parallel applications, and thus *Tejas* is two orders of magnitude faster.

VIII. Conclusion

In this paper, we have presented the design of the *Tejas* architectural simulator, which has been designed to be platform independent. We achieve this by decoupling the instruction emulation and the timing simulation. We can use a wide variety of platform dependent emulators that can provide us execution traces from many different platforms with different system software and different ISAs. The timing engine has been written in Java, making it platform independent. *Tejas* supports most of the advanced features provided by competing cycle accurate simulators and additionally outperforms other simulators in its class both in terms of performance and accuracy vis-a-vis native hardware.

References

[1] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *HPCA*, 2013.
[2] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, 2002.
[3] SESC: SuperESCalar Simulator. [Online]. Available: http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/
[4] T. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, 2011.
[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.
[6] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. R. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *PACT*, 2012.
[7] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *HPCA*, 2010.
[8] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "Fastmp: A multi-core simulation methodology," in *MOBS*, 2006.
[9] Tejas: A java based versatile micro-architectural simulator. [Online]. Available: http://www.cse.iitd.ac.in/%7Esrsarangi/files/papers/patmospaper.pdf
[10] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*, 2013.
[11] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi, "Partejas: A parallel simulator for multicore processors," in *ISPASS (Poster)*, 2014.
[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, 2005.
[13] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005.
[14] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *PACT*, 2010.
[15] G. Malhotra, S. Goel, and S. R. Sarangi, "Gputejas: A parallel simulator for gpu architectures," *HiPC*, 2014.
[16] M. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *ISPASS*, 2007.
[17] D. Alpert and D. Avnon, "Architecture of the pentium microprocessor," *Micro*, 1993.
[18] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *JILP*, 2006.
[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[20] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: a fast and accurate noc power and area model for early-stage design space exploration," in *DATE*, 2009.
[21] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *ISPASS*, 2014.
[22] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *DAC*, 2011.
[23] H. Kim, J. Lee, N. Lakshminarayana, J. Lim, and T. Pho, "Macsim: Simulator for heterogeneous architecture," *Georgia Institute of Technology*, 2012.

Appendix A
VISA Instruction Set

The simulated architecture in Tejas executes instructions of the custom instruction set VISA. VISA is a simple RISC instruction set. The different instruction types are listed in Table V.

TABLE V: VISA Instruction Set

Instruction Class	Instruction Type(s)	Description
Compute	integerALU integerMul integerDiv floatALU floatMul floatDiv	Takes two source operands (register or immediate) and a destination register
Memory	load store	Takes two operands – memory location and value (register or immediate)
Control Flow	jump branch	unconditional jump conditional jump
Data Movement	mov xchg	Move source operand to destination operand Exchange two operands
	nop inValid	No operation To mark the end of the input stream. Indicator to stop simulation.

Each VISA instruction can have up to three operands. The operand types provided are listed in Table VI.

TABLE VI: Operand Types

Operand Type	Description
integerRegister	Integer Register
floatRegister	Floating Point Register
immediate	Immediate Value
memory	Used to specify the address in loads and stores. A memory operand is recursively defined in terms of two operands – the first an integer or immediate operand, the optional second an immediate operand.

Appendix B
Translation in Tejas

Translation is composed of two phases: *riscify* and *fuse*.

A. *Riscify*

Riscify is a static task, done once at the beginning of every simulation run. It involves building a map from each CISC instruction in the benchmark executable to a functionally equivalent set of VISA micro-ops.

Consider the example illustrated in Figure 8. The x86 instruction considered adds the contents of register rax to the value at the memory location pointed to by rbx. The result is to be placed at the same memory location. The *riscification* of this instruction makes an entry in the map (indexed by the x86 instruction's IP) with a set of three VISA micro-ops:

- load (%reg1), %tmpreg1
- add %reg0, %tmpreg1, %tmpreg1
- store %tmpreg1, (%reg1)

The assembly has been described in AT&T like syntax with the destination operand mentioned at the end.

The *Riscification process* consists of two steps: (1) breaking one x86 operation in to multiple VISA operations (2) mapping each operand of x86 to an operand in VISA. For each operand in the x86, we maintain an operand in VISA. Notice that the value of the memory location is not known until the program is actually executed. This value is added by the Fuse phase.

For each x86 instruction, we issue only one read request to the instruction cache. The address of the read request is the x86 instruction's IP. This is keeping in line with CISC machines, where CISC instructions are fetched, decoded into RISC micro-ops in the decode stage, and subsequently executed.

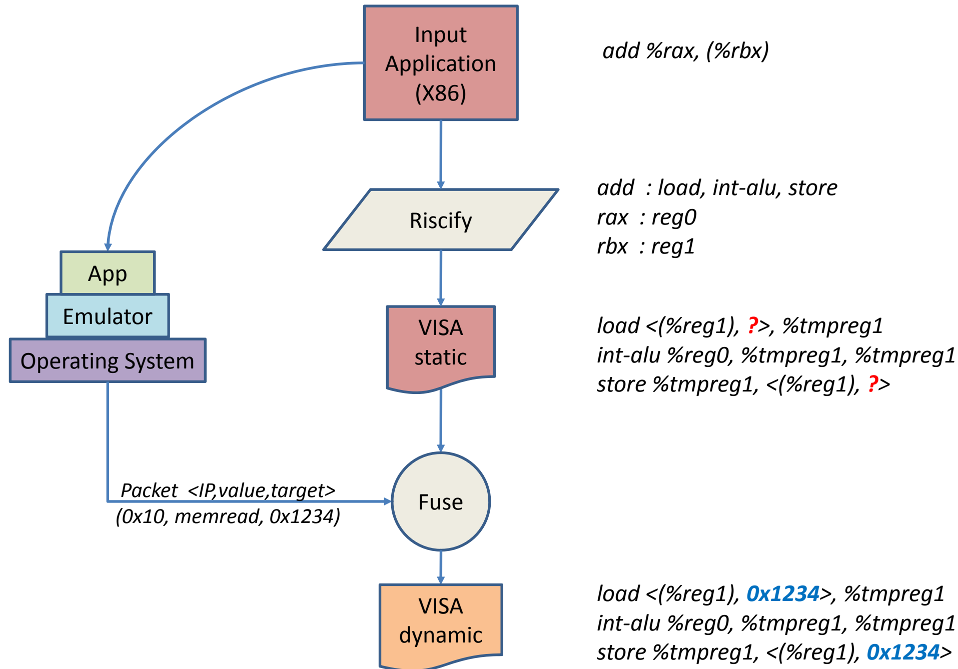

Fig. 8: Translation in Tejas

B. Fuse

Fuse is a dynamic task, done during the emulation of the benchmark. For each instruction emulated, a packet describing the emulation is sent to the translator by the emulator. The translator uses the IP to index the static map constructed in the *riscify* step. The micro-ops thus obtained are then simulated. It must be noted that the operation type and operands are not transferred from the emulator to the translator, thus optimizing on inter-process communication and improving simulation speed.

For every memory instruction, the actual address accessed is communicated by the emulator to the translator. The micro-ops are then *fused* with this information. Let us go back to the example described in Figure 8. The load instruction is now fused with the received address, say $0x1234$, to give:
load <(%reg1), 0x1234>, %tmpreg1
The store instruction is fused in a similar fashion.

Likewise, for every conditional branch instruction, the branch outcome (taken or not-taken) is communicated by the emulator to the translator. The translator fuses the branch micro-op with this information. The simulated pipeline then uses this when simulating the branch predictor.

Appendix C
Simulation Speed of Tejas

The details of the simulated architecture is given in Table VII.

TABLE VII: Simulation parameters

Parameter	Value	Parameter	Value
Pipeline			
Retire Width	4	Integer RF (phy)	160
Issue Width	6	Float RF (phy)	144
ROB size	168	Predictor	TAGE [18]
IW size	54	Bmispred penalty	8 cycles
LSQ size	64		
iTLB	128 entries	dTLB	128 entries
Integer ALU	3 units	lat = 1 cycle	RoT = 1
Integer Mul	1 unit	lat = 3 cycles	RoT = 1
Integer Div	1 unit	lat = 21 cycles	RoT = 12
Float ALU	1 units	lat = 3 cycles	RoT = 1
Float Mul	1 unit	lat = 5 cycles	RoT = 1
Float Div	1 unit	lat = 24 cycles	RoT = 12
RoT : reciprocal of throughput			
Private L1 i-cache, d-cache			
Write-mode	Write-Through	Block size	64
Associativity	8	Size	32 kB
Latency	3 cycles		
Private Unified L2 cache			
Write-mode	Write-Back	Block size	64
Associativity	8	Size	256 kB
Latency	6 cycles		
Shared L3 cache			
Write-mode	Write-back	Block size	64
Associativity	8	Size	15 MB
Latency	29 cycles		
Main Memory Latency		200 cycles	
NOC and Traffic			
Topology	Bus	Latency	1 cycle
Flit size	32 bytes		

A. Linux Platform

TABLE VIII: Details of the reference hardware

Parameter	Value	Parameter	Value
Microarchitecture	Intel Sandybridge	Number of cores	12
Frequency	2 GHz	Main Memory	32 GB
L3 Cache	15 MB	Hyper-threading	Disabled
DVFS	Disabled	Java Version	1.7
Operating System	Ubuntu 12.10 Linux 3.5.0-36-generic, 64-bit		

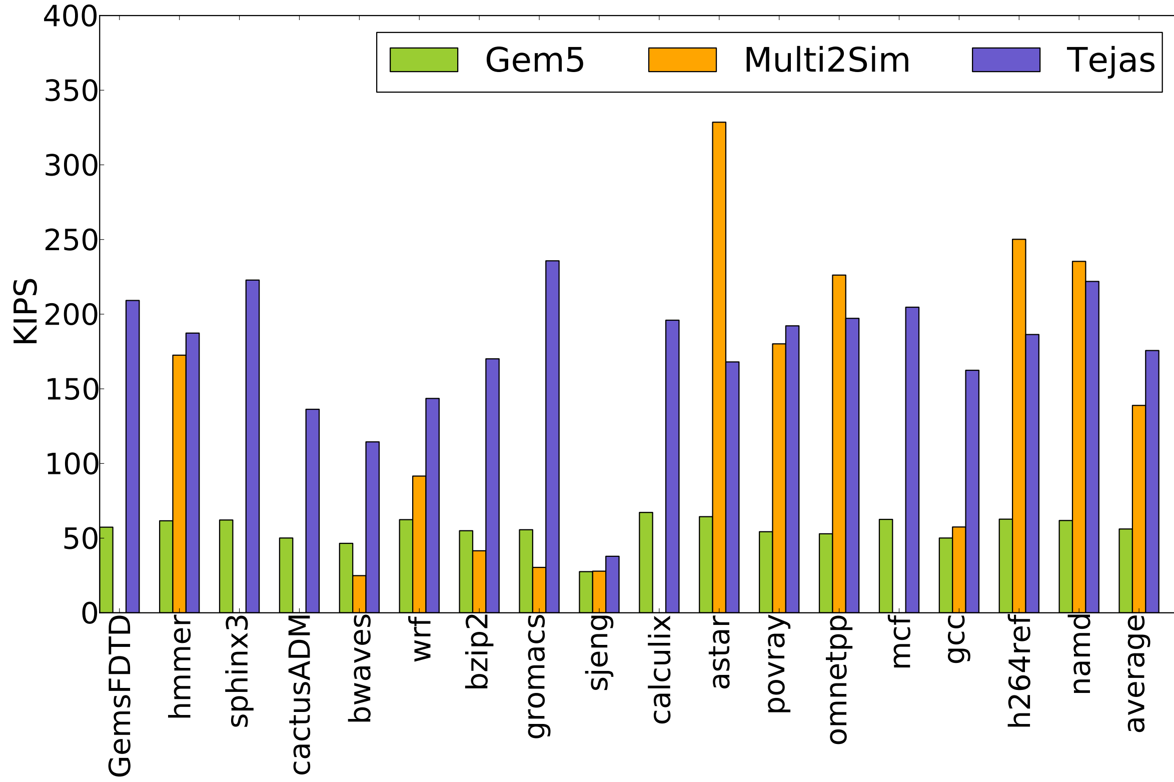

Fig. 9: Linux platform – Simulation speed comparison

B. Windows Platform

TABLE IX: Details of the reference hardware

Parameter	Value	Parameter	Value
Microarchitecture	Intel Ivybridge	Number of cores	4
Frequency	3.4 GHz	Main Memory	16 GB
L3 Cache	8 MB	Hyper-threading	Disabled
DVFS	Disabled	Java Version	1.8
Operating System	Windows 7 Enterprise, 64-bit		

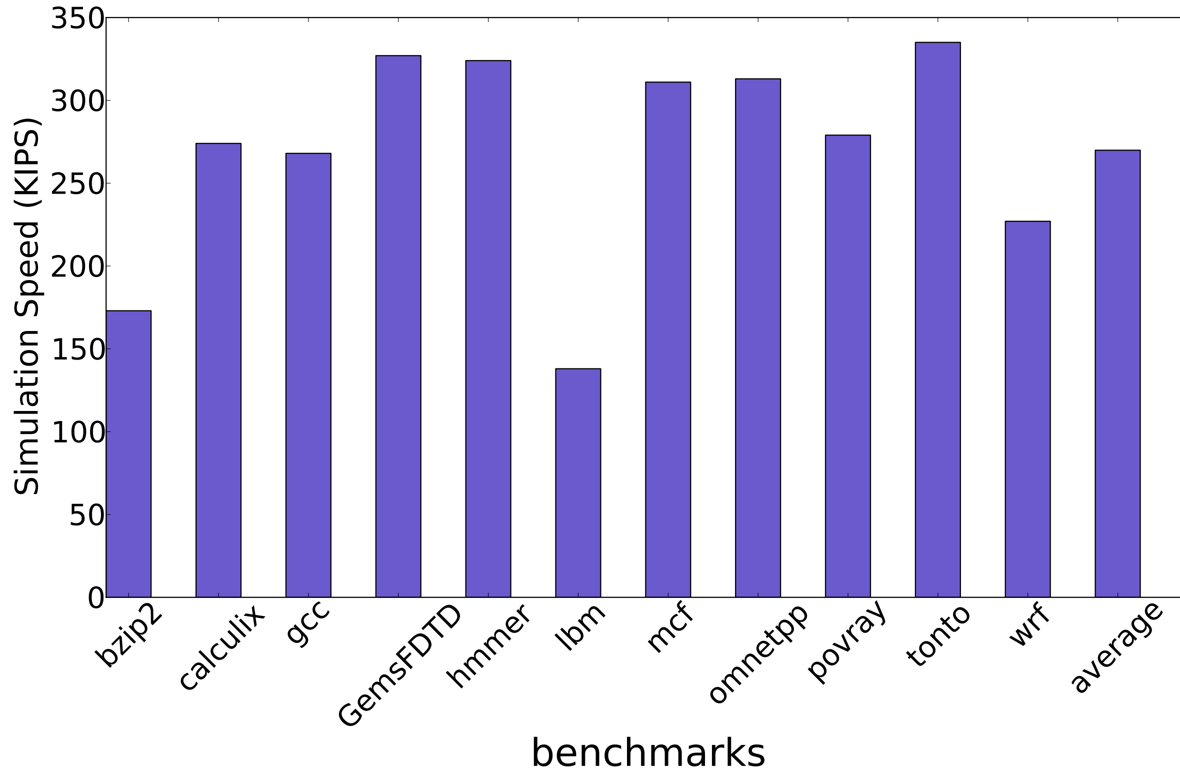

Fig. 10: Windows platform – Simulation speed

C. OSX Platform

TABLE X: Details of the reference hardware

Parameter	**Value**	**Parameter**	**Value**
Microarchitecture	Intel Haswell	Number of cores	4
Frequency	2.7 GHz	Main Memory	8 GB
L3 Cache	6 MB	Hyper-threading	Disabled
DVFS	Disabled	Java Version	1.6
Operating System	MacOS 10.10, 64-bit		

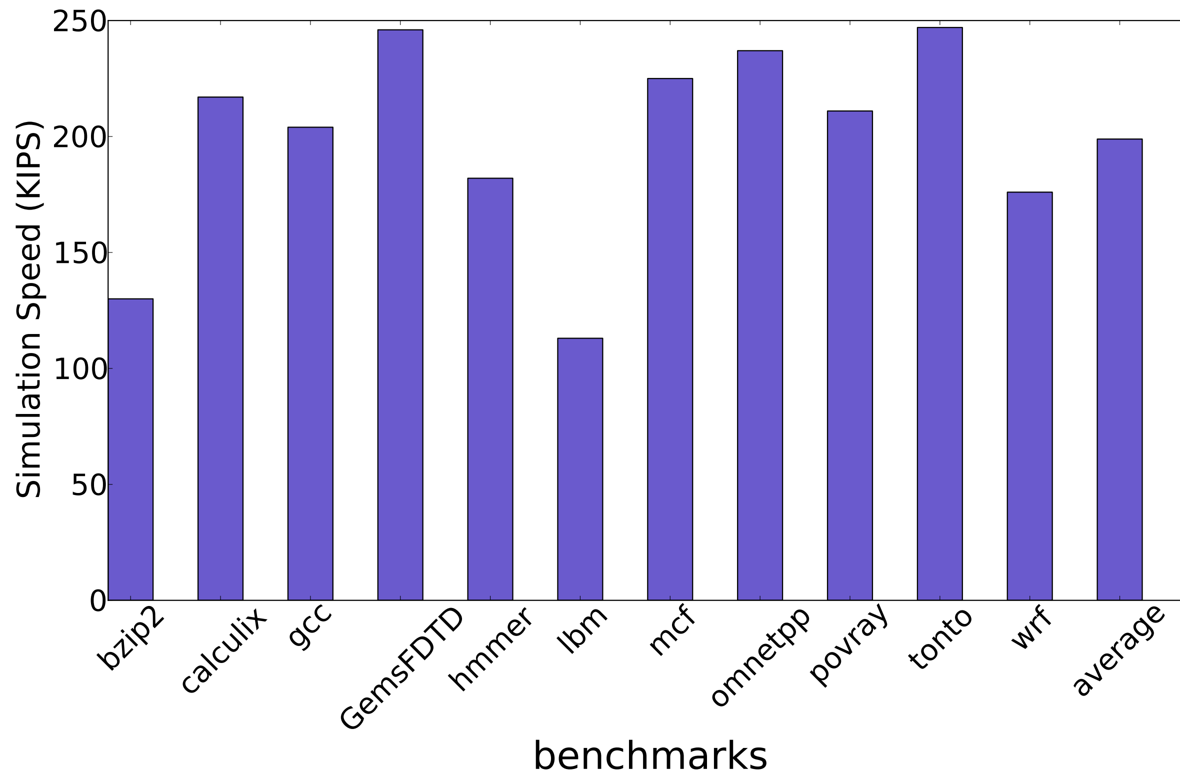

Fig. 11: OSX platform – Simulation speed